Dissertation

PERFORMANCE DRIVEN GLOBAL ROUTING FOR LARGE SCALE
MACRO-CELL BASED DESIGNS

Submitted by

Cengiz ALKAN

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall, 2005

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 3226108

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3226108

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

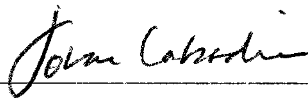
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

COLORADO STATE UNIVERSITY

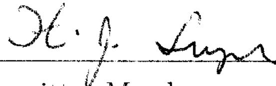
July 7, 2005

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY CENGIZ ALKAN ENTITLED PERFORMANCE DRIVEN GLOBAL ROUTING FOR LARGE SCALE MACRO-CELL BASED DESIGNS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work



Committee Member



Committee Member

Committee Member

Adviser

Department Head

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTARCT OF DISSERTATION

PERFORMANCE DRIVEN GLOBAL ROUTING FOR LARGE SCALE MACRO-CELL BASED DESIGNS

Technology scaling to the sub-micron feature sizes for complementary metal oxide silicon (CMOS) devices has led to decreased device cost and increased performance, but it has singled out interconnects which do not scale well. Such rapid scaling has introduced new challenges for CMOS interconnects. The main challenge is to support interconnect optimization with minimum performance cost. Thus, new design methodologies, algorithms as well as the integration of new materials, are needed to achieve the design closure.

The ability to concurrently synthesize interconnects and logic to achieve the best overall solution is the key for next generation of very large scale integration (VLSI) designs. The traditional sequential design flow implies that block design be completed prior to routing. Routing and timing failures often require changes to the block design in the early stages of the design flow. Consequently, any change to the block design will have an impact on the routing. This expensive design loop may converge slowly or may not converge to a desired solution at all. To allow a net-centric design methodology, design of interconnects needs to be an integral part of the overall design flow. Routing-driven/aware design methodologies also contribute positively to the quality of the final solution.

As design rules scale further down to the deep sub-micron region, achieving the design closure will get harder due to a variety of emerging problems:

- Congestion management is critical since a detour around congested areas as well as the possible coupling in congested areas has a negative impact on delay.

- More accurate delay analysis is needed to adequately address the nanometer effects.

- Layer assignment has to be done with tighter integration of performance-driven routing-tree construction and buffer insertion to obtain an optimal routing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Faster and more efficient algorithms are needed to accomplish the required tasks in reasonable time.

Current methods attack the above issues separately in a sub-optimal fashion. A greedy sequential router or routing net-by-net suffers from slow convergence and does not scale well with design complexity. There is a certain degree of uncertainty and unpredictability to the final routing result since its result is net order dependent. A multi-commodity flow router attempts to solve this net ordering problem by routing all the nets concurrently, but often fails to minimize congestion. More importantly, it suffers from long run-time due to the size and complexity of the optimization model. routng-tree construction algorithms also suffer from sub-optimality by considering timing-driven Steiner-tree construction, layer assignment and buffer insertion separately.

Addressing these issues due to the complexity of the overall global routing problem requires careful planning of runtime-quality trade-offs. In this dissertation, a new global routing solution to address the forementioned issues is implemented and presented. The approach to the global routing problem was to construct a performance-driven routing-tree simultaneously considering layer assignment and buffer insertion, and then optimizing the congestion, within the bounding box of the net to meet the projected delay requirements, using a mixed-integer programming routing model and a novel network-flow model. The global routier utilizes a new performance-driven buffered routing-tree construction algorithm, a mixed-integer programming pre-routing stage and a new network-flow routing model to complete the task.

In the first stage, we present a routing-tree construction algorithm that considers multi-objectives of performance, power and congestion concurrently. In contrast to the traditional algorithms which assume underlying routing-tree already exists, a concurrent Steiner-tree construction, layer assignment and buffer insertion algorithm is used to obtain a better overall result in terms of delay violations and routing resources. As routing-trees are being

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

constructed, critical nets are assigned to the appropriate metal layers and/or buffers are inserted to achieve a positive slack at sinks. Congestion is measured with balanced usage of routing resources among layers. An accurate delay calculation engine using an asymptotic waveform evaluation model is used to obtain delay violations at sinks. Simultaneous buffer insertion and layer assignment tends to produce routing-trees with shorter overall length.

After synthsizing the rouitng-trees, We use a three-stage global-routing algorithm based on mixed-integer programming and a novel network-flow model. While the performance tuned routing-trees provided by our Steiner-tree construction algorithm are routed by mixed-integer programming model considering the congestion on global level, the network-flow model based router performs more localized congestion optimization. Furthermore, we introduced various methods to improve the routing quality sin terms of cross-talk and via count.

The second stage provides a rough two-bend routing and manages congestion at the global level. This is achieved by utilizing zero-slack values of mixed-integer programming to determine if a routing segment can be minimized further. Since routing and congestion information was not known to the first stage, some nets might fail to route. A layer assignment algorithm based on quadratic programming is used to further tune the layer assignment.

In the third stage, we center on the congestionon optimization and via reduction by bisecting the layout hierarchically for each routing layer. A novel network-flow model, which can be solved hundreds of times faster than a mixed-integer programming model, is used to minimize the congestion through a cut line. Similar to the second stage, zero-slack flows are used to determine which routing segment can be further optimized. Via reduction is achieved by assigning cost values to the network arcs such that bends have higher cost.

In the fourth stage, a series of heuristics such as edge flipping, detour and a maze router are used to clear the remaining overflows and via pad violations.

The results from various test cases including a subset of the routes on a commercial

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

64-bit microprocessor core show that our method outperforms commercial CCT router. On average, we achieved 29% less delay violations, 40% less repeater usage on the resulting routing-trees, 20% less maximum delay violation and better congestion distribution.

Cengiz ALKAN
Electrical and Computer Engineering Department
Colorado State University
Fort Collins, CO 80523
Fall, 2005

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I would like to express my thanks to my advisor, Dr. Thomas Wei Chen for his encouragement, support, and dedication. His invaluable assistance on developing, writing and presenting this work is greatly appreciated. I would like to thank my committee members for their invaluable adivice and assistance. I want to thank Bruno Melli at Hewlett-Packard (HP) for his assitance on developing the code and providing the maze router source code. My warmest thanks go to VLSI lab. crew for their support and friendship. My special thanks go to my son, Bera ALKAN, and my family, for their patience, and thier support during difficult times.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

1 Introduction **1**

1.1 Background 1

1.1.1 Global Routing 1

1.1.2 Silicon Technology Challenges and Global Routing 2

1.2 Problem Definition 5

1.3 Research Objectives 7

1.4 Organization of the Dissertation 8

2 Existing Work **9**

2.1 Introduction 9

2.2 Delay Modeling 10

2.3 Layer Assignment 16

2.4 Buffer Insertion 23

2.5 Steiner-Tree Construction 28

2.6 Global Routing Algorithms 37

2.7 Conclusions 48

3 Global Routing (GR) **54**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 Introduction 54
3.2 Performance-Driven Buffered Routing-Tree with Layer Assignment . . 59
3.2.1 Routing-Tree Construction 60
3.3 Buffer Insertion and Removal 67
3.4 Layer Assignment 70
3.5 Simultaneous Performance-Driven Buffered Routing-Tree Construction with Layer Assignment 74
3.6 Two-Bend Global Routing with MIP, A Pre-Router 75
3.6.1 MIP Routing Model 79
3.6.2 Blockage Handling 80
3.6.3 MIP Optimization 82
3.7 Network-Flow Routing 87
3.7.1 Network-Flow Routing Model 89
3.7.2 Via Minimization 92
3.7.3 Blockage Handling 93
3.7.4 Network Optimization 94
3.8 QP Layer Assignment Tuning 97
3.8.1 Introduction 97
3.8.2 QP Model 98
3.9 Post Processing Heuristics 100
3.9.1 Introduction 100
3.9.2 Edge Flipping 100
3.9.3 Detour 101
3.9.4 Via Pad Elimination 101

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.9.5 Maze Router 102
3.10 Conclusions 104

4 Experimental Results 105

4.1 Simultaneous Buffered Routing-Tree Construction with Layer Assignment Results 107
4.2 GR Results 109

5 Concluding Remarks and Future Work 116

5.1 Concluding Remarks 116
5.2 Future Work 118

A Abbreviations and Acronyms 128

B Technology File Format 130

C Source Code 133

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

1.1 Delay for Metal 1 and Global Wiring versus Feature Size 3

2.1 RC Network for Elmore Delay Model 11

2.2 Two-Path Hybrid Ladder Model 12

2.3 RL Model for Table-Lookup Delay Model 12

2.4 Delay Model Using Moment Matching 14

2.5 Switching Graph 17

2.6 Performance-Driven Routing of Interconnect Trees 18

2.7 Routing Cost with Distance 19

2.8 Routing Demand can Vary Widely Depending on Tree Topology . . . 20

2.9 Potential Crosstalk 21

2.10 Alternate Steiner Point to Avoid Blockage 24

2.11 Interconnect Model for QP Formulation 25

2.12 Buffer Insertion Can Reduce Wire Cost Further in Steiner-Tree Construction 27

2.13 1-Steiner Algorithm 29

2.14 Decomposition and Reconnect of Sub-Trees 30

2.15 Sub-Tree Transformation 31

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.16 An Example Steiner-Tree Topology 32
2.17 LP Model for Bounded Steiner-Tree Construction 33
2.18 V Removal 34
2.19 U Removal 34
2.20 Progressive Steiner-Tree Construction 35
2.21 Optimal Delay Non-Hanan Steiner Point 36
2.22 Grafting 37
2.23 A Routing Grid Graph 39
2.24 Buffer Planning Graph 39
2.25 Shift and Transition Times 40
2.26 Congestion Measure 41
2.27 Approximation algorithm for fractional global routing 44
2.28 Bisection 46
2.29 An Assignment 47
2.30 Network Formulation without SSN 48
2.31 Network Formulation 49
2.32 Hierarchical Bisection Method Fails to Avoid Blockage 50
2.33 Hierarchical Decomposition 51
2.34 Slots and Virtual Terminals 51
2.35 Bottom-Up Rerouting 52

3.1 Global Routing Flow Chart 58
3.2 Steiner Trees 60
3.3 Distance for Steiner-trees 61
3.4 Performance driven progressive routing tree construction 62

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5 Progressive Steiner-tree Construction 64
3.6 Hanan Point 65
3.7 Possible Join Strategies 66
3.8 Non-Hanan Optimization 67
3.9 Iterative Methods Modifying the Existing Tree may Invalidate the Buffer Insertion 67
3.10 Delay of an interconnect 68
3.11 Candidate Buffer Locations 68
3.12 Buffer Insertion 69
3.13 Performance-Driven Layer Assignment 71
3.14 Layer Assignment 72
3.15 DV sort list for layer assignment 73
3.16 Simultaneous RT Construction 75
3.17 Simultaneous Performance-driven Buffered Routing-Tree with Layer Assignment 76
3.18 Simultaneous buffered routing tree construction with layer assignment 77
3.19 Two-Bend Routing Flexibilities 79
3.20 Effective Capacity of a Routing sSegment is Reduced by Blockages . . 80
3.21 Suboptimal MIP routing 83
3.22 Iterative MIP Optimization 84
3.23 Pre-Routing for Congestion Optimization 88
3.24 Suboptimal Network Routing 89
3.25 Hierarchical Bisecting and Network-flow model 90
3.26 Network-flow routing with pre-router 91
3.27 Via Minimization 93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.28 Network-Flow Routing Model w/Blockage Handling 94
3.29 Network-flow Congestion Optimization 96
3.30 Potential Cross-talk 98
3.31 L Flipping 100
3.32 Detour 101
3.33 Via Pad Violation 102
3.34 Ripping Scheme for the Maze Router 103
3.35 Maze Routing 103

4.1 Congestion Distribution 113

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

4.1 Sequential and Simultaneous Routing-Trree Construction Results . . 109

4.2 Wire Distribution 109

4.3 Comparison of Our Method with CCT 112

4.4 Comparison of Our Method with CCT (cont.) 114

4.5 Flow Distribution 115

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

1.1 Background

1.1.1 Global Routing

Laying out a very large scale integration (VLSI) chip is a process of placing and interconnecting a set of modules and multi-terminal nets. After placement, the routing region is often partitioned into tiles by global routings (GRs) to find a path (a sequence of tiles) for each net. The detailed routing (DR) then finds an exact path dictated by the GR results. As seen from the chip layout process, VLSI chip routing is done in two stages, the GR and the DR. A GR focuses on optimizing the various aspects of interconnects toward a successful DR. The objectives of the GR mainly consist of obtaining the timing closure, routability (congestion optimization), cross-talk minimization, power minimization, layer utilization and the buffer insertion. While achieving the timing requirements of a net for the projected clock frequency is a major requirement for GR [1], the other aspects, such as congestion optimization, power

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

minimization, repeater insertion and cross-talk, are important for the final routing quality.

GR is essentially an abstract routing stage before the actual routing. In general, it is achieved by partitioning the layout into smaller tiles and then assigning a path for each wire such that there is no overflow on the tile boundaries and projected net properties, such as delay, are still in acceptable limits. This loose wiring is then used to guide the subsequent detailed wiring. To obtain the best overall result, various technologies are employed along the way. To reduce the delay and crosstalk, buffer insertion [2, 3, 4, 5, 6], layer assignment [7, 8, 9, 10, 11], and performance-driven routing-tree (RT) construction [12, 13, 14, 2, 15, 16, 17] are used. To improve the routabilty, congestion estimation [9, 7, 18, 19] is used in GR. Most of these technologies are inter-dependent; the result of one greatly impacts the success of another. For example, poor congestion optimization forces nets to detour increasing the delay of the net in the final solution.

The following subsection provides an overview of the current technology challenges and their relation to the GR.

1.1.2 Silicon Technology Challenges and Global Routing

The scaling to the deep sub-micron (DSM) feature sizes for the complementary metal oxide silicon (CMOS) VLSI has led to decreased device cost and increased performance. But, starting from the $250nm$ process technology node, the interconnect delay dominates the gate delay [7] due to poor salability of interconnects as illustrated in Figure 1.1. The reason behind this is that interconnects do not scale as well as gates [20, 21, 22]. Calculations show that using the values from International Tech-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

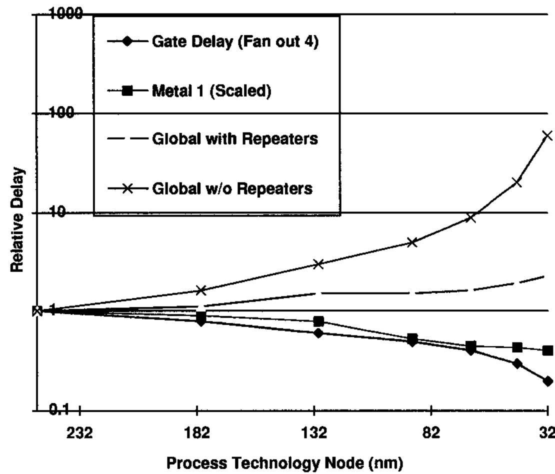


Figure 1.1: Delay for Metal 1 and Global wiring versus feature size [1]

nology Road Map for Semiconductors (ITRS 2003) for technology generations from 180 nm down to 15 nm, the delay of scaled wires increases by approximately 10 times while the delay of fixed length wires increases by 2000 times [1]. The performance of future VLSI chip is no longer determined by glogic ates but rather by interconnects. Consequently, such rapid scaling requires dramatic changes in the design flow and in routing tools. RT synthesis is an icreasingly important part of interconnect design. Although the integration of the new materials such as copper/low-k provides some relief, it is now widely accepted that technology itself cannot solve the interconnect problem, interconnect-centered architectures and tools must come together to provide an optimized integrated system [1]. At the center of this integrated system, the GR plays an important role.

The major challenge of GR is to meet the delay requirements of critical nets [1]. Hence the timing closure requires performance-driven routing schemes employing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

timing-driven Steiner-tree (ST) construction, buffer insertion, and layer assignment [23, 6, 5, 24, 12, 25]. In addition to the delay optimization, today's high density devices present an additional challenge to the GR in terms of congestion optimization [18, 9, 7, 19, 26] due to increased chip density. Congested areas are hard to route and often cause detours for the critical nets. Congested areas are also more vulnerable to coupling between nets which in turn has a negative impact on delay. Highly congested routes are detrimental to DFM since well known DFM schemes, such as double via and wire spreading, require additional space. Hence, one of the objectives of a good GR algorithm is to route more evenly among routing layers while trying to meet the performance requirement.

Efficient buffer insertion and layer assignment algorithms are also among the critical challenges. A typical VLSI chip requires tens of thousands of buffers to be inserted to meet the timing requirements of critical nets. These buffers can greatly increase the power consumption by more than 30% [1]. Yet, buffer insertion is a key technology to reduce the delay and crosstalk. Today's macro-cell over-the-cell routing uses multiple routing layers. These routing layers vary in parasitic capacitance, resistance, width and height. The RC constant can be 10 times different between lower metal layers and upper metal layers. Therefore, efficient utilization of layers can also greatly improve the performance of the nets.

The forementioned issues are tightly inter-dependent. They compete for the limited routing resources. A GR has to simultaneously optimize the competing requirements and objectives for the best overall result. Due to the complexity of the overall problem, addressing these issues requires careful planning of runtime-quality trade-offs in GR.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 Problem Definition

A GR problem can simply be defined as: a set of given nets and their pins and timing requirements for the nets, find a path for each net such that all pins are connected. Furthermore, delay from any source pin to sink pins for a net is less than the required arrival time (RAT) and there is no over use of any routing resource (overflow). A GR also inserts buffers along paths when the RAT or the required signal slew rate cannot be met. Furthermore, the metal layer assignment of the paths can be changed to improve the delay for the critical nets. Hence, routing a high performance VLSI chip depends on the following tasks: *routing-tree construction, buffer insertion, layer assignment and congestion optimization with the objectives of no delay violation, routability, lower power, less via and crosstalk.*

The choice of RT construction algorithms directly impacts delay. The well known rectilinear ST [12] is usually employed as the RT. It is well suited for VLSI routing due to the fact that routing a layer is usually restricted to one routing direction (Manhattan routing). The minimum length ST problem is known as NP-hard [12]. Therefore various heuristics are used [12, 14, 15, 17, 27, 16, 13]. Minimum length ST algorithms ignore the fact that the length from source to sink is not necessarily minimal. ST algorithms minimizing the length from the source to the sinks [16] still does not guarantee minimum load along the critical source-sink pair. Therefore, RT construction must consider minimizing the maximum delay violation (performance-driven) [13, 15, 14, 2, 17].

When a net's required timing can not be obtained with a performance-driven RT construction, buffers are inserted [2, 3, 6] along paths to reduce the delay on the critical path. Buffer insertion helps reduce the delay in two ways:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. by breaking the path into smaller isolated paths.

2. by reducing the load on the critical source-sink path.

The number of buffers and buffer locations are important to keep the power consumption within an acceptable level while still satisfying the RAT at the sinks.

In a multi-layer environment, routing layers can be used to improve the performance of the critical nets by assigning those critical nets to the wider metal layers. Since routing resources are limited on a particular layer, the layer usage among available layers has to be optimized. This optimization also helps reduce the congestion by not crowding any metal layer. There are usually two different approaches to layer assignment. The first approach is a post-process step with a constrained layer assignment problem; where two-dimensional global paths are first defined, then wires are assigned to layers to optimize the objectives such as delay, crosstalk, routability and via count [8, 10, 11]. The second approach is a pre-process step using interference analysis without actual routing [7]. In this approach, interference [7] between a pair of nets is estimated and used towards an initial layer assignment.

Finally, the congestion optimization aims to distribute interconnects homogeneously throughout the die. Possible detours and the coupling are reduced by the congestion optimization algorithms. Congestion optimization is usually bounded by delay; one can only minimize congestion on a routing resource if the affected nets will not violate the timing requirements.

The main challenge is to implement the forementioned tasks, concurrently for the best overall result, with minimal impact on the performance and run-time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3 Research Objectives

The objective of this research is to develop new GR techniques for high performance VLSI chips that can optimize congestion and meet timing requirements. The advantages of the concurrent implementation of RT construction, layer assignment and buffer insertion algorithm is explored. Also, the result of an aggressive optimization of the congestion is compared to the traditional algorithms which only focus on eliminating overflows.

Our method is as follows: first, a simultaneous performance-driven buffered RT construction with layer assignment algorithm is used to obtain the underlying RT. An accurate delay calculation method using asymptotic waveform evaluation (AWE) is integrated for better delay estimation. Second, a mixed integer problem (MIP) model is used to optimize the congestion globally for given RTs. An aggressive optimization technique is used to minimize the congestion for each tile boundary rather than just eliminating the overflow. The rough two-bend routing provided by the MIP model is further optimized by bisecting the die and using a novel network-flow model in the third stage. A via minimization technique is also introduced for the network model. Since the routing and the congestion information was not available during the RT construction in the first stage, sub-optimal layer assignments are further tuned using a quadratic programming (QP) model integrated into the second and the third stages. Finally, the last stage uses a series of heuristics (edge flipping, detour and maze router) to clear up the remaining routing failures.

Our contributions to the GR are as follows:

- A better performance-driven RT construction by simultaneously considering ST construction, layer assignment, and buffer insertion

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- A pre-router based on a simplified MIP model to optimize congestion globally and overcome the sub-optimality of hierarchical routing.
- A hierarchical routing based on a novel network-flow model to reduce run-time.
- A new congestion optimization method for the MIP model and network model based on zero-slack values in order to homogeneously distribute the congestion throughout the die.
- Introduction of via reduction method in the network-flow routing model.

1.4 Organization of the Dissertation

The organization of the dissertation is as follows: In Chapter 2, existing GR, layer assignment, delay modeling, buffer insertion and ST construction algorithms are reviewed. Chapter 3 first explains the three techniques, performance driven ST construction, layer assignment, and buffer insertion, separately, and then combines them into one concurrent RT construction algorithm. Section 3.6 of Chapter 3 presents a new congestion optimization technique using the two-bend MIP modeling. Hierarchical bisection and further optimization of the congestion by using the network-flow model is explained in Section 3.7. Section 3.9 presents heuristics for edge flipping, detour and maze router, used to clear the final routing failures. Experimental results are given in Chapter 4. Finally, the concluding remarks and future work suggestions are given in Chapter 5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Existing Work

2.1 Introduction

This chapter reviews the existing works and their shortcomings on GRs. A GR algorithm employs a series of steps to complete the task. A layer assignment algorithm is used to assign the tree edges to the layers for performance improvement of the critical nets and for cross-talk minimization. Naturally, the delay modeling is necessary for the timing-driven GRs to accurately calculate the delay and slack at sinks. A buffer insertion algorithm is also used to meet the timing requirements for the projected clock. Finally, a GR algorithm itself lays the actual wires using congestion estimation in addition to the mentioned steps.

The following sections categorize and overview the existing GR algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Delay Modeling

In order to adequately account for the nanometer effects in GRs, accurate delay models must be used during delay calculation. The lumped delay model [28] and Elmore delay model [29] are no longer sufficient due to the resistance shielding and increased inductance effect for high speed VLSI circuits in nanometer feature sizes. Unfortunately, many algorithms [30, 31, 32, 33, 16, 3] rely on the most popular Elmore delay model because of its simplicity. Some algorithms also use look-up tables [34] and simulation engines [35] to accurately calculate the delay. An obvious drawback of the look-up tables is that they are technology dependent. The look-up table has to be regenerated to accommodate different technology nodes. Moreover, the number of parameters needed to accurately model the nanometer effects can significantly increase the table size. A simulation engine such as SPICE provides great accuracy, but it is very costly in terms of run-time. Today's complex routing algorithms need millions of delay calculations. Therefore, this many calls to a simulation engine can't be afforded. The more accurate analytical models are based on moment matching [36, 37, 35]. Run-time and memory requirement of the moment matching techniques are considerably high due to their computational complexity. Fortunately, various techniques have already been explored to improve both run time and memory requirements of the moment matching techniques.

Elmore delay model is the only analytical delay model providing a closed form of the delay. Unfortunately, when resistance shielding and inductance are in effect, it is not accurate enough. Elmore delay can be easily calculated at an output node i for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

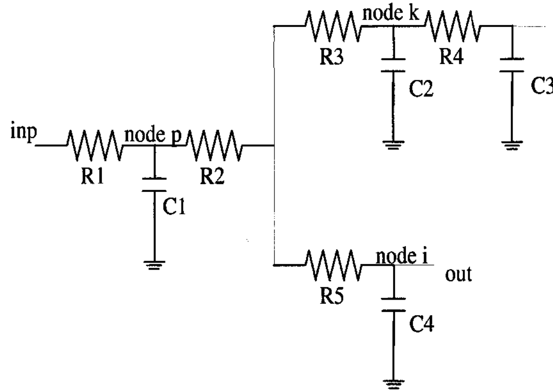


Figure 2.1: RC network for Elmore delay model

a given RC network in Figure 2.1 using:

$$T_i = \sum_k R_k C_k \tag{2.1}$$

Where R_k is the resistance along the path from input node to the output node and C_k is the lumped down stream capacitance from node k. Elmore model provides a simple closed form of delay calculation, but it overestimates the delay. Therefore, routing algorithms using Elmore model tends to overuse routing resources. For example, A buffer insertion algorithm can insert more buffers than necessary, thus increasing the overall power and possibly the routing area of the chip.

In order to account for various parameters that affect delay and can't be calculated easily by analytical models, Look-up tables are used to estimate the delay. A look-up table uses a simplified RLC circuit to model the various parameters used to calculate the delay. Hu [34] proposed a compact timing and noise analysis model using the two-path hybrid ladder model in Figure 2.2. The two-path hybrid ladder model includes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

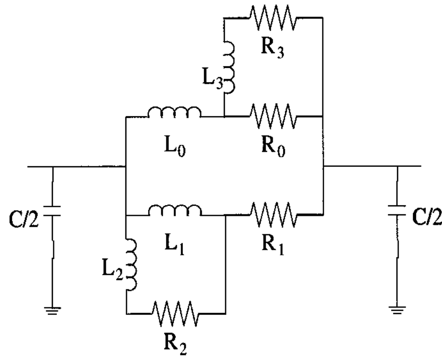


Figure 2.2: Two-path hybrid ladder model

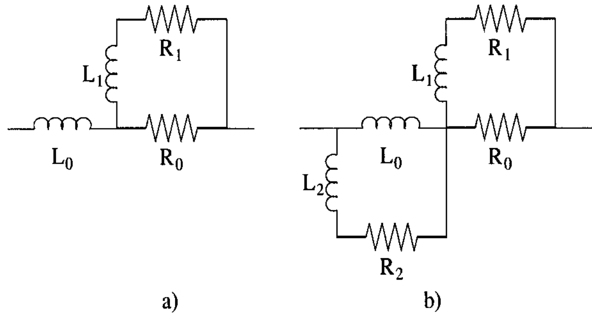


Figure 2.3: a) The RL ladder circuit. b) The hybrid ladder model

components which compensate for higher and lower frequencies. The structure in Figure 2.3 (a) is known to be adequate for lower frequencies and the structure in Figure 2.3 (b) attempts to perform high-frequency compensation through R_2 and L_2, but R_0 and L_0 are required to be involved in both the high-frequency and the low-frequency. With a simplified approach, each entry in the table corresponds to a set of the following values:

1. The metal layer on which the signal line lies.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. The width of the switching lines.

3. The length of the switching lines.

4. The distance to the nearest supply grid line

5. Shielded and unshielded cases.

A look-up table can be a very quick and effective way to calculate the delay but is not easily applicable to RTs with more than two pins. Moreover, the number of parameters needed in nanometer timing analysis can easily aggregate the table size.

Higher order delay calculation techniques are preferred for their accuracy by today's chip design tools. Pillage [36, 37], proposed a moment matching technique, asymptotic waveform evaluation (AWE), to accurately calculate the delay. The moments of a transfer function of a interconnect from expanding the transfer function into Taylor series around $s = 0$ is given by:

$$H(s) = \frac{1 + a_1 s + a_2 s + \cdots + a_m s}{b_0 + b_1 s + b_2 s + \cdots + b_n s} = m_0 + m_1 s + m_2 s + \cdots \tag{2.2}$$

To illustrate the relation between moment and poles/residues, $H(s)$ can be written:

$$H(s) = \frac{k_1}{s - p_1} + \frac{k_2}{s - p_2} + \cdots + \frac{k_n}{s - p_n} \tag{2.3}$$

By expanding each term, moment can be expressed as:

$$\begin{aligned} m_0 &= -(\frac{k_1}{p_1} + \frac{k_2}{p_2} + \cdots + \frac{k_n}{p_n}) \\ m_1 &= -(\frac{k_1}{p_1^2} + \frac{k_2}{p_2^2} + \cdots + \frac{k_n}{p_n^2}) \\ m_{2n-1} &= -(\frac{k_1}{p_1^{2n}} + \frac{k_2}{p_2^{2n}} + \cdots + \frac{k_n}{p_n^{2n}}) \end{aligned} \tag{2.4}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

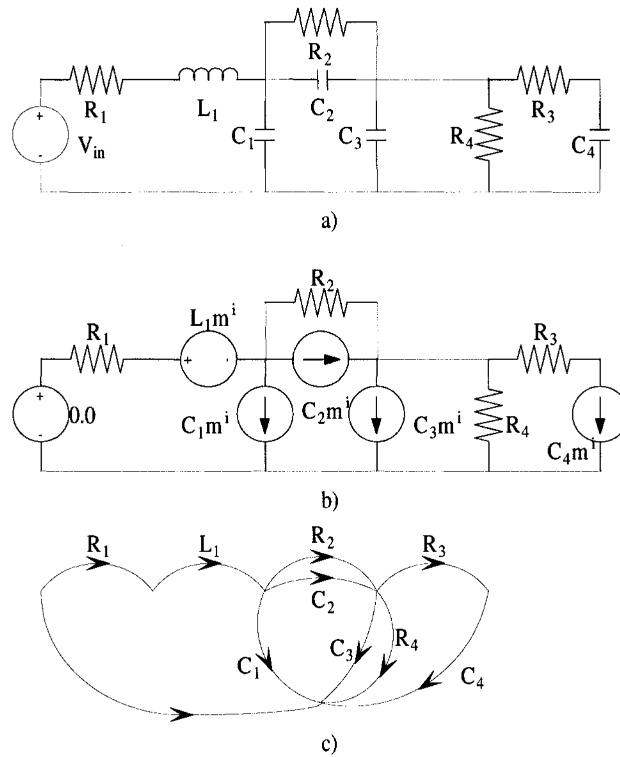


Figure 2.4: a) An RLC example circuit. b) the dc equivalent. c) and the directed graph

Equation 2.4 stresses the dominant poles with smaller magnitudes. Therefore, these dominant poles are of most interest when evaluating timing. The moments around $s = 0$ can easily be calculated for tree structured interconnects. The basis of the AWE is to estimate a circuit's time-domain response by matching the initial boundary conditions and first $2q-1$ moments of the exact response to a lower order q-pole model. The moments of a circuit in Figure 2.4 (a) are computed in AWE by recursively solving an equivalent DC circuit as illustrated in Figure 2.4 (b) with all capacitors replaced by current sources and all inductors replaced by voltage sources. Initially, all sources are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

set to zero except the independent input voltage. This is used to calculate the initial set of moments. For subsequent moment generations, each capacitor current source is set to the product of its capacitance and its previous moment, while each inductor voltage source is set to the product of its inductance and its previous moment.

AWE [36] technique's efficiency can be improved by exploiting some structures. Curtis [35] introduced their rapid interconnect circuit evaluator (RICE) based on an improved AWEsim [32] technique. RICE exploits tree-like structures, which is the most common for interconnects, to obtain linear speed-up and to reduce the memory requirement. The most important aspect of RICE is that it uses the path-tracing technique to analyze the equivalent DC circuit. The path-tracing simply analyses the circuit by traversing a spanning tree in Figure 2.4 (c) of the circuit graph to compute currents and voltages as shown in Figure 2.4 (b).

Moment matching process by which moments are determined does not allow calculating the moments at a few selected nodes of a circuit. Moments have to be calculated at all nodes since $(i+1)$th moment at any node depends on the (i)th moment at all nodes. However, the AWE technique [36, 35] only uses the moments at a single node at a time to calculate the response at that node, hence, called single-point moment matching (SMM). The SMM techniques calculate more moments to improve accuracy. Ismail [38] showed that accuracy can also be improved by using the information in the moments at different nodes simultaneously, hence, called multi-point moment matching (MMM). The reason MMM uses less moment is that it exploits the fact that there is a common set of poles at all the nodes of a circuit. By only considering a single node at a time, SMM requires $2q$ moments to solve for $2q$ variables. However, by adding more nodes in MMM, the number of variables does not increase by $2q$ for each extra node. Since the q poles are common to all the nodes, adding an extra node

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

only adds q new variables. Hence, MMM need only to match $q(q+1)$ moments which are $q+1$ moments at q nodes. Using $q+1$ moments instead of $2q$ moments not only doesn't reduce accuracy, but also improves memory requirements of the algorithm. The advantages of MMM over SMM are as fallows:

- Number of moments required for the same accuracy is significantly lower than SMM.
- MMM has much better numerical stability compared to SMM.
- MMM is highly suitable for parallel processing.

In conclusion, length, linear delay model and Elmore delay model based algorithms are no longer sufficient to construct the performance-driven RTs. Many GR algorithms [30, 31, 3, 6, 39] use Elmore delay model for it's simplicity and closed form to compute the delay which may differ greatly from the actual delay. Since DSM interconnects are much thinner and longer, downstream capacitance is shielded by the interconnect resistance. Elmore delay does not take resistive shielding into account accurately, resulting in remarkably large overestimated errors. More accurate timing model such as AWE must be used to accurately calculate the delay in DSM VLSI chip routing.

2.3 Layer Assignment

In a multi-layer routing environment, layers have very different physical and electrical properties. These layers are not interchangeable, resulting in many wiring prediction problems. Thus, the layer assignment has a large impact on the interconnect delays. In multi-layer CMOS processes, the higher metal layers (i.e., layers further away from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

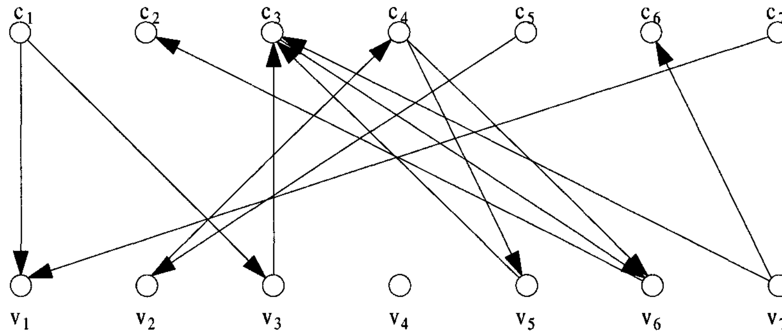


Figure 2.5: Switching Graph

transistors) tend to be wider and ticker. Signal transmissions on these layers will be faster than those on the lower metal layers. We, therefore, refer the higher metal layers as *good layers.* A layer assignment algorithm assigns critical nets to the good layers while preventing excessive use of the good layers to reduce the congestion. This technology is also widely used to reduce crosstalk between nets.

Thang [10] proposed a method based on a genetic algorithm (GA) with application to the via minimization. The GA is used to solve the switching graph model as shown in Figure 2.5, which in-flow and out-flow of a vertex equals to the number of vias introduced in the corresponding layer assignment. Thang's algorithm is designed as a post-processing algorithm to reduce the number of vias and does not consider routability or timing; the resulting RTs may not satisfy timing requirements and/or be routable due to congestion. Although the algorithm can be restricted to only feasible layers, which satisfies routability and timing for the given tree, determining routabilty on the candidate layers can be costly and inaccurate.

Layer assignment is not only used for via reduction but also for performance improvement of the critical nets. Saxena [11] proposed a layer assignment algorithm to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

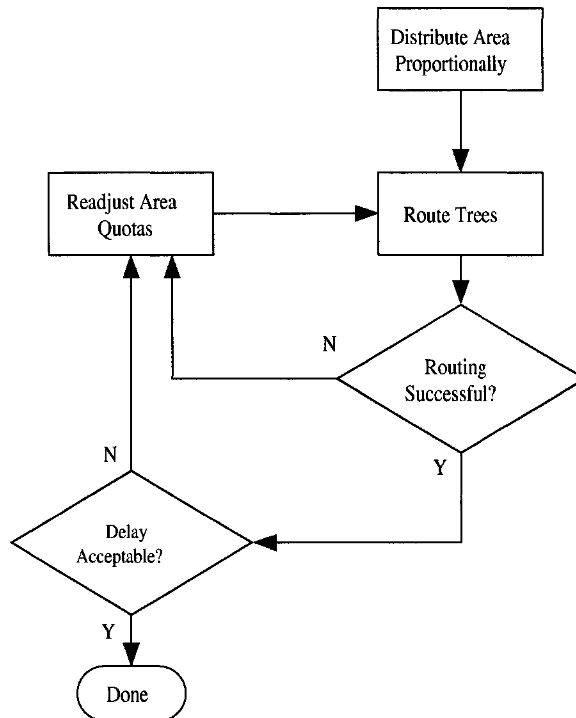


Figure 2.6: Performance-driven Routing of Interconnect Trees (PRIT)

minimize the peak interconnect delays during GR. Their algorithm can be integrated into any routing algorithm to iteratively assign the layers using area quotas on the routing layers. Saxena's algorithm initially distributes all the nets proportionally among the layers (same area quota for each layer), then if routing fails or achieved delays are not acceptable, it transfers a certain area quota from one layer to another. The flow of the Saxena's algorithm is illustrated in Figure 2.6. A look-ahead key represented by Equation 2.5 is used to decide if the edge being routed should be routed in the current routing layer or should be postponed to the next layer. The look-ahead key is simply derived from the difference between the contribution of the edge to total tree delay when the edge is routed in the current layer and when the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

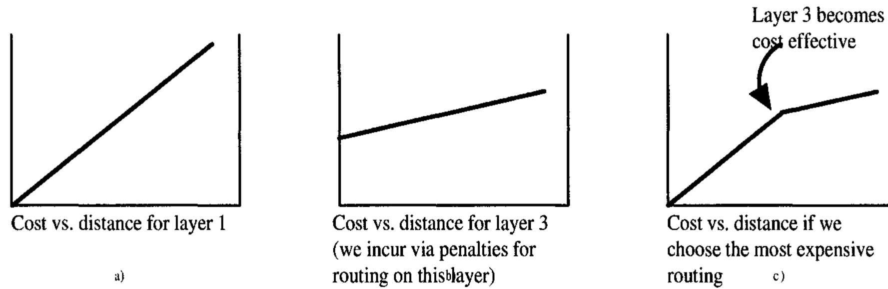


Figure 2.7: Routing cost with distance

edge is routed in the next layer:

$$\Delta_{k_i}(e_i) = \theta_{k_i'}(e_i) - \theta_{k_i}(e_i) \tag{2.5}$$

k_i and k_i' in Equation 2.5 are the current and the next layer respectively; $\theta(e_i)$ is the contribution of edge e_i to the delay expression. As seen from the definition of the look-ahead key, the routing is done layer by layer and edge by edge. Thus, it is not suitable for the parallel routers but only for the sequential routers.

Yildiz [8] extended planar rectilinear ST heuristic [40] to include the layer assignment and via minimization. This algorithm uses layer specific costs to calculate a *cost vs. distance graph* and then determines which layer is cost effective for the current edge. The upper layers become cost effective for long wires while the shorter wires are routed on lower layers due to the additional cost of vias. As shown in Figure 2.7, layer 3 becomes cost effective for routing edges of certain length after certain distance while layer 1 is preferred for shorter edges since via cost incurred in layer 3 dominates the length cost. Although Yildiz's algorithm assigns shorter nets to the lower layers and longer nets to the higher layers which is generally desirable, it is not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

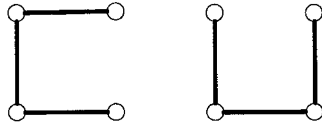


Figure 2.8: Routing demand can vary widely depending on tree topology a) Higer horizontal layer demand b) higher vertial routing demand

really performance driven layer assignment, tight timing requirements may require shorter wires to be assigned to the upper layers and vice versa. It is conceivable that a short net can also be critical. Furthermore. it can result in over-crowded layers since it doesn't consider congestion. Layer assignment should be done by performance requirements, not by length alone.

Preferred STs [8] can result in excessive use of certain layers creating congested areas. Ameya [9] has combined the preferred STs [8] with ST construction [40] to reduce the congestion by layer balancing. Ameya observed that the ratio of horizontal to vertical demand can be as much as 70:30. Thus, the algorithm focus on modifying the tree topology without increasing tree length for structures shown in Figure 2.8. The steps of the algorithm are as follows:

1. Routing cost for all layers is set to initial default values.

2. Preferred STs are constructed

3. Resource usage is examined; if a layer is over utilized, the cost of that layer is increased slightly and step 2 is repeated.

Layer assignment is not only performance-driven but also cross-talk and congestion driven. Cho [7] proposed a pre-routing process to minimize the crosstalk. A potential crosstalk measure in Equation 2.6 is used to model the problem as a net-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

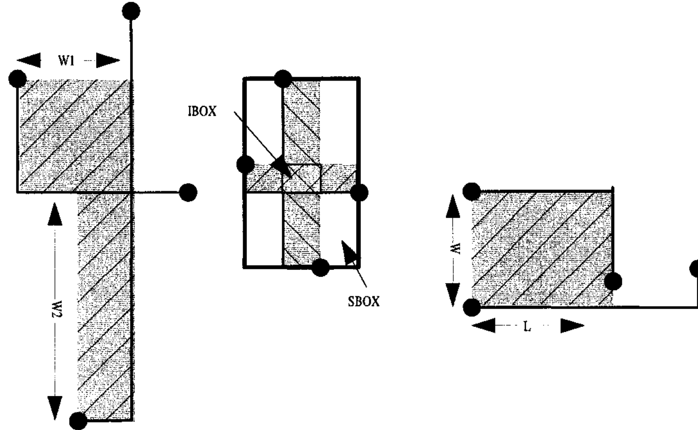


Figure 2.9: Potential Crosstalk

work interference graph (NIG) as modeled in Equation 2.7. The potential cross-talk measure is based on the assumption that a larger intersection area of two nets makes it harder to find non-intersecting routes for the nets:

$$w_{i,j} = \frac{(cong_i + cong_j + nterms_{ibox})area_{ibox}}{area_{sbox}} \tag{2.6}$$

where $cong_i$ is the number of nets who's bounding boxes intersects with net_i, $ibox$ is the intersecting box and $sbox$ is the bounding box as illustrated in Figure 2.9. The vertices of the NIG must be colored using the minimum number of colors, such that no two adjacent vertices receive the same color. This problem is known to be very hard and is impractical to try to find a perfect solution. Since the edges in the NIG are the potential interference, a small number of bad edges can be tolerated. Thus, the NIG can be solved with *max cut k-color partitioning* where each color corresponds

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a routing layer.

$$\begin{aligned} & C_i \text{ is a color} \\ \text{maximize} \quad & \sum_{(i,j)\in E} w_{i,j}\Delta(C_i, C_j) \qquad (2.7) \\ \text{where} \quad & \Delta(C_i, C_j) = \begin{cases} 0 & \text{if } C_i = C_j \\ 1 & \text{if } C_i \neq C_j \end{cases} \end{aligned}$$

The idea is to bipartition the graph into The delay calculation errors resulting from ignoring layer assignment and buffer insertion can also yield sub-optimal RT topologies. two sub-graphs recursively, at each step, producing a maximum cut between two sets (equivalently, minimizing interference inside each set). Although, the algorithm is efficient to reduce the crosstalk, it is not performance-driven either. Critical nets being assigned to the lower The delay calculation errors resulting from ignoring layer assignment and buffer insertion can also yield sub-optimal RT topologies. layers may greatly reduce the chance of obtaining the timing closure at the GR stage.

In summary, the existing layer assignment algorithms are generally done without the knowledge of RT topology resulting in suboptimal layer assignment. This is due to the fact that critical sink RT construction algorithms [2, 23, 14] progressively builds the RT which, at each iteration, a node is connected to the subtree (or modified if the starting tree is a sub-optimal tree) so that the maximum delay is minimized (or maximum delay violation or total delay is minimized). Since the delay of a RT has to be calculated in order to decide which connection is next in the suboptimal (or partially completed) tree, resistance and the capacitance of the tree edges should be known which is a function of the routing metal layer that tree is being routed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Therefore, layer assignment must be a integral part of the RT synthesis.

2.4 Buffer Insertion

Buffer insertion is widely recognized as a key technology for improving VLSI interconnect performance by converting quadratic RC delay into linear delay by shielding the down-stream capacitance. This technique also improves interconnect integrity, noise immunity and lateral coupling capacitances. As design complexity increases, the number of buffers needed can be as much as $800,000$ buffers for the $50nm$ technology node [41]. Thus, efficient buffer insertion algorithms are needed to process thousands of nets. The main challenge is to determine the optimal number of buffers and their locations under delay and power constraints.

Ginneken's classic dynamic programming based algorithm [3] uses Elmore delay to calculate the required departure time-load pairs for each node in a RT. The capacitance of each sink and the required arrival time are assumed given. Buffers are also characterized by the input capacitance and an internal delay. Using Elmore delay, recursive formulas can be used to calculate all the possible time-load pairs at node k:

$$
\begin{aligned}
T_k &= min\{T_n, T_m\} \\
L_k &= L_n + L_m
\end{aligned}
$$

where n and m represents left and right sub-trees. This bottom-up process aims to have a departure time at the source as late as possible. A second top-bottom search process actually places the buffers which led to the latest departure time at the source.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

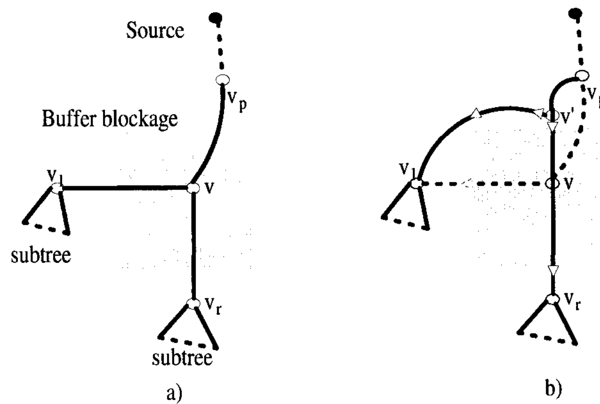


Figure 2.10: Alternate Steiner point to avoid blockage, a) Steiner point in the blocked area b) Alternate Steiner point and buffer location

Later, Hu [4] improved on the Ginneken's algorithm by introducing the buffer blockages to avoid areas already occupied by macros or IP blocks. Hu's algorithm works exactly as Ginneken's algorithm except when a Steiner point is in the blocked area. Then, the Steiner point is moved to the closest candidate location outside of the blocked area. A simple axample process is shown in Figure 2.4. Hu proposed a heurustic to find and choose alternate Steiner points and buffer locations.

Both algorithms assume buffers at Steiner points which may yield sub-optimal solutions if long edges exist requiring buffers in the middle. Although the long wires can be segmented into smaller wires, the optimal segmentation scheme is not clear.

Lillis [5] used a similar technique to include wire sizing and signal slew rate with the focus on minimization of the dynamic power reduction. Lillis calculates two sets of solutions for each node using Elmore delay. Each solution for a node, depending on the chosen objective, may include down tree capacitance, wire size, slew rate and power. The first set (bottom solutions) includes the possible sets of solutions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

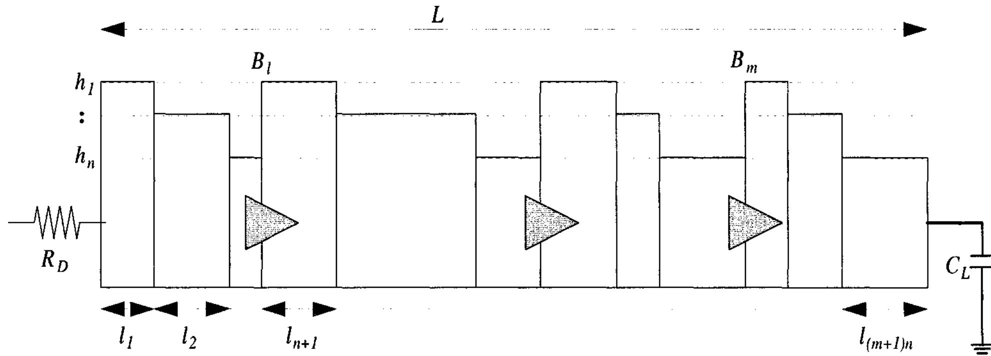


Figure 2.11: Interconnect model for QP formulation

in the sub-tree of the current node. The second set (top solutions) is similar to the first set but includes solutions augmented by the wire from the parent of the current node. The bottom-top process calculates those set of solutions for each node and then the top-bottom process tracks down the best solution. Although a prune operation is suggested to reduce the search space, Lillis' algorithm is exhaustive search in nature and the run-time is a concern. Lillis' algorithm also assumes the underlying ST already exist and does not modify the tree topology. As shown in Figure 2.12, buffer insertion can greatly impact the optimal location of the Steiner points. Lillis' algorithm later is improved [30] to include real-word issues: handling of routing and buffer blockages, cost minimization, critical sink isolation and sink polarities.

Combinatorial optimization methods can also be applied to the buffer insertion problem. A QP approach to the buffer insertion and wire sizing is proposed by Chris [6]. The proposed algorithm minimizes the Elmore delay written in the quadratic form using Equation 2.8 with pre-determined buffer locations and wire sizes as shown in Figure 2.11.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
\text{minimize} \quad & \frac{1}{2} l^T \Phi l + \rho^T l \\
\text{subject to} \quad & l_1 + \cdots + l_n = L \\
& l_i \geq 0 \text{ for } 1 \leq i \leq n
\end{aligned} \tag{2.8}
$$

l_i in Equation 2.8 is the length between segments and the objective calculates Elmore delay using Φ and ρ coefficient matrices in Equation 2.9 and 2.10.

$$
\Phi = \begin{pmatrix}
c_1 r_0/h_1 & c_2 r_0/h_1 & c_3 r_0/h_1 & \cdots & c_n r_0/h_1 \\
c_2 r_0/h_1 & c_2 r_0/h_2 & c_3 r_0/h_2 & \cdots & c_n r_0/h_2 \\
c_3 r_0/h_1 & c_3 r_0/h_2 & c_3 r_0/h_3 & \cdots & c_n r_0/h_3 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_n r_0/h_1 & c_n r_0/h_2 & c_n r_0/h_3 & \cdots & c_n r_0/h_n
\end{pmatrix} \tag{2.9}
$$

$$
\rho = \begin{pmatrix}
R_D c_1 + C_L r_0/h_1 \\
R_D c_2 + C_L r_0/h_2 \\
R_D c_3 + C_L r_0/h_3 \\
\vdots \\
R_D c_n + C_L r_0/h_n
\end{pmatrix} \quad \text{and} \quad l = \begin{pmatrix}
l_1 \\ l_2 \\ l_3 \\ \vdots \\ l_n
\end{pmatrix} \tag{2.10}
$$

The formulation in Equation 2.8 is given for the wire sizing only for simplicity, Chris expands this formulation to include buffer locations and sizes, and shows that their QP formulation is convex which can be solved in polynomial time. However, the tight integration of the Elmore delay makes the delay calculation inaccurate. Use of higher order delay models for better accuracy is impossible since formulation would be no longer quadratic. Furthermore, it is not applicable to RTs but only to two-pin nets.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

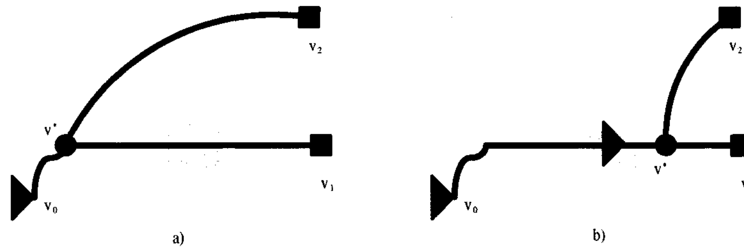


Figure 2.12: a) Steiner point is closer to the driver to reduce delay b)Buffer insertion can reduce wire cost further in Steiner-tree construction

The Elmore delay model is used in the mentioned algorithms to compute the delay which may differ greatly from the actual delay. Since DSM interconnects are much thinner and longer, downstream capacitance is shielded by the interconnect resistance. Elmore delay does not take resistive shielding into account accurately, resulting in remarkably large overestimated errors. Consequently, algorithms that rely on Elmore delay tend to insert more buffers than necessary, increasing the overall power consumption. Spatnekar [2] suggested a greedy iterative buffer insertion algorithm under higher order delay model AWE. The buffer insertion non-Hanan optimization (BINO) algorithm proposed by Spatnekar tests all the buffer locations for the improvement on the cost function. Algorithm inserts a buffer to each edge tentatively followed by a non-Hanan optimization algorithm. The improvement on the cost is saved and the buffer-edge pair with the largest improvement is chosen in the end. Then, the non-Hanan optimization algorithm tries to improve the ST structure by disjoining and re-connecting each node in search of an improvement in the timing. This process repeats until there is no further improvement. Spatnekar's buffer insertion algorithm integrates into ST construction as shown in Figure 2.12, hence it can further reduce the wire cost. BINO algorithm assumes fixed layer assignment for the RT and does

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not modify layer assignment which can greatly improve overall timing of the ST.

RTs generated without respect to buffer insertion yield suboptimal star-like tree topologies. Q-Trees [17] attempts to further optimize the suboptimal buffered trees by *hannan grafting* with a possible buffer replacement. Other algorithms such as P-Tree [39] and C-Tree [42] incorporate multi-objectives of wire sizing, delay/area tradeoff simultanesouly, but assume fixed layer asignment and ignore the impact of buffer insertion on RT topology.

As a summary, buffers are usually inserted last for any nets with delay violations. Therefore, RTs are constructed with incomplete information about buffer insertion. The resulting RTs are often suboptimal. Thus, performance-driven RT construction tends to increase the length with star-like tree structures in order to meet the timing requirements. This will greatly increase the routing resource demand and potentially increase the use of repeaters.

2.5 Steiner-Tree Construction

Global and detailed routers use STs to define the routes. A ST algorithm connects all the nodes (net pins) such that the given objective is at a minimum. The rectilinear ST has a special focus on routing since it is appropriate for VLSI routing due to the restricted routing directions on metal layers. Thus, various algorithms have been proposed [12, 14, 15, 17, 27, 16, 13] to obtain rectilinear STs. Since ST construction problems is known to be NP-Hard [12], heuristics are usually employed involving iterative modifications to a sub-optimal ST. In high performance VLSI routing, naturally, the primary objective cannot be length minimization; delay is a better candidate for optimal ST construction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

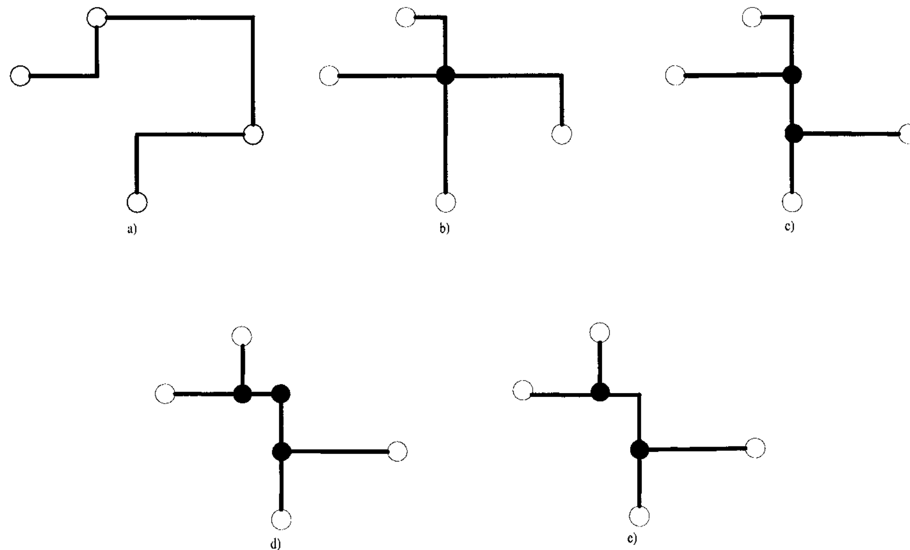


Figure 2.13: 1-Steiner algorithm a) Initial spanning tree b-e) A Steiner point is inserted at each iteration

Robbins [12] was the first to introduce 1-Steiner algorithm for construction of the minimum length rectilinear ST and later improved 1-Steiner algorithm with a faster implementation [25]. The 1-Steiner algorithm repeatedly modifies a spanning tree by reconnecting a node, each time introducing a new Steiner point at the Hanan grid which maximizes the total length improvement as illustrated in Figure 2.13. The process terminates when there is no edge that improves the total length. Later, Robbins reduced the complexity from $O(n^4 logn)$ to $O(n^3)$ by restricting search space to the closest eight neighborhoods of a node. minimum length rectalinear Steiner-tree (MRST) minimizes the total tree length which is preferable in terms of routing resource demand. But, they won't necessarily satisfy the timing constraints due to the fact that minimum length RT doesn't necessarily mean the length from source to the critical sink node is minimized. In VLSI routing, delay minimization is more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

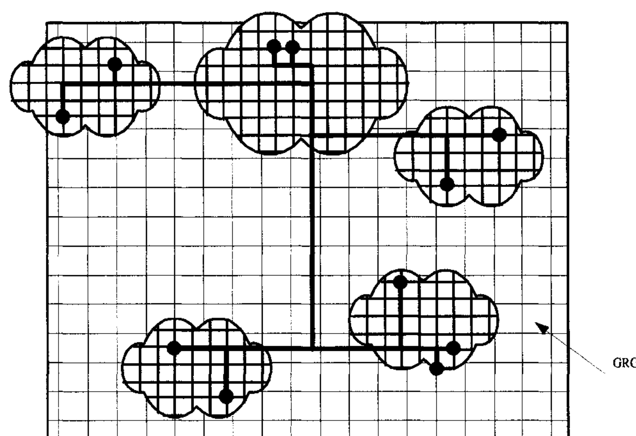


Figure 2.14: Decomposition and reconnect of sub-trees

desirable.

Further run-time improvement for the ST construction can be achieved by dividing the problem into sub-problems. Xu [13] hierarchically decomposes the problem into sub-trees based on a dynamic programming technique. Then, the final-tree is constructed by reconnecting sub-trees at each level with the objective of minimizing delay on the critical path. The algorithm decomposes a block of pins into a sub-block while constructing the minimum delay ST for each sub-block. Then, each sub-block is regarded as a single node at the center of gravity. These nodes are again connected by minimizing the delay from source node to the critical sink node. The decomposition process is illustrated in Figure 2.14. Xu's algorithm can speedup the ST construction of nets with large numbers of pins, but can be sub-optimal since, during the sub-tree construction stage, the rest of ST is not known to the sub-tree construction algorithm. Furthermore, sub-trees are connected together using a virtual node at the center of gravity of each sub-tree which may again yield sub-optimal tree construction.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

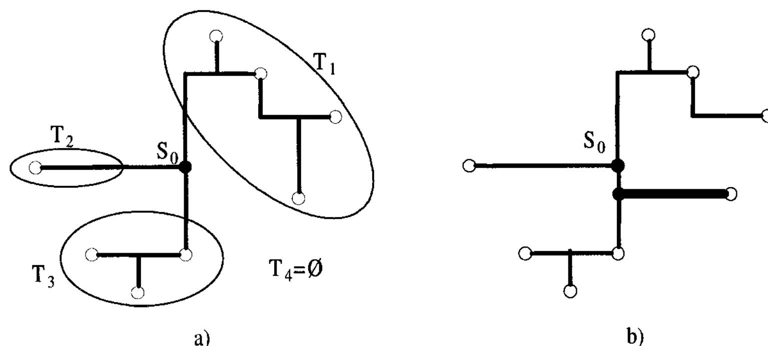


Figure 2.15: Sub-tree transformation, the node in group T_1 is joined to the group T_3

A similar iterative tree transformation method is proposed by Tsujii [15]. Tsujii divides the initial tree into four sub-trees originating at the source. Then, a sink is moved from one sub-tree to another while recording the improvement on the delay. The best move is chosen which maximizes the delay improvement. In Figure 2.15, the sink in the sub-tree group T_1 is moved into sub-tree group T_3 reducing the total delay of the ST. The initial tree is also constructed from a spanning tree using the same method. In Tsujii's algorithm, the choice of dividing the tree into exactly 4-subtree's is not clear. The optimal number of sub-trees may depend on the tree topology.

Jaewon [16] approaches the problem with a linear programming (LP) model for delay bounded ST construction. LP minimizes the total tree length of the ST shown in Figure 2.16 under Steiner and delay constraints. Figure 2.17 shows the corresponding LP formulation. The *Steiner constraints* in the LP formulation dictate that for every possible source-sink pair, the path length should be bigger than the distance of source-sink pair in order to have a feasible solution. Linear *delay constraints* also place upper and/or lower bounds for each source-sink pair. Obviously, Jaewon's algorithm can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

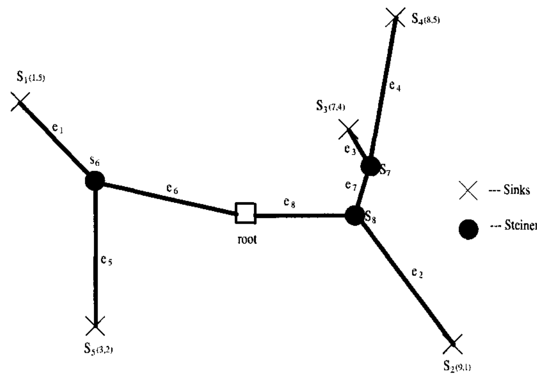


Figure 2.16: An Example Steiner-tree topology

only work with an already given ST topology which is the most important part of the ST construction problem. Furthermore, delay constrains have to be linear in order for the LP model to be linear. Linear delay model can only be achieved using the lengths as shown in the LP formulation in Figure 2.17. Thus, source-sink length is bounded rather than actual delay as claimed in the paper. Even with the minimum critical source-sink path length, the RT may not be suitable for VLSI routing since the delay at any sink is not only the function of source-sink length but also the topology of the rest of the tree.

Boese [14] addressed the critical sink routing-tree (CSRT) construction problem and improved on it with the proposal of global slack removal technique. The critical sink (CS) algorithm iteratively connects a node to the existing tree with the shortest delay. Three variants for construction of the ST are proposed:

- a direct connection from sink to source.
- a direct connection with the shortest path to the existing tree
- a connection by trying all shortest connections from sink to existing tree. it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{array}{rl}
\text{Min} & e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 + e_8 \\
\text{Subject to} & \\
\text{Steiner Constraints} & \\
& e_1 + e_6 + e_8 + e_2 \geq 12 \\
& e_1 + e_6 + e_8 + e_7 + e_3 \geq 7 \\
& e_1 + e_6 + e_8 + e_7 + e_4 \geq 7 \\
& e_1 + e_5 \geq 5 \\
& e_2 + e_7 + e_3 \geq 5 \\
& e_2 + e_7 + e_4 \geq 5 \\
& e_2 + e_8 + e_6 + e_5 \geq 7 \\
& e_3 + e_4 \geq 2 \\
& e_3 + e_7 + e_8 + e_6 + e_5 \geq 6 \\
& e_4 + e_7 + e_8 + e_6 + e_5 \geq 8 \\
\text{Linear Delay Constraints} & \\
& 4 \leq e_1 + e_6 \leq 6 \\
& 4 \leq e_2 + e_8 \leq 6 \\
& 4 \leq e_3 + e_7 + e_8 \leq 6 \\
& 4 \leq e_4 + e_7 + e_8 \leq 6 \\
& 4 \leq e_5 + e_6 \leq 6
\end{array}
$$

Figure 2.17: LP model for bounded Steiner-tree construction

performs timing analysis, and returns the tree with the shortest delay.

The latter is preferred since it is timing driven; however, the complexity is the highest. Global slack removal post-processor aims to maximize monotonicity of all source-sink paths and reduces the Elmore delay without increasing overall tree cost. As shown in Figure 2.18 and Figure 2.19, this is done by removing V and U shapes and introducing new Steiner nodes. Removing V and U shapes reduces the source-sink path length to the current node and its descendants, and does not affect the source-sink path lengths

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

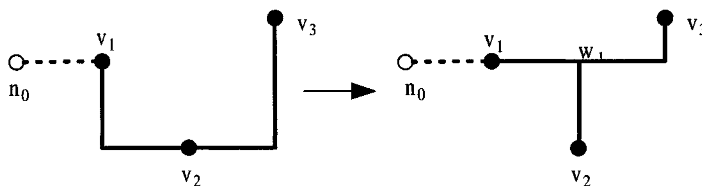


Figure 2.18: V Removal

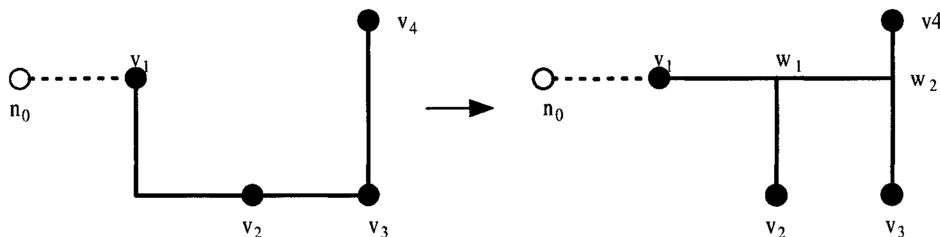


Figure 2.19: U Removal

anywhere else. Hence, the resulting tree is guaranteed to dominate the previous tree. These two operations can affect the Elmore delay in three ways:

1. changing the length of the path,
2. changing the tree capacitance along the path
3. shifting the tree capacitances along the path.

V and *U* operations will reduce the path length, reduce the tree capacitance and shift the tree capacitance closer to the source, thereby reducing the Elmore delay to all pins. *V* and *U* operations may also reduce the total cost or leave it unchanged by the triangle inequality of removing a low-degree Steiner point. Global slack removal is an efficient post-processing step to further optimize sub-optimal STs.

In an iterative ST construction algorithm, delay calculation can easily be replaced

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

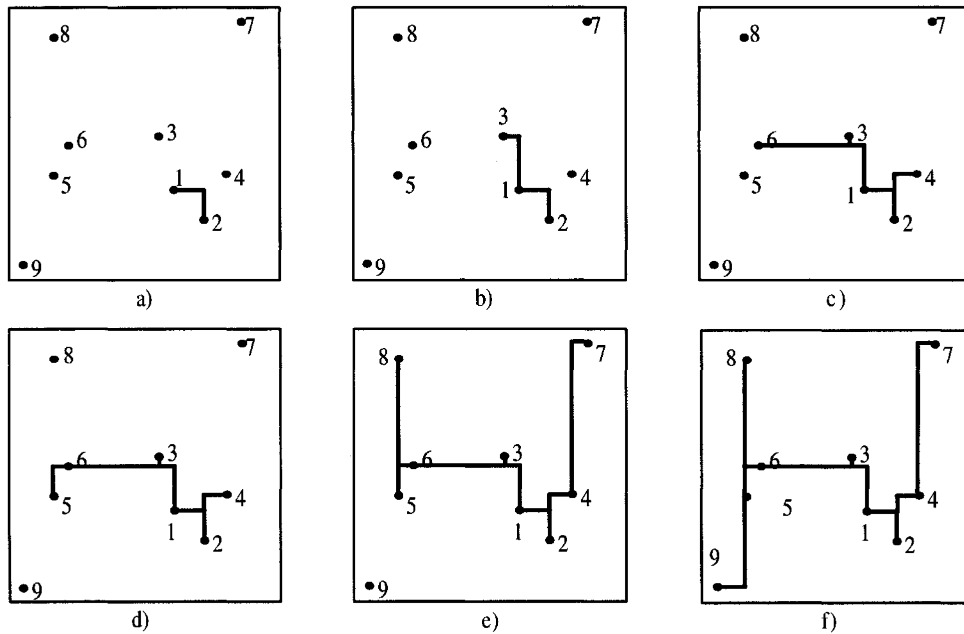


Figure 2.20: a-f) Progressive Steiner-tree construction

by more accurate moment matching techniques such as AWE. Spatnaker [2] proposed a greedy iterative algorithm using the AWE delay model. steiner AWE routing-tree (SART), starting with source node, grows a partial RT in a greedy fashion. A nine pin ST construction example is illustrated in Figure 2.20. In each step, a previously unconnected sink is selected and connected to an edge in the partial tree such that the maximum delay under the AWE model is minimized. Spatnekar shows that optimal Steiner point for performance driven RTs is not at the Hannan grid but somewhere on the upstream edge as shown in Figure 2.21. If a Steiner node is to be inserted in the SART algorithm, a binary search is performed on the upstream edge which minimize the delay violation on the critical sink, non-Hanan shifting. Spatnekar also integrated their buffer insertion algorithm [2] into SART for simultaneous RT

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

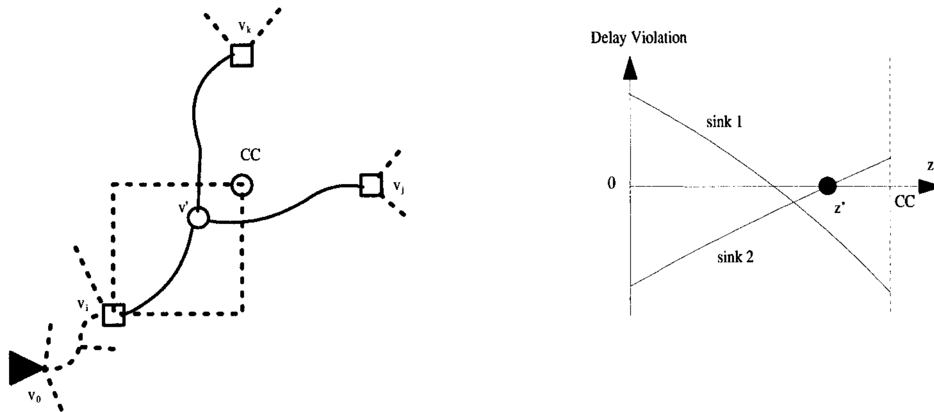


Figure 2.21: Optimal delay non-Hanan Steiner point

construction and buffer insertion which can significantly improve overall performance of the ST.

Non-Hannan-sliding which shifts the Steiner point upstream to reduce the delay does not modify the tree topology which may be required if buffers are inserted into the tree. Kahng [17] adapted the basic edge replacement operation, referred as Hanan-grafting [40], by allowing possible buffer insertion. The greedy Q-Tree algorithm iteratively replaces a tree edge, same as in [40], but now assumes a buffer inserted on the new edge. Grafting with the buffer insertion is illustrated in Figure 2.22. They observed that Hanan-grafting achieves the largest required arrival time improvement per unit extra wire length except that introduction of extra wire length makes a different sink critical. The proposed Q-tree algorithm first tests the performance improvement of the tree by the grafting process. If there is no improvement, it proceeds with the sliding.

In conclusion, RT synthesis must be performance-driven for VLSI routing. Length reduction is no longer the only objective of RT synthesis process. Moreover, accurate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

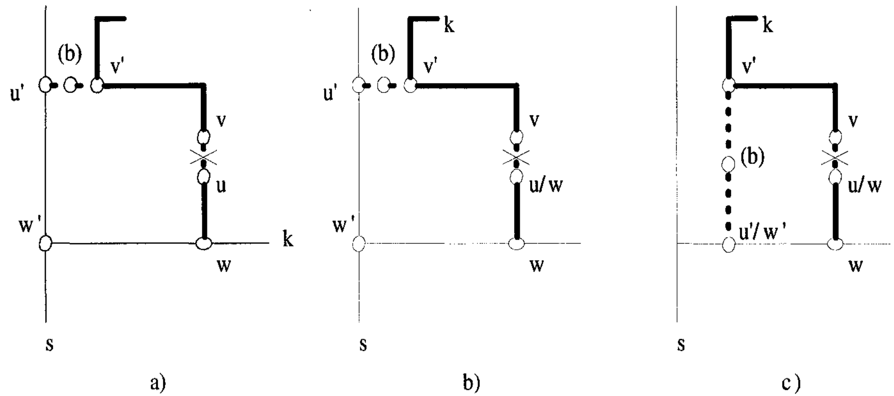


Figure 2.22: Grafting, When the edge uv is removed and edges $u'v'$ is added, buffer b is considered in the delay calculation

delay modeling should be used in order to take DSM effects into account. Thus, closed form of delay calculations (Lumped, Elmore) can not be used anymore. Buffer insertion and layer assignment must be the integral part of the RT synthesis since length, delay and the topology of the RT is greatly affected by those factors.

2.6 Global Routing Algorithms

GR algorithms use a collection of the above algorithms to obtain a path for each net. Although achieving the connectivity between pins without overflow on routing resources is the goal, a GR algorithm must also take into account delay, crosstalk and congestion. After all, a timing failure is considered as an unsuccessful routing even though all pins are connected successfully without overflow.

Various approaches have been proposed for the GR [19, 43, 27, 44, 31, 45, 46, 7, 24, 2, 32, 18]. Although optimization methods vary greatly we can classify them into two main categories: sequential and parallel routers. Sequential algorithms obtain

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

routing net by net. Sequential routers provide good solutions for the first a few nets since they are able to focus on one net at a time, but result in poor routing for later nets due to their poor congestion management. It is very common that highly demanded routing regions exist in today's chips. Sequential routers tend to consume the routing resources in high demand areas in the early stages, forcing later critical nets to detour. Therefore, they usually perform a rip-up and reroute loop to handle the failing nets. Consequently, a sequential router's result depends on the net ordering. A parallel router attempts to handle the net ordering problem by routing all the nets at once. Although parallel routers yield better overall results, they suffer from long run-times due to the large size of the optimization problem used to model the routing. Hierarchical approaches are usually employed to divide the problem into smaller sub-problems to improve the run-times and the memory requirements. Unfortunately, if not designed carefully, hierarchical approaches can yield sub-optimal solutions.

Well known maze router, also called Dijkstra's shortest path algorithm, finds the shortest path between two nodes by propagating a wave front from the source node. When the wave front hits the destination point, the path is tracked back to the source revealing the shortest path between two points. Maze router is very popular among net by net routers and has been modified extensively to handle different situations.

Huang [32] modified the maze algorithm to simultaneously insert buffers while laying wires. It was demonstrated in [33] that simultaneous routing and buffer insertion can produce significantly better results. Instead of a routing grid graph, used in traditional maze routers, a buffer planning graph (BPG) shown in Figure 2.23 is used to construct the RT. The algorithm basically inserts a vertex into the graph for each possible buffer location, and then a maze router is applied to find the shortest

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

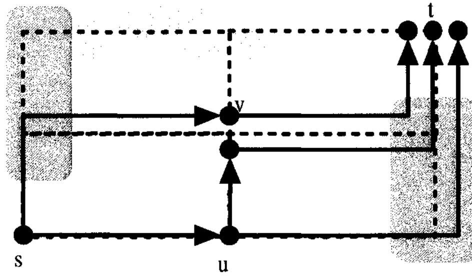


Figure 2.23: A routing grid graph

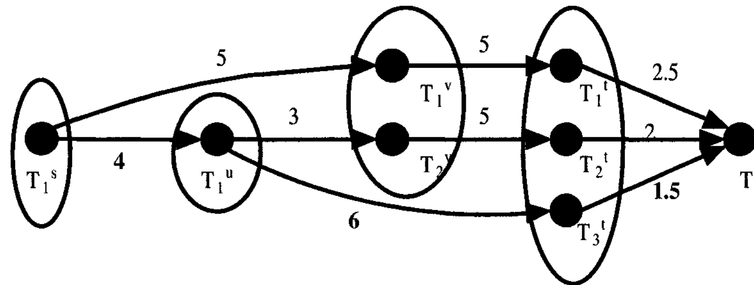


Figure 2.24: Corresponding Buffer Planning Graph (BPG) of routing grid graph in Figure 2.23

path from the source to the sink while checking the transition time constraints from a look-up table for each step. Buffer-to-buffer shift times are assigned as edge weights in BPG. Therefore, the delay from the source node to the sink node is the sum of the shift times along the path plus 50% of the transition time at the last buffer. Shift time and transition time are illustrated on a signal in Figure 2.25. The shortest path returned by the maze router is the minimum delay path from the source to the sink. Unfortunately, during the construction of a BPG, each possible path has to be checked exhaustively for routability between buffers and for timing, which poses a problem for large complex chips with thousands of possible buffers. Furthermore,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

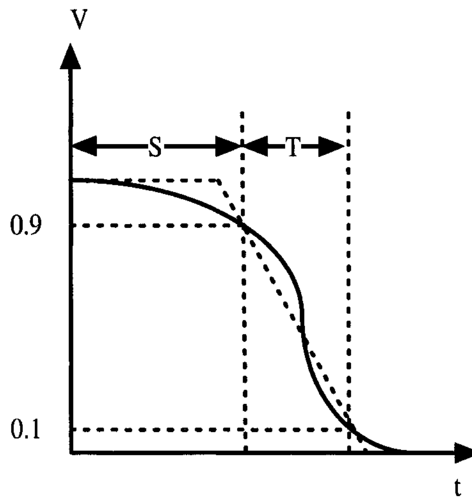


Figure 2.25: An arbitary signal is represented by a finite ramp with shift time S and transition time T

for long nets requiring many buffers, the number of nodes required for a buffer in the BPG increases geometrically by the number of possible paths from other buffers to that buffer as can be seen from the example in Figure 2.24, .

To better handle the rip-up and re-route problem, Hadsell [18] proposed a new congestion estimation method. Their algorithm extends the previous routing [47] by using an amplified congestion estimate to influence the rip-up and reroute approach. Their proposal of using an amplified congestion measure is based on the observation that routing becomes difficult when routing demand approaches its physical capacity. Therefore, the algorithm simply assigns unit cost to an edge until it reaches 80% of the capacity, and increases the cost linearly until it reaches 40% above the capacity as illustrated in Figure 2.26. The intuition for the linear cost is that a maze router will utilize an un-congested routing graph without consideration for the demands of later routes. Later routes then will detour around congested areas, increasing overall

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

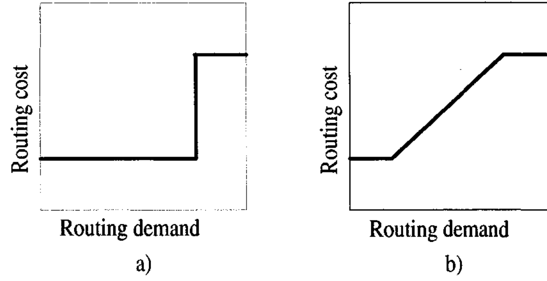


Figure 2.26: a)Traditional congestion measure, b) Amplified congestion measure

routing demand and the net's delay considerably. By amplifying routing cost in congested areas, a hint is generated for the GR where to expect congested areas in the later routings. Proposed congestion estimation measure can be efficient to prevent sequential routers from using excessive routing resources in the congested areas but impractical for parallel routers where accurate congestion information is not known before the routing.

Bao [19] proposed a similar approach to the rip-up and re-route scheme to reduce the effects of net ordering. Bao simply uses a penalty function E_t defined in Equation 2.11 for each net to determine if the net really needs routing resources in the congested area. E_t is proportional to the number of bends, tree length and the penalty congestion value which measures congestion with a penalty similar to [18].

$$E_t = \alpha w_t/max_{x \in T_n}(w_x) + \beta l_t/max_{y \in T_n}(ly) + \gamma b_t/max_{z \in T_n}(b_z) \tag{2.11}$$

In Equation 2.11, w_t is the total penalty congestion of all the ST edges, l_t is the total length and b_t is the number of bends. Then, Bao randomly selects a sub-set of nets from the congested routing segment to be re-routed simultaneously. However,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the net ordering problem still exists between *selected congested nets* and *non-selected congested nets*. Bao proposes a set of six heuristics to further eliminate the net ordering problem in their algorithm. Although Bao suggests simultaneously route ripped nets, there is no method provided for this simultaneous routing. Furthermore, despite the length that is used in the penalty function, the delay is not considered in the formulation. Thus, critical nets can be ripped (since they may have a lesser penalty value) and take detours further degrading the performance of the critical net.

Alti [43] proposed a neural network routing model to avoid obstacles in the layout. The energy equation for parallel neural network model is modified to handle the crowding problem for use of the rip up and reroute process.

Another different approach is based on genetic algorithm (GA)s by Esbensen [27]. This two phase algorithm first constructs a ST for each net using a GA. The cost is assumed as the length of the edge; hence it builds a rectilinear minimal ST. The result of the first GA is several RTs for each net. Then, a second GA selects solutions while minimizing the area and secondarily the interconnect length. Esbensen's algorithm primarily focuses on area minimization but singles out delay.

The GR problem can be formulated as a MIP to route all the nets at once. Albrecht's algorithm [44] pre-calculates all the possible STs and lets the MIP model choose which one is the best by minimizing the maximum congestion. The MIP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

formulation is expressed in Equation 2.12.

$$\begin{aligned}
&\min \lambda \\
&\text{subject to} \\
\sum_{i,T:e\in T\in T_i} w_{i,e}x_{i,T} &\leqslant \lambda c(e) && \text{for } e \in E \\
\sum_{T\in T_i} &= 1 && \text{for } i = 1,\dots,k \\
x_{i,t} &\in 0,1 && \text{for } i = 1,\dots,k;\ T \in T_i
\end{aligned} \tag{2.12}$$

where the λ is the maximum relative congestion, $c(e)$ is the capacity (number of the free channels on edge e), $x_{i,T}$ is a $0-1$ variable each representing a pre-calculated ST and $w_{i,e}$ is the with of the edge if wire sizing is different for each edge. In this formulation, the first constraint forces that the number of edges occupying each routing resource must be smaller than its capacity while the second constraint forces the MIP to choose only one ST for each net. As well known, solving the MIP model of a GR is difficult. Albrecht converts the original MIP formulation in Equation 2.12 to a relaxed formulation of LP formulation of Equation 2.13 and solved it with randomized rounding [48, 49].

$$\begin{aligned}
&\min \lambda \\
&\text{subject to} \\
\sum_{i,T:e\in T\in T_i} w_{i,e}x_{i,T} &\leqslant \lambda c(e) && \text{for } e \in E \\
\sum_{T\in T_i} &= 1 && \text{for } i = 1,\dots,k \\
x_{i,t} &\geq 0 && \text{for } i = 1,\dots,k;\ T \in T_i \\
\sum &= 1
\end{aligned} \tag{2.13}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  y_e := 1/c(e) for all e ∈ E
2  x_{i,T} := 0for all i = 1,...,k; T ∈ T_i
3  T_i; = φ for i = 1....,k.
4  while (∑_{e∈E} c(e)y_e < M)
5  begin
6     for i := 1 to k
7     begin
8        if T_i = φ or ∑_{e∈T_i} w_{i,e}y_e > (1 + γε)z_i
9           Find a minimal Steiner tree T_i ∈ T_i
10          for net i with respect to length w_{i,e}y_e, e ∈ E.
11          z_i := ∑ e ∈ T_i w_{i.e}y_e
12       x_{i,T_i} := x_{i,T_i} + 1
13       y_e := y_e e^{ε w_{i,e}/c(e)} for all e ∈ T_i
14    end
15 end
```

Figure 2.27: Approximation algorithm for fractional global routing

The fractional GR problem is approximated to the final solution using the dual of the LP shown in Equation 2.14 which provides the lower bound of the solution.

$$\begin{aligned} &\max \sum_{i=1}^{k} z_i \\ &\text{subject to} \\ &\sum_{e \in E} c(e) y_e = 1 \\ &\sum_{e \in T} e_{i,e} y_e \geq z_i \quad \text{for } i = 1 \ldots k;\ T \in T_i \\ &y_e \geq 0 \quad \text{for } e \in E \end{aligned} \tag{2.14}$$

Figure 2.27 shows the pseudo-code of the proposed algorithm. As seen from algorithm, While $\sum_{i=1}^{k} z_i$, the objective of the dual problem is maximized, z_i can be substituted by the minimum length of all STs and by rescaling y_e such that the first inequality in Equation 2.14 holds. Albrecht claims that not only is the maximum

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

relative congestion minimized, but the congestion of the edges is distributed equally which is desirable for crosstalk minimization.

GR algorithms using the MIP model may have certain drawbacks:

- Solution space is restricted since, when all the possible STs included in the MIP, the MIP size can greatly increase, certain degree of approximation to reduce search space is unavoidable.

- Timing calculation is difficult to integrate into the MIP model since delay modeling is not linear.

Hence, timing constrained models involving hierarchical simplified MIP models are used for routing. Spatnekar [2] proposed a timing-constrained network-flow based assignment algorithm. As shown in Figure 2.28, this algorithm hierarchically bisects the routing region and assigns *soft edges* to the cell boundaries along the bisector line. A soft edge is an edge that can be routed anywhere in the bounding box without increasing the edge length and the delay. First, initial routing is done through a timing-driven ST construction algorithm without regard to the congestion. Then, a node is inserted into the network as shown in Figure 2.30 for each edge. Arcs connecting soft edge nodes to the cell boundaries represent a possible assignment of the soft edges. Naturally, there might be no feasible solution to the network problem since routing demand (number of output arcs) may exceed routing resources (capacity of input arcs). Spatnekar introduced a min-cut based heuristic to expand boundaries of soft edges such that the network is feasible. A min-cut algorithm is run to see if the maximum feasible flow is less than the number of soft edges to be assigned to a routing segment. In that case, the network might be unfeasible, with additional arcs being added from congested area (shaded area in Figure 2.30) to un-congested areas.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

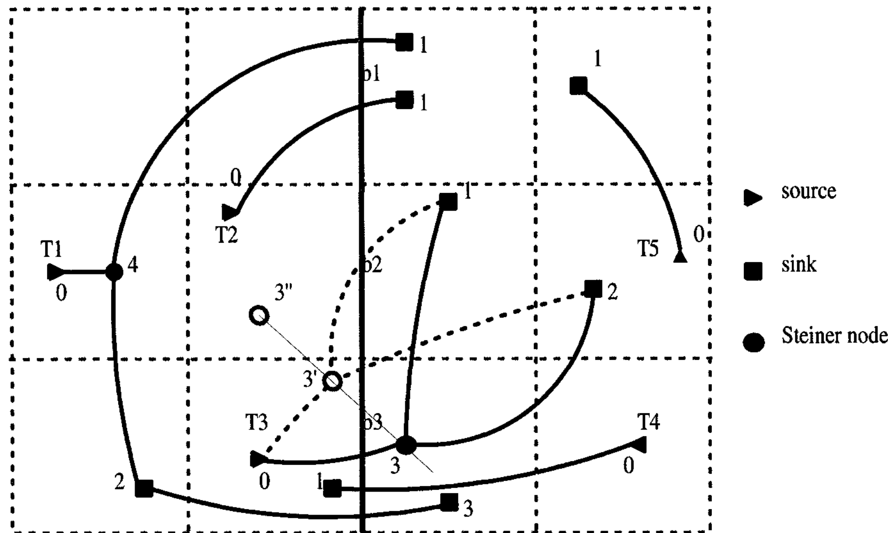


Figure 2.28: Bisection

This process is repeated until it is guaranteed that there is a feasible solution. The extended soft edges will obviously increase the delay of the RT, therefore, the cost of expanded edge arcs is set to the $cost(b_k, e^i_{jl}) = (S_{old}(T^i) - S_{new}(T^i) + 1)^2$ to discourage the network solver to assign extended soft edges. S_{old} and S_{new} are the slacks when the soft edge is routed inside and outside respectively. As seen from the network formulation, the algorithm focuses on eliminating overflows while minimizing delay separation from the initial performance driven RT. This can seriously hurt congestion optimization since it can result in congested areas while less congested areas are available for routing. Moreover, perhaps the main disadvantage of this algorithm is that it is suboptimal due to bisection. An available routing area in a certain cut line can force the same edge to route through a congested or blocked area on both sides of the edge. The example in Figure 2.32 shows that the assignments of the wire on both bisect lines are feasible solutions for the network, but it fails to avoid the blockage

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

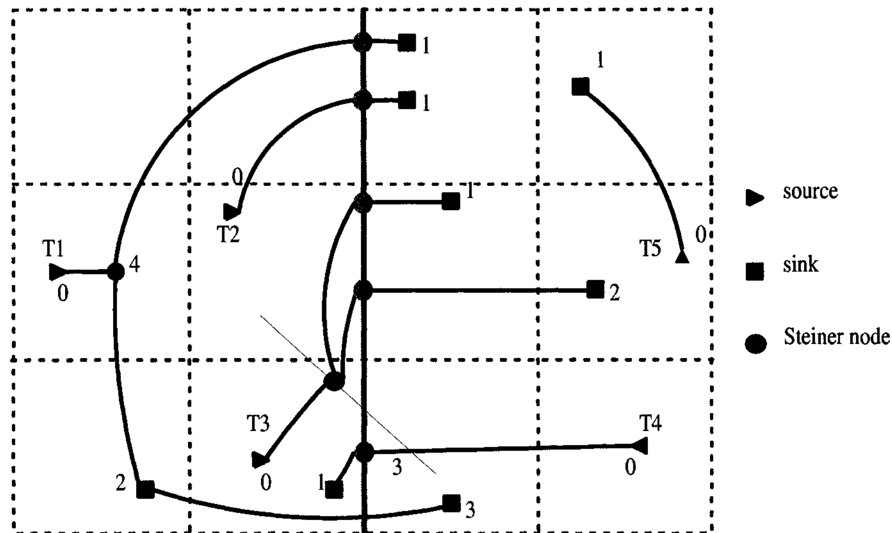


Figure 2.29: An Assignment

in the middle, possibly causing an unnecessary detour. When bisection occurs in the blocked area, the net is forced to detour increasing delay while a better solution was available in the first place in the first cut. Spatnekar also introduced Slide-able Steiner Nodes (SSN) as illustrated in Figure 2.29 to better utilize the Steiner points and then modified the network model accordingly with the introduction of a pseudo node for the Steiner point as shown in Figure 2.31. If slide-able Steiner points are involved, conventional network flow algorithms to solve the network model can not be applied. Fleischer-Wayne algorithm [49] is used as the optimization engine which is slower than conventional network solvers.

Deguchi [31] also proposed a hierarchical routing scheme by dividing layout into 4x4 regions. A three phase algorithm first partitions the routing area into 4x4 tiles as shown in Figure 2.33. Routing in this stage is similar to [44] , An MIP is used to choose the best ST from a set of pre-calculated STs for each net. Then, in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

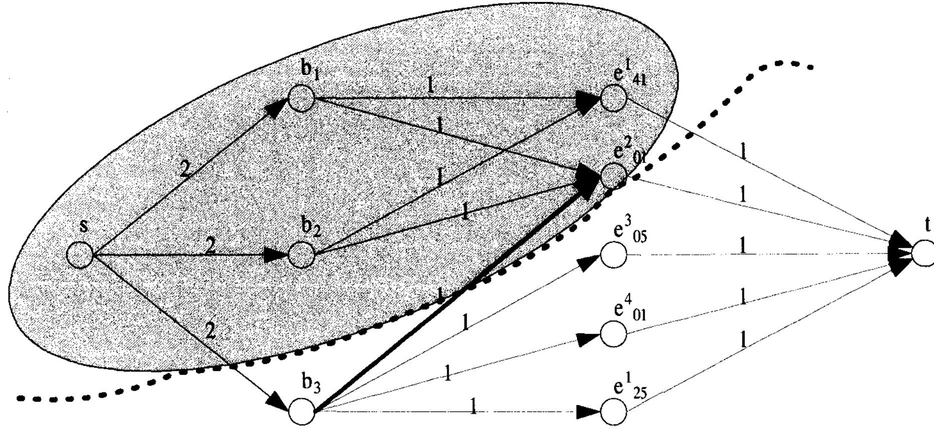


Figure 2.30: Network Formulation of the example in Figure 2.28 without SSN

second phase, each block is hierarchically decomposed into 4x4 block for the next hierarchical level. In this section, a pair of virtual source-sink pair is inserted at slots (on block boundary) as illustrated in Figure 2.34. After repeating phase 1 and 2 until the lowest level is reached, some net might cause timing violation due to the discrepancy of estimation of the interconnect delay. Phase 3 uses a weighted maze router to route the nets at level $l+1$ and level l. This obtained route is rerouted at level $l+1$ again using the same procedure at Phase 2 as shown in Figure 2.35.

2.7 Conclusions

In this chapter, we reviewed the existing work on technologies used for GR. Length, linear delay model, and Elmore delay model based algorithms are no longer sufficient to construct the performance-driven RTs. More accurate timing model such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

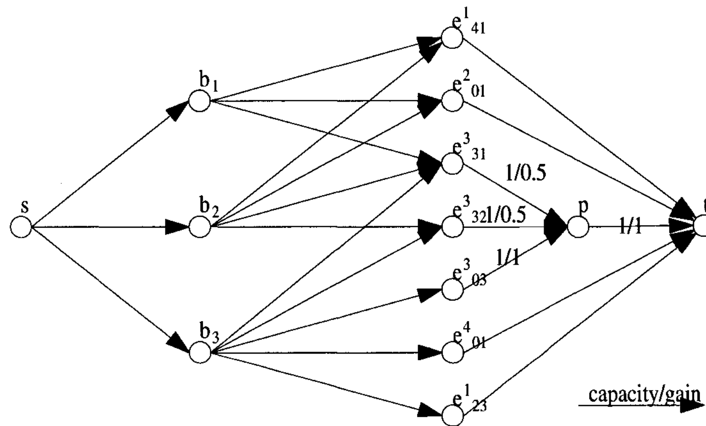


Figure 2.31: Network Formulation

as AWE must be used to accurately calculate the delay in DSM VLSI chip routing. The delay calculation errors resulting from ignoring layer assignment and buffer insertion can also yield sub-optimal RT topologies. The simultaneous implementation of performance-driven ST construction, layer assignment and buffer insertion algorithms will result in better overall result in terms of delay violation. The length minimization and layer utilization are also important as the secondary objectives to prevent the overuse of the routing resources. Traditional routing flow first builds a RT topology. Layer assignment is performed before or after the routing process. Buffers are usually inserted last for any nets with a delay violation. Some attempts were made [8, 30, 2, 31, 32, 33, 17, 11, 6] to incorporate buffer insertion or layer assignment into the routing process. But no attempt has been made so far to integrate both buffer insertion and layer assignment into the RT construction. Therefore, RTs are constructed with incomplete information about layer assignment or buffer insertion. The resulting RTs are often suboptimal. Thus, performance-driven RT construction

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 2.32: Hierarchical bisection method fails to avoid blockage

tends to increase the length with star-like tree structures in order to meet the timing requirements. This will greatly increase the routing resource demand and potentially increase use of repeaters. Moreover, the resulting RT topology may also uneven layer resource usage due to failed nets being reassigned to other layers during the rip-up and reroute process.

Current methods address the GR problem in a sub-optimal fashion. A greedy sequential router or routing net-by-net [32, 9, 18] suffers from slow convergence and doesn't scale well with design complexity. Sequential routers provide good solutions at the beginning since they are able to focus on one net at a time, but results in poor routing for later nets due to their poor congestion management. Thus a rip-up and re-route scheme must be employed for the failing nets. Consequently, a sequential router's result depends on the net ordering. To better handle the rip-up and re-route problem to some degree, Hadsell [18] proposed an amplified congestion estimate to influence the rip-up and reroute approach. Bao [19] proposed a similar approach using a penalty function for each net to determine if the net really needs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

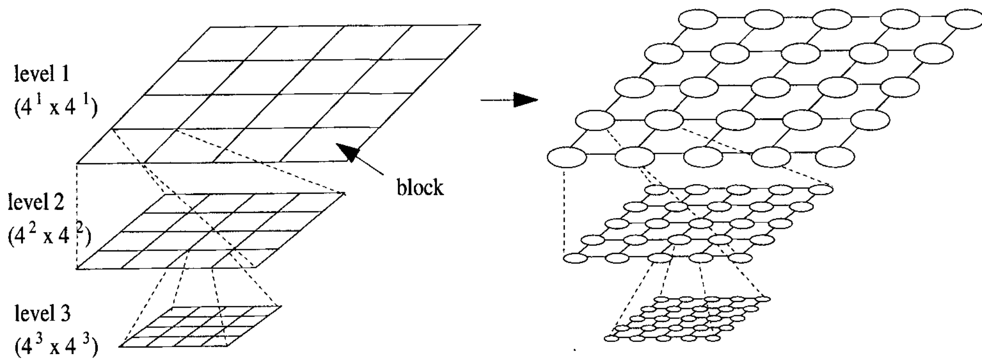


Figure 2.33: Hierarchical decomposition

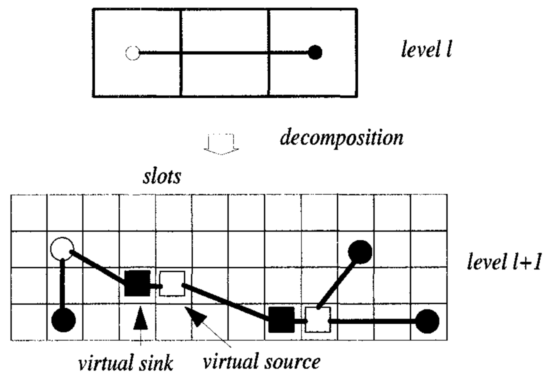


Figure 2.34: Slots and virtual terminals

routing sources in the congested areas. A parallel router employing multi-commodity flows attempts to solve this net ordering problem by routing all the nets concurrently, but it suffers from long run-time due to the size and complexity of the optimization model. Albrech's algorithm [44] pre-calculates the possible RTs and lets the MIP model minimize the maximum congestion. To cope with the issue of long run-time, Albrecht employs randomized rounding [48, 50]. MIP based algorithms can manage the congestion better, but the solution space is limited. Considering all the possible

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

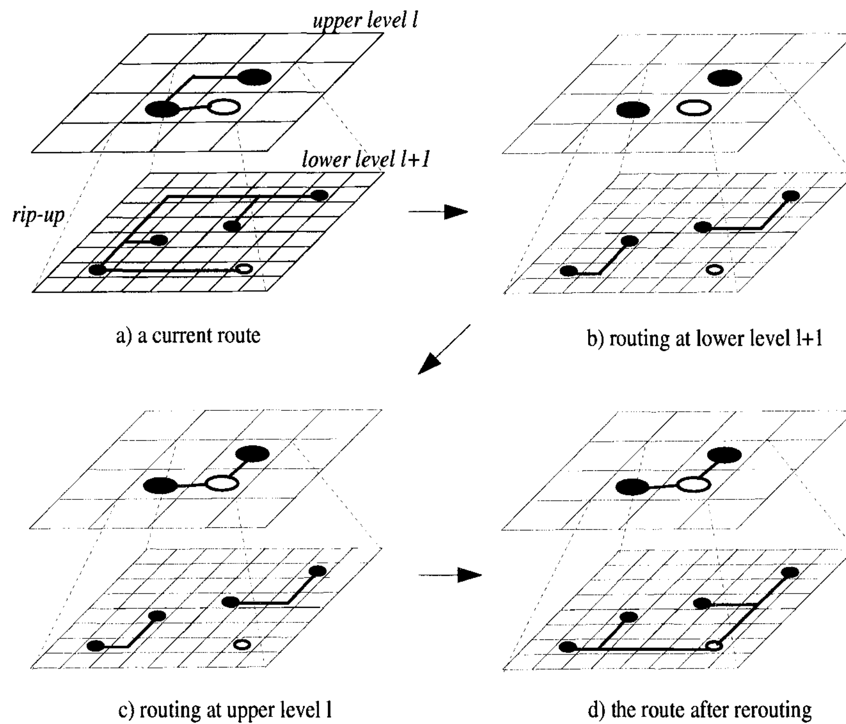


Figure 2.35: Bottom-up rerouting

routes is deemed impractical for MIP. Hierarchical approaches are usually employed to divide the problem into smaller sub-problems to improve the run-times and the memory requirements. Spatnekar [2] proposed a network-flow based assignment algorithm. Spatnekar's algorithm hierarchically bisects the routing region and assigns soft edges to the cell boundaries along the bisector line. Network model reduces the problem size by dealing with a bisect line of the die one at a time and reduces the run-time greatly since network models can be solved much faster than MIP. However, such a tradeoff often results in suboptimal solutions. Moreover, all multi-commodity based solutions minimize the maximum congestion. This effectively prevents them

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from achieving more homogeneous congestion distribution. This is because in practice, highly congested areas exist. However, it doesn't mean the rest of the chip can not be further optimized. This is illustrated in Figure 3.21 where the highly congested area on the top of should not prevent the router from optimizing the routes at the bottom in terms of congestion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Global Routing (GR)

3.1 Introduction

This Chapter uses a new GR algorithm to address the following issues:

- **Performance-driven routing:** The RT construction, layer assignment and buffer insertion algorithms are performance-driven to obtain the timing closure for the projected clock speed. Congestion optimization is bound by the delay of the nets by performing the routing within the bounding box of the net as much as possible.

- **Multi-criteria RT construction:** Existing GRs often construct the RT by ignoring the impact of the layer assignment and the buffer insertion on the final routing structure. Buffer insertion and layer assignment are generally carried out in isolation from RT construction [16, 13, 10, 11, 8, 7, 2, 6, 17, 5, 15, 30, 14, 4]. Thus, the RT is often under-optimized or over-optimized due to the delay estimation errors incur from ignoring the layer assignment and the buffer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

insertion. Our algorithm performs RT construction, layer assignment and buffer insertion *concurrently* for better overall results.

- **Utilization of routing layers:** Routing resources are limited on each routing layer. Routing congestion on a layer can occur if layer assignment is sub-optimal. Thus, congested routing areas can be created resulting in detours of the critical nets and increased cross-talk in the congested areas. Our method try to distribute the nets equally among routing layers such that utilization is the same for each layer, while trying meet the delay requirements.

- **Accuracy of delay modeling:** Use of lumped or Elmore delay model in the DSM VLSI interconnect synthesis can yield erroneous solution. For example, Elmore delay tends to overestimate the delay resulting in overuse of routing and buffer resources. The GR uses moment matching based delay calculation engine (AWE timing model [36]) in order to adequately calculate the delay.

- **Performance of parallel routing:** Sequential routers tends to use high demand routing areas in the early stages of the routing [24, 2, 46, 19, 44]. Therefore, later nets fail to route often requiring rip-up and reroute. The multi-commodity flow based routers overcome this problem by routing all the nets at once. However, the existing multi-commodity flow based routers suffer from extremely long run-time due to the complexity of the optimization model. We use a hierarchical routing scheme using a novel network-flow model to simplify the optimization model. Our network-flow model avoids the MIP formulation often found in the conventional approaches, thus, achieving significant speedup over the conventional multi-commodity flow formulation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- **Congestion optimization:** A hierarchical routing scheme often yields sub-optimal solution and fails to optimize congestion globally since the optimization occurs in a small portion of the routing area at each hierarchical level. Our method overcomes this shortage by performing a pre-route using a simplified MIP model to optimize the congestion globally. Then, the loose routing produced by the pre-router is further optimized by the hierarchical network-flow router. The MIP problem size is much smaller than traditional MIP problems. Hence, the impact on the run-time is minimal. Furthermore, our algorithm focuses on the congestion minimization even when there is no overflow on the routing resource. This will result in a more homogenous congestion distribution. When the routing demand is close to the routing capacity of a routing area, DR is still hard even though the fact that routing demand hasn't exceeded the routing capacity.

- **Blockage handling:** Our algorithm takes into account the blockages and existing routes throughout the routing process. The routing resource consumed by the blockages and the existing routes (such as power and clock grid) is included in layer utilization. Moreover, optimization models used in the MIP pre-router and the network-flow router are constructed by taking the existing blockages and routes into account.

- **Via minimization:** Vias in the interconnect routing incur additional cost and potentially pose reliability problems. Via minimization is achieved by assigning a higher cost to the bends (straight paths are preferred). Further via minimization is also achieved by assigning the whole net to one layer-pair initially. With later routing stages not modifying the layer assignment of the net, the via usage

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can be minimized.

The GR process starts by constructing the RT for each net using the timing-driven rectilinear STs. As the RT being constructed, critical nets are assigned to the good layers and/or buffers are inserted to achieve the *positive slack* at sinks. Layer assignment algorithm also prevents over-utilization of the good layers by equally distributing the layer usage (blockages + existing wires + estimated net length in that layer).

Once the RT topology, layer assignment, and buffer locations for each net are known, a rough two-bend routing for each net (pre-route) is obtained from possible two-bend paths using a MIP routing model. With this step, congestion is optimized globally since the next hierarchical network-flow routing can only optimize the congestion on the bisect line. The actual routing is done by hierarchically bisecting the routing area and using a network-flow routing model to assign the nets to the routing segments.

Finally, a series of heuristics (edge flipping, detour, maze routing, via pad violation elimination) are used to clear up any remaining overflows.

Since accurate congestion knowledge was not available during tree construction for each net, the layer assignment obtained in the first step may be suboptimal and may result in overflows later in the routing stages. A QP model is used in both the MIP routing and the network-flow routing to further fine-tune the layer assignment.

Top level of the overall GR algorithm is illustrated in Figure 3.1.

The remaining of this chapter is as follows: Section 3.2 explains in detail the simultaneous performance-driven buffered RT construction with layer assignment. Section 3.6 presents the MIP routing model and overviews the new congestion optimization method. Section 3.7 explains the hierarchical bisection process and constructs the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

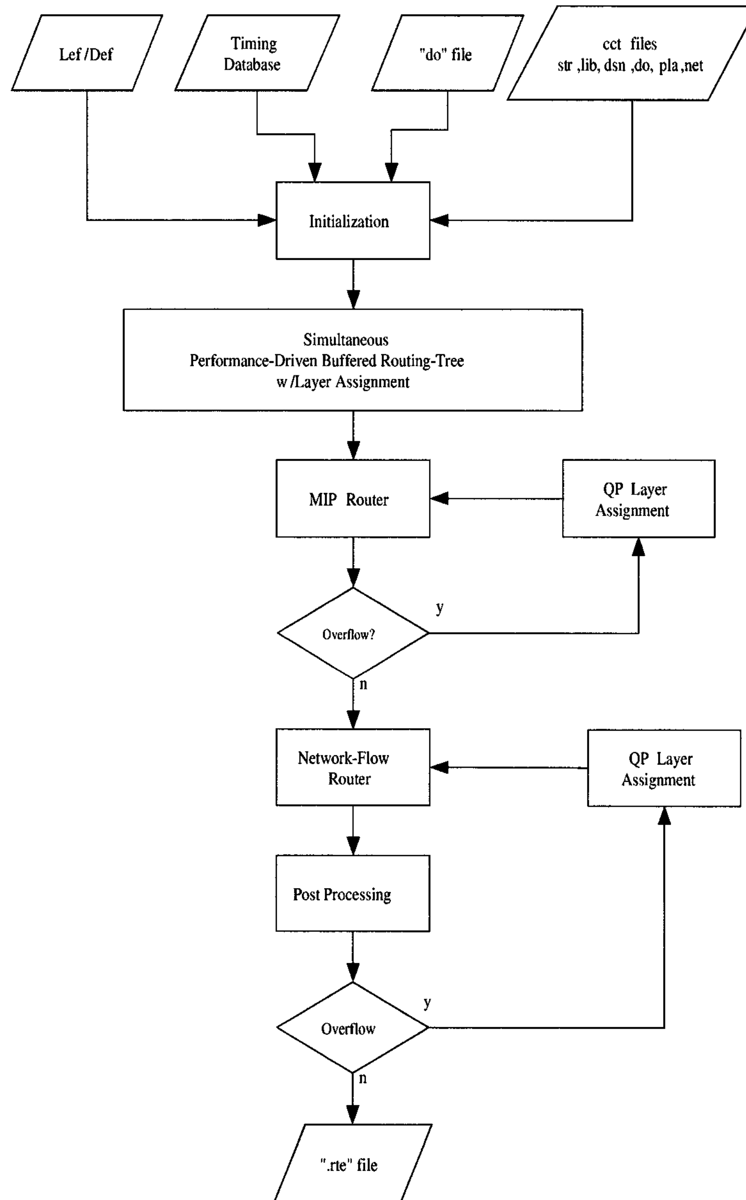


Figure 3.1: Global Routing flow chart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

network-flow routing model. Section 3.8 presents the QP layer assignment model using an interference metric to minimize the congestion and the cross-talk. Section 3.9 summarizes four post-processing heuristics to clear up the remaining overflows.

3.2 Performance-Driven Buffered Routing-Tree with Layer Assignment

If a net has more than two pins, a tree structure is needed to establish the connectivity of all the pins. We will call this tree structure a routing-tree (RT) in this dissertation. It is often desired that the RT is not cycling due to the complexity of the delay calculation and a certain level of unpredictability of the delay from one pin to another on different paths of a cyclic tree. From now on, we assume a RT is a non-cyclic tree.

The RT algorithm incorporates buffer insertion and layer assignment as dual objective in RT construction to achieve delay and power improvement over existing methods. Each step of the RT construction takes advantage of partially constructed, but layer and buffer optimized, RT to construct the next routing segment.

While the primary objective of the mentioned algorithm is delay reduction, the secondary objectives may also exist:

- RT construction may also minimize the tree length to reduce the routing resource usage.
- Layer assignment may also try to equally utilize the routing layers to prevent the crowding.

In this section, performance-driven ST construction, layer assignment and buffer insertion algorithms are first discussed separately. Then, these three algorithms are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

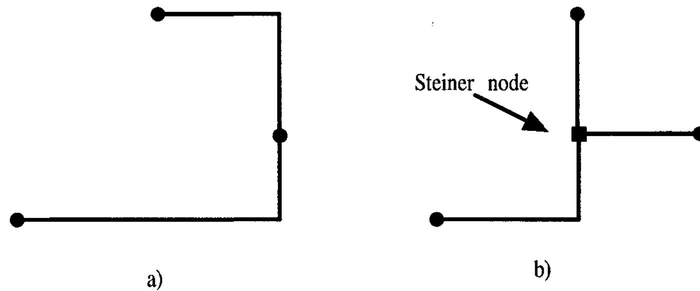


Figure 3.2: a) Spanning tree, b) Steiner tree

combined to obtain a simultaneous RT construction algorithm.

3.2.1 Routing-Tree Construction

To obtain the connectivity of a net with multiple pins, a tree-construction scheme is needed. If every edge in the tree connects only two pins, the tree is called a *spanning tree* [12] as illustrated in Figure 3.2 (a). But further cost reduction (mostly delay and length in VLSI routing) can be achieved using virtual pins, called *Steiner nodes*, which connects more than two pins together, hence called a Steiner-tree (ST) [12] as shown in Figure 3.2 (b). The objective of a ST construction algorithm is generally to minimize the total length of the tree to reduce the use of routing resources. But, high performance VLSI routing problem dictates constraints on the delay from source pin to the sink pins. Therefore, the primary objective should be the positive *delay slack* (or negative *delay violation*) at all sinks while the length becomes secondary objective. When the objective is related to the delay, we call the RT construction *Performance-Driven* or *Timing-Driven*. Let t_n be the actual delay from source pin to sink pin n; and rat_n be the required arrival time from source pin to sink n. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

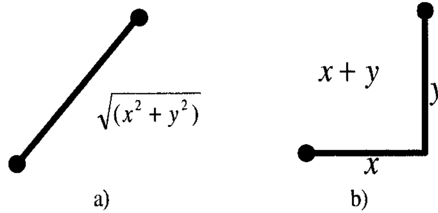


Figure 3.3: a) Euclidian distance, b) Manhattan distance

delay violation is defined as:

$$dv_n = t_n - rat_n \tag{3.1}$$

In order to meet the timing requirement, $dv_n \leq 0$ or $t_n \leq rat_n$ must be true. The rat value is usually given in a database file for the projected clock frequency. But, the delay t is a function of RT structure, driver size and the capacitance/resistance of the ST edges.

Today's over-the-cell VLSI routing uses multiple layers for the routing. Routing layers are usually restricted to horizontal or vertical routing directions (Manhattan routing) and the wires with angles are not allowed due to issues associated with sub-wavelength lithography. Consequently, the Manhattan distance is used rather than the Euclidian distance. Thus, ST with the edges restricted on horizontal or vertical direction (Rectilinear Steiner-Tree) has a special focus on VLSI routing. In this dissertation *Steiner-Tree* or *Routing-Tree* refers to a *Rectilinear Steiner-Tree* [12]. Figure 3.3 illustrates the Manhattan distance versus Euclidian distance. Manhattan routing suggests that each edge has to be assigned to a layer-pair to accommodate the route. Thus, in this dissertation, a *layer assignment* refers to a *layer-pair assignment* (one vertical, one horizontal)

The algorithm of RT construction is shown in Figure 3.4. For a given netlist, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  BuildPartialTree(T)
2   #one iteration
3  bestDV = -INF
4  bestLen = INF
5  foreach unconnected node n ∈ T
6    bestDVPin = INF
7    bestLenPin = INF
8    foreach edge e ∈ T
9      JoinNodeAtEdge(T, n, e);
10     awe(T);
11     if (DV(T) < 0 & Len(T) < bestLenPin)
12       bestLenPin = Len(T)
13       bestEdge = e
14     elsif (DV(T) < bestDV)
15          bestDVPin = DV(T)
16          bestEdge = e
17     DisjoinNode(T, n)
18   if (DV(T) < 0 & bestLenPin < bestLen)
19     bestLen = bestLenPin
20     bestNode = n
21   elsif (besDVPin > bestDV)
22        bestDV = bestDVPin
23        bestNode = n
24 JoinNodeAtEdge(T, bestNode, bestEdge)
```

Figure 3.4: Performance driven progressive routing tree construction

algorithm starts from the source node of each net and iterates through every sink node until the entire tree is constructed for every net in the netlist. At each iteration, *BuildPartialTree* chooses the most timing critical sink node from the unconnected sink nodes. If a net is a non critical net, i.e. there is no delay violation, *BuildPartial-Tree* chooses the sink node which minimize the total tree length. The most critical sink node is decided by tentatively connecting every sink node to the partial tree, and by calculating the maximum delay violation of the tree. *BuildPartialTree* joins the critical sink node to the existing tree such that the maximum delay violation of the tree is minimized or the total length of the tree is minimized if there is no delay violation. Althought any progressive RT construction can be adapted here, this scheme is similar to the CSRT approach [14] with *HBest* variant and without

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

global Slack Removal (GSR) post processing. The major difference lies in the order of unconnected sinks to be chosen to connect to the tree: We choose the most critical sink that minimize the delay violation whereas GSR chooses sink which minimizes the tree delay is chosen. Our preliminary experiments indicated slight improvement with our scheme.

For a partially constructed tree with i pins, the maximum number of edges is $2i - 3$ and the remaining number of pins is $n - i$ where n is the total number of pins. The possible permutations of tree to connect the rest of the pins can be expressed as:

$$\sum_{i=1}^{n}(2i-3)(n-i) \tag{3.2}$$

Hence, the complexity of the RT construction algorithms is $O(n^3)$ due to the inner search loop for the most critical pin.

The connection of a sink to an existing tree edge is obtained using the *hanan-point* [12] to achieve the secondary objective: length minimization. Hanan-point simply says that the optimal Steiner point location which minimizes the total tree length is the median of the three node coordinates as shown in Figure 3.6. Let $e_{u,d}$ be an existing edge connecting up and down nodes; and x_i and y_i be the x and y coordinates of the node i, respectively. Let $st = (median(x_u, x_n, x_d), median(y_u, y_n, y_d))]$ be the Hanan point, i.e., the new Steiner location. The optimal connection minimizing the total length is the edges, $e(u, st)$, $e(st, n)$ and $e(st, d)$ as illustrated in Figure 3.6. If the Steiner point is too close to any of the nodes, it can be joined with that node to minimize the via count. The increase in the total length and change in the delay will be negligible. The possible results are illustrated in Figure 3.7.

Hanan-point aims to have a minimal length tree. However, hanan-point may not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

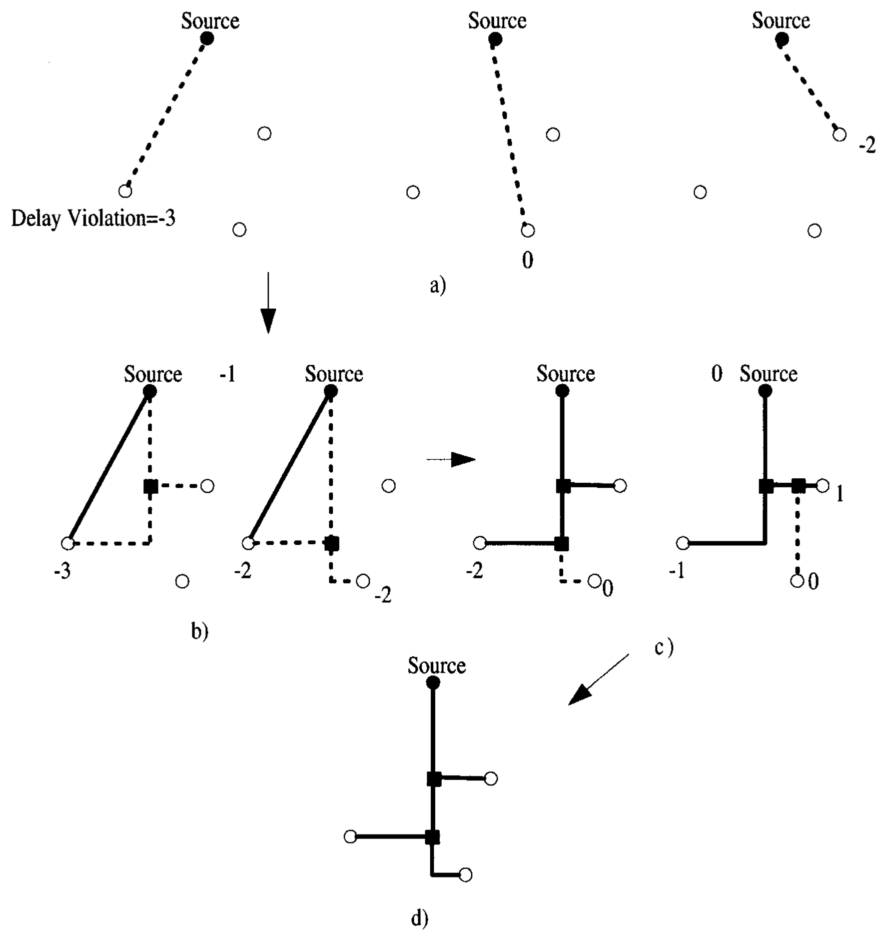


Figure 3.5: Progressive Steiner-tree construction,
a) sink 1,2,3 are tentatively connected to the source and delay violation is calculated, $e_{0,1}$ is chosen for it's minimum delay violation.
b) sink 2 and 3 are again tentatively connected to the only edge in the tree $e_{0,1}$, sink 3 is connected to the $e_{0,1}$ with the Steiner s_a
c) remaining sink 2 is tentatively connected to the existing edges $e_{a,1}$ and $e_{a,3}$. sink 2 is connected to the $e_{a,0}$ with the Steiner s_b
d) final Steiner-tree topology

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

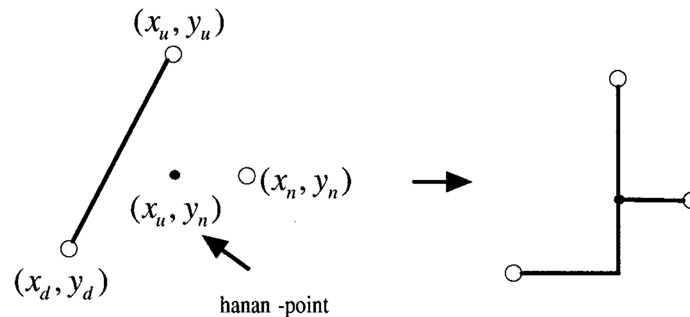


Figure 3.6: Hanan point

be optimal in terms of delay. Further optimization can be achieved by shifting the Steiner point closer to the upper node (shifting the sub-tree capacitance closer to the driver) as shown in Figure 3.8. Our experiments show that the search operation of the non-Hanan point increases the run-time substantially while the improvement on the delay is limited. Nevertheless, It can still be applied to critical nets.

In the RT synthesis method, the primary objective of delay violation minimization is achieved by connecting sinks such that the maximum delay violation is minimized; and the secondary objective of length minimization is achieved by choosing the hanan-point as the Steiner point. This progressive tree construction is particularly suitable for our purpose. It can easily be combined with layer assignment and buffer insertion algorithm in a concurrent fashion since it doesn't modify the partially constructed tree topology in each step while the algorithms iteratively modifying an existing sub-optimal tree (such as a spanning tree) may invalidate the current buffer insertion. A simple example is illustrated in Figure 3.9. The new edge introduced into the tree to improve the tree cost invalidates the already inserted buffer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

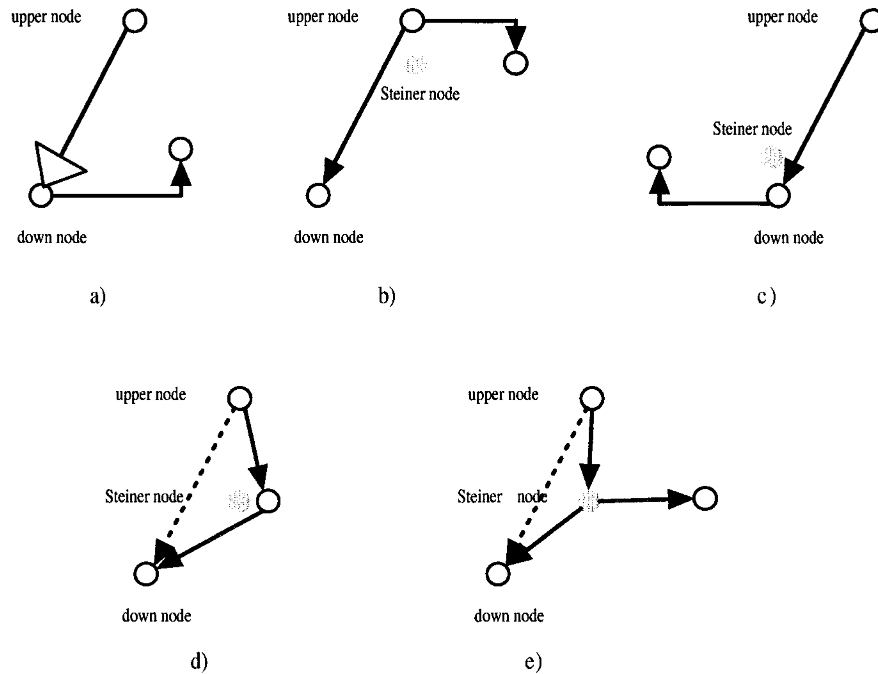


Figure 3.7: Possible join strategies,
a) $node_d$ is a root node (Driver or buffer output), Then direct connection of the node n to the down node will provide substantial delay improvement, node is joined to the tree with $e(d, n)$. No Steiner point is inserted.
b) $manh - distance(u, st) < \varepsilon$, node is too close to the upper node, node is joined to the tree with the $e(u, n)$. No Steiner point is inserted.
c) $manh - distance(st, d) < \varepsilon$, node is too close to the down node, node is joined to the tree with $e(d, n)$. No Steiner point is inserted.
d) $manh - distance(st, n) < \varepsilon$, node is too close to the Steiner node, node is joined to the tree with $e(u, n)$ and $e(n, d)$. No Steiner point is inserted.
e) otherwise, node is joined to the tree with (u, st), (st, d) and (st, n). Steiner node at Hanan-point.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

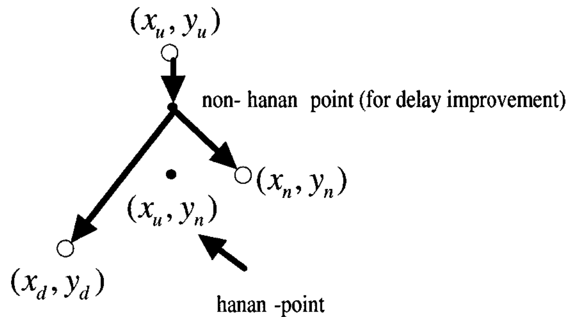


Figure 3.8: Non-Hanan optimization

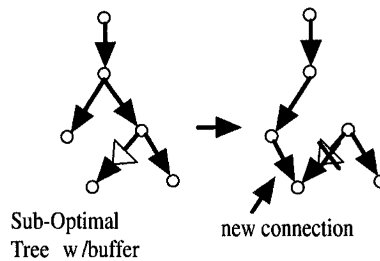


Figure 3.9: Iterative methods modifying the existing tree may invalidate the buffer insertion

3.3 Buffer Insertion and Removal

The buffer insertion problem is NP-hard [3], heuristics [17, 5] are often employed. Equal distance buffer insertion generally yields good *delay reduction-complexity* trade-off [23]. This is due to the fact that if the distance between buffers d_i is equal to one other, $\sum d_i^2$ is minimized provided that $\sum d_i$ is constant [51].

Interconnects in high speed VLSI circuits often fails to meet the timing requirement even with the shortest distance between two pins. This is due to the delay being not linear but quadratic with regard to the wire length as shown in Figure 3.10. An

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

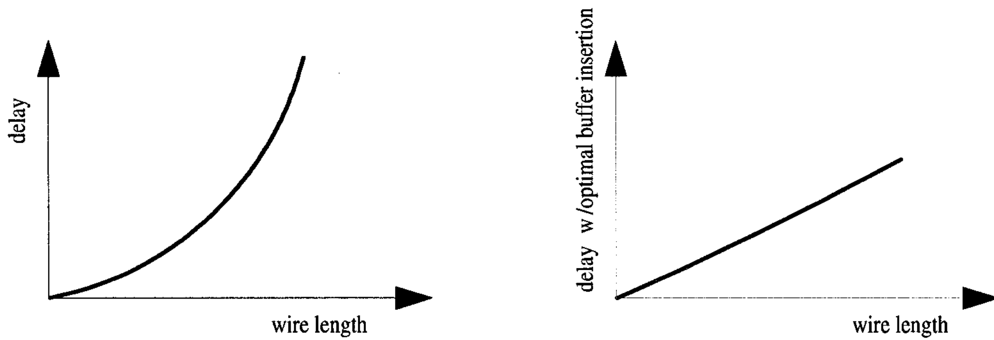


Figure 3.10: a)Delay vs. Length, b) Delay vs. Length w/buffer insertion

optimal buffer insertion makes the delay linear by shielding the down stream capacitance from the driver. The buffer insertion also improves the interconnect integrity, noise immunity and lateral coupling capacitances. The algorithm assumes buffers at Steiner points and in the middle of an edge if both upper node and down node has already been occupied by a driver or a sink respectively. The candidate buffer locations on a net are illustrated in Figure 3.11. Long edges requiring more than one

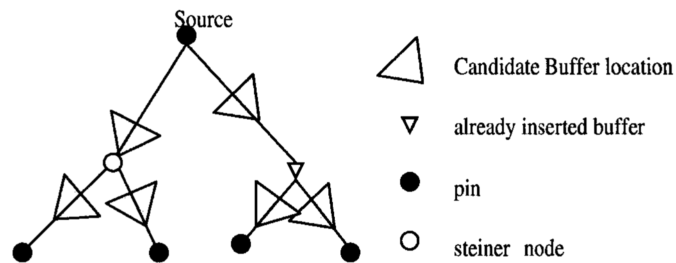


Figure 3.11: Candidate buffer locations

buffer is automatically segmented into smaller edges once a buffer is inserted on the edge. Realize that, at this stage, routing of the edges is not known; hence, the exact

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  InsertBuffer(N)
2  begin
3    bestDV = INF
4    bestEdge = φ
5    inserted = True
6    while (DV(N) > 0 and inserted)
7          foreach edge e ∈ N
8            b = InsertBufferAtEdge(N, e)
9            awe(N)
10           if (DV(N) < 0.9 * bestDV)
11             bestEdge = e
12             bestDV = DV(N)
13             inserted = true;
14           RemoveBuffer(N, b)
15         InsertBufferAtEdge(N, bestEdge)
16 end
```

Figure 3.12: Buffer Insertion

locations of the inserted buffers are unknown. Once the routing is done, exact buffer locations can be fixed.

The algorithm inserts a buffer which maximizes the delay violation reduction by evaluating delay reduction with respect to possible buffer locations. The algorithm evaluates buffer locations at Steiner points as well as on routing segments.

The overall combined buffer insertion algorithm consist of two phases. During the first phase, *InsertBuffer*, buffers are inserted for critical nets in the netlist. The second phase, where excessive buffers are removed as shown in Figure 3.18 (lines 25-29), re-optimizes the RTs by further trading off performance against power. At each iteration during the first phase, a buffer is tentatively inserted into each candidate buffer location. The buffer location with the best delay violation improvement is chosen. The iteration loop stops when the amount of delay improvement is below a preset threshold for all possible buffer locations, or the timing constraints for the net

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are met. When the first phase is completed, the second phase starts by re-visiting every net once to perform further performane power and delay trade-off. This is a desirable step due to the fact that inserted buffers on a partially built tree may be suboptimal after the rest of the tree is completed. The complexity of the overall buffer insertion algorithm is $O(n)$.

3.4 Layer Assignment

Multiple layers are reserved for the routing in VLSI chips. The properties of the routing layers vary greatly layer by layer forcing GRs to take the layers into account. Layer assignment of nets can be used to improve routing quality in three ways:

- Delay for the same length wires is better on the upper layers. Hence, critical nets can be assigned to the upper layers to improve timing.
- Coupling can be significantly less between layers. Therefore, crosstalk can be reduced by placing high-interference nets into separate layers.
- If a routing area on a particular layer is congested, some nets can be assigned to a different layer to reduce the congestion.

Although the accurate timing, congestion and cross-talk information are not available prior to routing, an estimated metric can be used to assign a net to a layer to improve the routing. Our algorithm uses estimated layer usage to calculate the utilization of layer. Layer usage is defined as:

$$U_l = \sum_{i \in B} \overline{b_i^l} / pitch^l + \sum_{j \in N+S} \overline{d_j^l}_{\;S} \, d_j^l \tag{3.3}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The layer usage incurred from blockages is calculated as the sum of the track lengths covered by the blockage. The b_i^l is the area of the blockage i at layer l. The $pitch^l$ is the distance between routing tracks at layer l. The layer usage incurred from existing routing N and current routing S is calculated by adding the horizontal (or vertical) distance d_j^l of the ST edges of each net.

```
1  LayerAssignment(S)
2  do
3      x = S.end()
4      y = x − 1
5      do
6          x = x − 1
7          y = y − 1
8          if l_x = l_y
9             if DV_x < DV_y
10               swap(x, y)
11         elsif x and y are on neighboring layers
12               if U_{l_x} > 1.1 * U_{l_y}
13                  l_x = l_y
14               elsif U_{l_x} < 0.9 * U_{l_y}
15                    l_x = l_y
16               elsif max(DV_{x^{l_y}}, DV_{y^{l_x}}) < max(DV_x, DV_y)
17                    swap(x, y)
18     while y ≠ S.begin()
19 while swapped
```

Figure 3.13: Performance-driven layer assignment

The algorithm in Figure 3.13 uses timing-driven bubble sort list as shown in Figure 3.14 to assign the critical nets to the upper layers. The layer usage is distributed among layers to prevent the crowding of one layer. Initially, all the nets are randomly inserted in a list. At each iteration of the bubble sort, two neighboring net is compared such that:

- If the layer assignment of both net is the same, the net with the higher delay

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

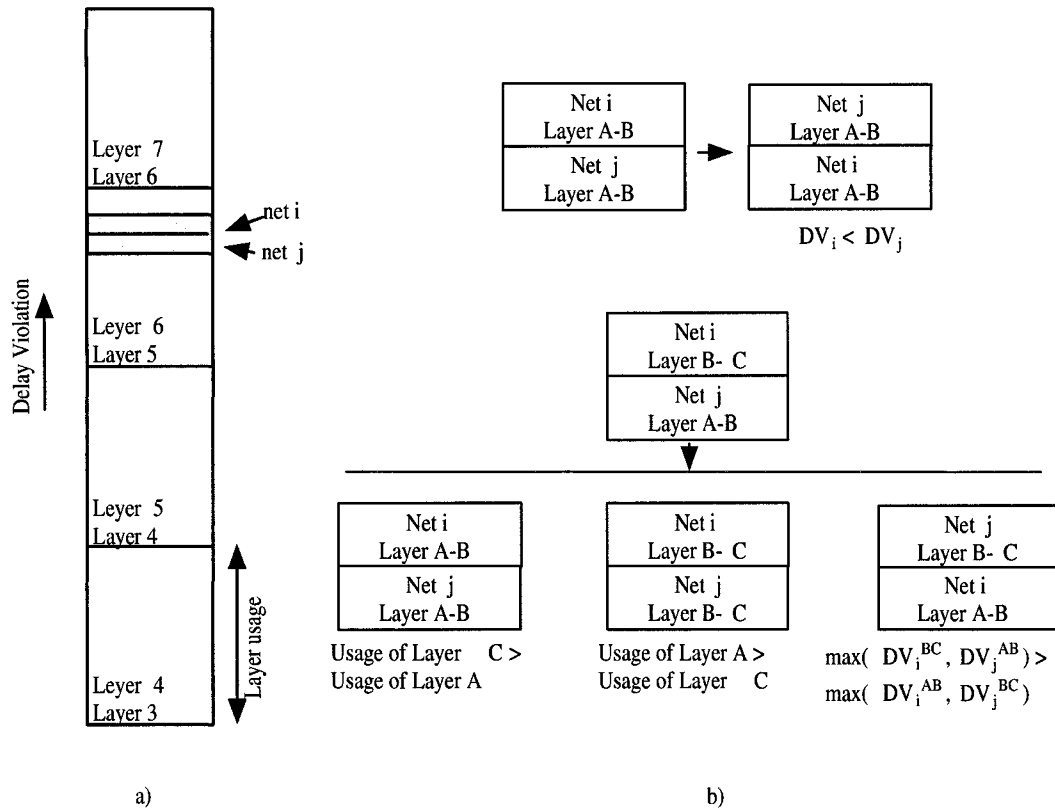


Figure 3.14: Layer assignment, a)Bubble sort list, b) Timing-Driven Swap operation

violation bubbles-up and vice versa.

- If the nets are in different layers, layer usages are first compared:
  - if the upper layer's usage is bigger than lower layer's usage, both net are assigned to the lower layer and vice versa.
  - If the layer usage is similar, then swap is done tentatively to compare the maximum delay violation. The best delay violation case is chosen for the permanent assignment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

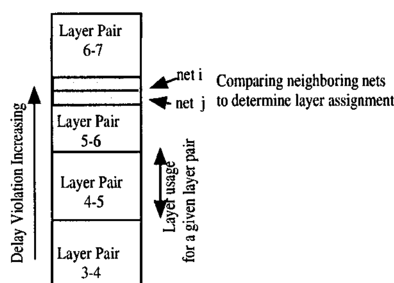


Figure 3.15: DV sort list for layer assignment

Some possible swap cases for the bubble sort are illustrated in Figure 3.14 (b). The process is repeated until there is no legal swap left in the list. Realize that layer change of a net only occurs on one layer of the layer-pair assigned to the net. For example, if net_a is assigned to the METAL3-METAL4 and net_b is assigned to the METAL4-METAL5, after the swap net_a is assigned to the METAL4-METAL5 and net_b is assigned to the METAL3-METAL4. Thus, only METAL3 and METAL5 usage is compared since METAL4 is common on both net's layer assignments.

Fig 3.13 gives the layer assignment algorithm. The resulting layer assignment moves critical nets to the upper metal layers. However, this is counter balanced by the requirement that the layer usage distribution be as homogeneously as possible among different layers to obtain less coupling and less congestion when the routing is actually performed. It is worth noting that nets in the list are swapped throughout the routing process in concurrent fashion with repeater insertion and RT construction.

Althgouht the worst case complexity of the sort algorithm is $O(n^2)$, sort is usually completed around $O(n)$ except during the first iteration where nets are randomly inserted into the list since delay (hence criticality) is unknown initially. In the subsequent iterations, list is mostly sorted and swap operation doesn't occur frequently.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5 Simultaneous Performance-Driven Buffered Routing-Tree Construction with Layer Assignment

To achieve the best overall result, three separate steps namely RT construction, layer assignment and buffer insertion, are combined to achieve concurrency. Simultaneous RT construction, layer assignment and buffer insertion enables the algorithm better optimize the tree topology. A simple example is given in Figure 3.16. The buffer inserted into the partial tree in Figure 3.16 (b) enables the next connection to better location reducing the tree cost. The intuition of simultaneous algorithm is that when the buffer insertion and layer assignment is applied to the partial tree, It will update the timing of the partially constructed tree allowing the next sink to connect to a better location. In other words, the performance-driven RT construction algorithm tends to connect each sink closer to the source in order to obtain the positive slack, possibly increasing the total tree length. By inserting buffers or improving the delay of the tree by layer assignment, sinks can now connect to the lower edges in the tree topology (or a closer edge to the sink), effectively reducing the total tree length.

As shown in the flow chart of the combined algorithm in Figure 3.17, initially all the nets are distributed randomly among layers. At each iteration, only one unconnected sink is joined to the partial tree and the delay violation is computed for each net. It is followed by layer assignment using the bubble sort to assign the critical nets to the upper layers. During buffer insertion, only one buffer is inserted to each net with a delay violation after layer assignment. Layer assignment and buffer insertion are repeated until no buffer can be inserted. Next iteration starts the process

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

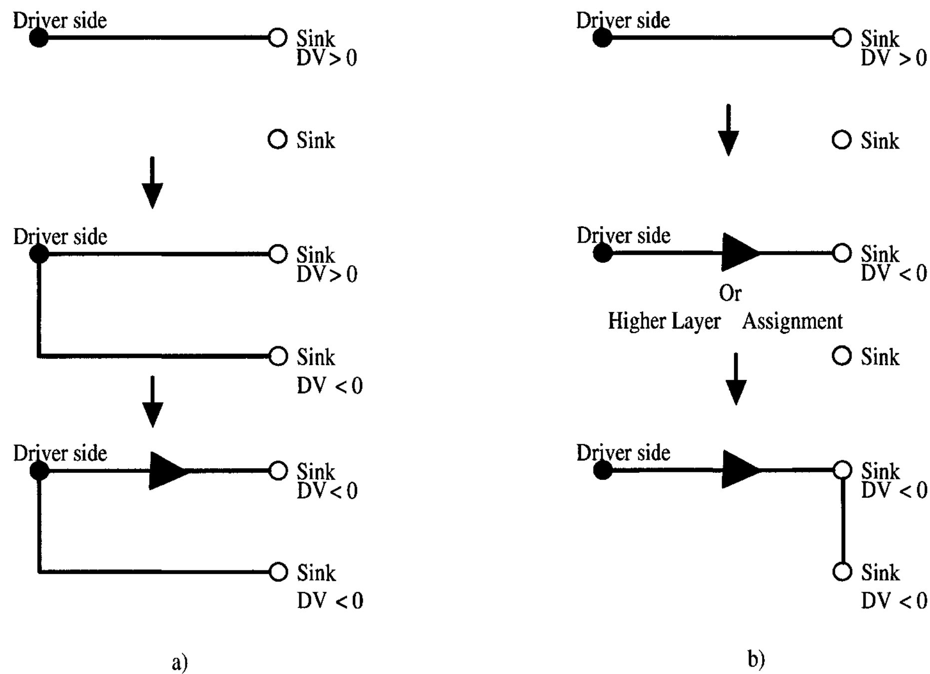


Figure 3.16: a) RT construction followed by buffer insertion, b) Concurrent implementation enables further optimization

again with the next unconnected sink is joined to the tree. Algorithm stops when all the sinks are connected and all the RTs are completed.

3.6 Two-Bend Global Routing with MIP, A Pre-Router

The algorithm accepts pre-constructed STs as input and centers on optimizing congestion with little deviation from original RTs as possible. In general, a MIP routing pre-calculates all possible RTs T_i for net i, and a 0-1 variable x_i^j represents each tree

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

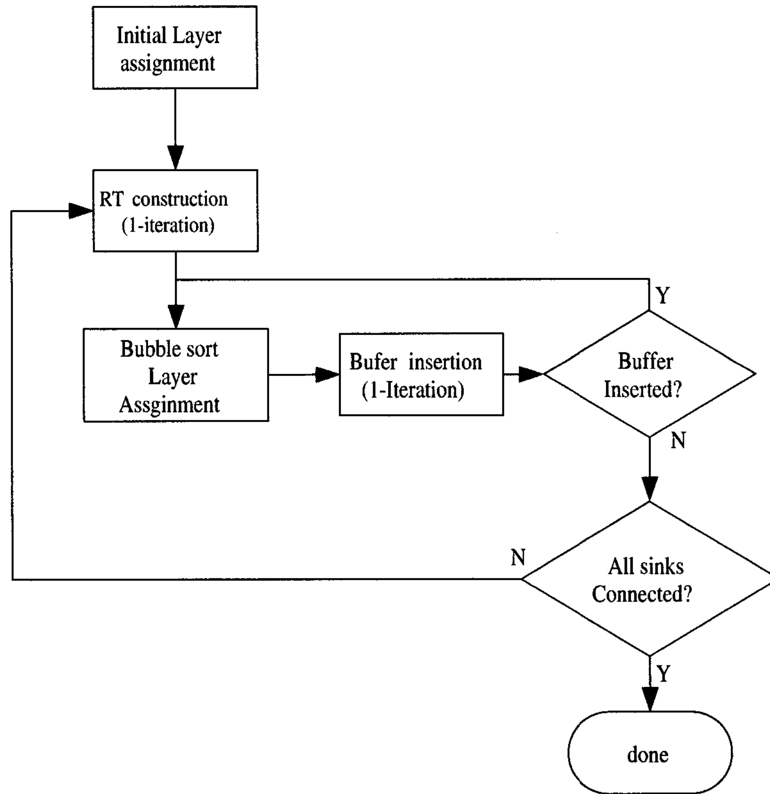


Figure 3.17: Simultaneous performance-driven buffered routing-tree with layer assignment

j for the net in MIP model:

$$\begin{aligned}
&\min W \\
&\text{subject to:} \\
\sum_{j \in T_i} x_i^j &= 1, \text{ for each net } i \qquad (3.4) \\
\sum_{i,j \in S_m} x_i^j &\leq W, \text{ for each routing segment } m \\
x_i^j &\in \{0, 1\}
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  LABI(S)
2  begin
3    S = All Nets
4    InitialLayerAssignment(S)
5    n_net = unconnectedSinks(net), net ∈ S
6    while (n_net! = φ, ∀ net ∈ S)
7          #Start BuildPartialTree
8          foreach net ∈ S
9             m_net = partialTree(net)
10            n_net = unconnectedSinks(net)
11            if (net is critical)
12               s = criticalSink(m_net, n_net)
13         else
14               s = minLengthSink(m_net, n_net)
15            m_net = m_net + s;
16            n_net = n_net − s;
17            GrowTree(m_net, s)
18         #End BuildPartialTree
19         do
20             AdjustLayerAssignment(S)
21             foreach net ∈ S
22                m_net = BldpartialTree(net)
23                InsertBuffer(m_net)
24         while (Buffer inserted for any net in S)
25 foreach net ∈ S
26   foreach buf ∈ net
27     RemoveBuffer(net, buf)
28     if (DV_net > 0)
29        ReinsertBuffer(net, buf)
30 end
```

Figure 3.18: Simultaneous buffered routing tree construction with layer assignment

where T_i is the set of two-bend paths for net i, and S_m is the set of RTs passes through routing segment m. The MIP optimization engine minimizes the maximum congestion W (objective) such that only one RT is chosen for each net. A few drawbacks are observed:

- The delay of the possible RTs is not accounted for in the MIP model, resulting in unpredictable RT delay.
- The MIP size and the run-time increases dramatically when all possible routes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are considered.

- The MIP routing minimizes the maximum congestion which fails to distribute the congestion homogenously. Thus, if there is a high demand area in the routing region, this high demand area will effectively set the lower limit of the objective; remaining routing areas will be left suboptimal.

To overcome the run-time bottle neck, divide-and-concur type hierarchic methodologies [2] are used. However, these heuristics yield suboptimal solutions and fail to optimize congestion adequately since optimization occurs only on a portion of the routing area. To overcome the sub-optimality of the hierarchical routing, we use a simplified MIP routing as a pre-router to optimize the congestion globally. A new aggressive congestion optimization method using zero-slack values is also introduced to further optimize the congestion for each routing segment to obtain the homogenous congestion distribution. Our method also improves on the run-time by only modeling the flexibilities of the RT edges (up to two-bend paths) rather than all the possible RTs as in the traditional MIP routers. Two additional side benefits are include:

- There is no need to pre-calculate all the possible RTs. Only the most optimal RT generated by the RT construction algorithm is used in the routing, further reducing the runtime for the RT construction algorithm.

- The resulting RT delay stays true to the initial RT delay due to the fact that the flexibilities are only used for the routing.

Moreover, since this is a pre-routing step, the tile size is much larger than the tile size used for the actual routing to provide the enough flexibility for the actual routing stage, reducing the MIP problem size further.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

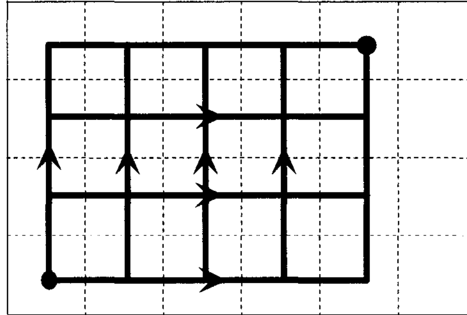


Figure 3.19: Two-Bend routing flexibilities

3.6.1 MIP Routing Model

The algorithm takes the *performance-driven buffered RT with layer assignment* output and models each edge in the RT using 2-bend flexibilities as shown in Figure 3.19. The MIP routing model is constructed as:

$$
\begin{aligned}
&\min W \\
&\text{subject to:} \\
&\sum_{j \in T_i} x_i^j = 1, \text{ for each wire } i \text{ (Wire Constraints)} \qquad (3.5) \\
&\sum_{i,j \in S_m} x_i^j \leq W, \text{ for each routing segment } m \text{ (Routing Constraints)} \\
&x_i^j \in \{0,1\}
\end{aligned}
$$

Where x_i^j is a 0-1 variable representing each 2-bend path j for wire i, T_i is the set of 2-bend paths for the same wire, S_m is the set of 2-bend paths going throught routing segment m, and W is the maximum congestion (objective) . The *wire constraints* here allows only one path per wire, while the *routing constraints* keeps the flow below

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

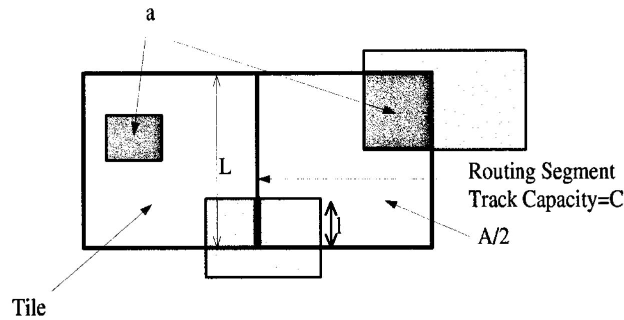


Figure 3.20: Effective capacity of routing segment is reduced by blockages

maximum congestion (objective). Left side of the routing constrains is called flow and represents the number of wires assigned to the tile boundary. Although the MIP model in Equation 3.5 looks similar to the traditional MIP routing model in Equation 3.4, each 0-1 variable x_i in our model represents a 2-bend routing flexibility for a edge rather than a pre-calculated whole RT.

3.6.2 Blockage Handling

In a typical routing area, at each layer, some areas are blocked for global routes such as clock and power. The proposed method incorporates these blockages into the routing constraints as existing flows in each tile.

In addition to the number of tracks occupied by the blockages and existing routing, a routing segment's proximity to a blockage can also reduce the effective capacity of the routing segment. Let l_i be the length of the intersection line of blockage i with the routing segment, and a_i be the intersecting area of blockage i with the tile associated with the routing segment as illustrates in Figure 3.20. If blockage instersect with routing segment, the flow on a routing segment due to the blockages can be calculated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as:

$$b = C\frac{\sum_i l_i}{L} \tag{3.6}$$

Otherwise, the flow on a routing segment due to the blockages in the vicinity can be estimated from the size of blockage as:

$$b = C\frac{\sum_j a_j}{A} \tag{3.7}$$

where C is the full capacity of the routing segment, L is the length of the routing segment, A is the total area of the neighboring tiles, i and j are the intersecting blockages with routing segment and the intersecting blockages with neighboring tiles respectively. This formulation calculates an estimated virtual flow on the routing segments incurred from the blockages. The blockages right on top of the routing segment directly reduce the routing capacity while the near-by blockages reduce the routing capacity proportional to their size. The flow of existing routes are also calculated in the same fashion by converting each existing route to rectangle like the blockages. Now, we can safely add the calculated flow b_m for the routing segment m consumed by the blockages into the MIP formulation as:

$$\begin{aligned} &\min W \\ &\text{subject to:} \\ \sum_{j \in T_i} x_i^j &= 1, \text{ for each wire } i \\ \sum_{i,j \in S_m} x_i^j + b_m &\leq W, \text{ for each routing segment } m \\ x_i^j &\in \{0,1\} \end{aligned} \tag{3.8}$$

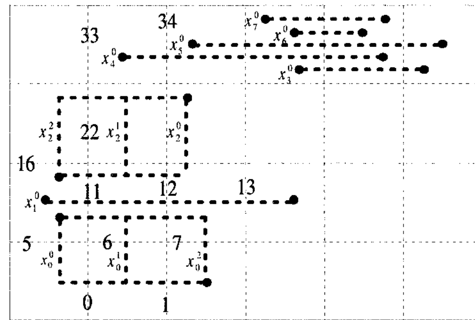
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the next section, we will expand our formulation for an aggressive congestion optimization method which can minimize congestion at each routing segment rather than minimizing the maximum congestion

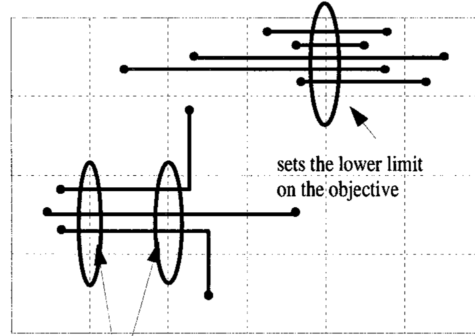
3.6.3 MIP Optimization

In a real word environment, highly congested areas are expected. It is even highly possible that maximum congestion can exceed the routing segment capacity. As seen from routing constraints, once a routing segment's flow is below maximum congestion, MIP optimization does not further influence the actual flow of the segment. This yields over-utilization of some routing segments while the others are under-utilized. A simple example of this situation is illustrated in Figure 3.21. The routing segment 36 is a high demand area in the routing region, thus the solution's lower limit is set to five. However, the MIP solution in Figure 3.21 (b) is a optimal feasible solution but is also a sub-optimal solution in terms of congestion distribution. The routing segments 11 and 12 are over-utilized. A better solution exists as shown in Figure 3.21 (d). To distribute the congestion homogenously and minimize the congestion for each routing segment, we propose an iterative optimization method using zero-slack constraints to determine which routing segments can be further optimized. This is achieved by substituting a variable W_m for the objective variable W in the routing

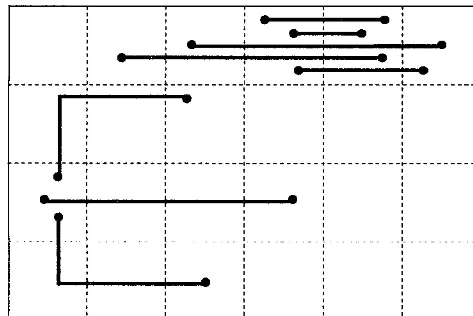
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



a)



c)



d)

b)

min : W

subject to :

$x0 : x_0^0 + x_0^1 + x_0^2 = 1$

$x1 : x_1^0 = 1$

$x2 : x_2^0 + x_2^1 + x_2^2 = 1$

$x3 : x_3^0 = 1$

$x4 : x_4^0 = 1$

$x5 : x_5^0 = 1$

$x6 : x_6^0 = 1$

$x7 : x_7^0 = 1$

$0 : x_0^0 < W$

$1 : x_0^0 + x_0^1 < W$

$5 : x_0^0 < W$

$6 : x_0^1 < W$

$7 : x_0^2 < W$

$11 : x_0^1 + x_0^2 + x_1^0 + x_2^0 + x_2^1 < W$

$12 : x_0^1 + x_0^2 + x_1^0 + x_2^0 + x_2^1 < W$

$13 : x_1^0 < W$

$16 : x_1^2 < W$

$17 : x_1^1 < W$

$18 : x_1^0 < W$

$22 : x_1^2 < W$

$23 : x_1^1 + x_1^2 < W$

$34 : x_4^0 < W$

$35 : x_4^0 + x_5^0 < W$

$36 : x_3^0 + x_4^0 + x_5^0 + x_6^0 + x_7^0 < W$

$37 : x_3^0 + x_5^0 < W$

Figure 3.21: a) Routing Problem with possible 2-bend paths b) MIP model c) Optimal MIP solution, but congestion is not optimized on tile boundaries 11 and 12 d) Better Solution

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  OptimizeMIP(MIP)
2  do
3      Optimize MIP using CPLEX Optimization Engine
4      foreach zero-slack row in the MIP
5         Set Right Hand Side to the objective value - 1
6         if Mip is infeasible
7            then
8                  Set Right Hand Side to the Objective
9            else
10                 Set Right Hand Side to the Objective Value
11 while Objective > 0
```

Figure 3.22: Iterative MIP optimization

constraints and moving the inequalities to the *optimization constraints*:

$$\begin{aligned}
&\min W \\
&\text{subject to:} \\
\sum_{j \in T_i} x_i^j &= 1, \text{ for each wire } i \qquad (3.9) \\
\sum_{i,j \in S_m} x_i^j + b_m &= W_m, \text{ for each routing segment } m \\
W_m &\leq W, \text{ optimization constraints} \\
x_i^j &\in \{0, 1\}
\end{aligned}$$

A *slack value* for a MIP constrain is defined as the difference of the bound and the actual value of the constraint. Hence, slack of a general MIP constraint $\sum X < Y$ is defined as $Y - \sum X$. Realize that if the slack is zero, the constraint is bounded and any decrease on the bound value Y can make the MIP problem unfeasible.

The method exploits this zero-slack property to determine which routing segments are full and can't be optimized further. The optimization is performed as illustrated in Figure 3.22 using *optimization constraints* as follows: At each iteration, zero-slack

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

optimization constraints are in focus: A zero-slack constraint simply says that flow in that routing segment is equal to the objective value. There is a possibility that the routing segment with the zero-slack value is minimized to its absolute minimum. It is further examined by testing for the feasibility of the problem by tentatively reducing the upper limit by one. If the new MIP problem is infeasible, then the routing segment can't be optimized further. The algorithm continues to optimize the rest of the routing segments with the new MIP problem. When the upper limit of the all optimization constraints becomes constant, optimization is completed. The following example illustrates how the zero slack values is used to modify the MIP model in Figure 3.21 (b) to obtain the optimal solution in Figure 3.21 (d).

Step 1 **Values:**

Objective : $W = 5$,

Wires : 1 set $(x_0^2, x_1^0, x_2^0, x_3^0, x_4^0, x_5^0, x_6^0, x_7^0)$

Routing Segments : $W_7 = 1, W_{11} = 3, W_{12} = 3, W_{13} = 1, W_{18} = 1, W_{34} = 1, W_{35} = 2, W_{36} = 5, W_{37} = 2$

Zero Slack Rows : $W_{36} \leq W$

Modifications to the MIP Problem:

$W_{36} \leq 5 - 1$: Infeasible MIP

Set $W_{36} \leq 5$

Step 2 **Values:**

Objective : $W = 3$,

Wires : 1 set $(x_0^2, x_1^0, x_2^0, x_3^0, x_4^0, x_5^0, x_6^0, x_7^0)$

Routing Segments : $W_7 = 1, W_{11} = 3, W_{12} = 3, W_{13} = 1, W_{18} = 1, W_{34} =$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$1, W_{35} = 2, W_{36} = 5, W_{37} = 2$

Zero Slack Rows :$W_{11} \leq W, W_{12} \leq W$

Modifications to the MIP Problem:

$W_{11} \leq 3 - 1$: feasible MIP

$W_{12} \leq 3 - 1$: feasible MIP

Set $W_{11} \leq 2, W_{12} \leq 2$

Step 3 **Values:**

Objective : $W = 2$,

Wires : 1 set $(x_0^0, x_1^0, x_2^0, x_3^0, x_4^0, x_5^0, x_6^0, x_7^0)$

Routing Segments : $W_0 = 1, W_1 = 1, W_5 = 1, W_{11} = 2, W_{12} = 2, W_{13} = 1, W_{18} = 1, W_{34} = 1, W_{35} = 2, W_{36} = 5, W_{37} = 2$

Zero Slack Rows : $W_{11} \leq W, W_{12} \leq W, W_{35} \leq W, W_{37} \leq 2$

Modifications to the MIP Problem :

$W_{11} \leq 2 - 1$: feasible MIP

$W_{12} \leq 2 - 1$: feasible MIP

$W_{35} \leq 2 - 1$: infeasible MIP

$W_{37} \leq 2 - 1$: infeasible MIP

Set $W_{11} \leq 1, W_{12} \leq 1$

Step 4 **Values:**

Objective : $W = 1$,

Wires : 1 set $(x_0^0, x_1^0, x_2^2, x_3^0, x_4^0, x_5^0, x_6^0, x_7^0)$

Routing Segments : $W_0 = 1, W_1 = 1, W_5 = 1, W_{11} = 1, W_{12} = 1, W_{13} =$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$1, W_{16} = 1, W_{22} = 1, W_{23} = 1, W_{34} = 1, W_{35} = 2, W_{36} = 5, W_{37} = 2$

Zero Slack Rows : $W_0 \leq W, W_1 \leq W, W_5 \leq W, W_{11} \leq W, W_{12} \leq W, W_{13} \leq W, W_{16} \leq W, W_{22} \leq W, W_{23} \leq W, W_{34} \leq W$

Modifications to the MIP Problem:

$W_0 \leq 1 - 1$: infeasible MIP

$W_1 \leq 1 - 1$: infeasible MIP

$W_5 \leq 1 - 1$: infeasible MIP

$W_{11} \leq 1 - 1$: infeasible MIP

$W_{12} \leq 1 - 1$: infeasible MIP

$W_{13} \leq 1 - 1$: infeasible MIP

$W_{16} \leq 1 - 1$: infeasible MIP

$W_{22} \leq 1 - 1$: infeasible MIP

$W_{23} \leq 1 - 1$: infeasible MIP

$W_{34} \leq 1 - 1$: infeasible MIP

Optimization Completed; the set $(x_0^0, x_1^0, x_2^2, x_3^0, x_4^0, x_5^0, x_6^0, x_7^0)$ is the optimal solution in Figure 3.21 (d).

3.7 Network-Flow Routing

A divide-and-concur method hierarchically bisects the chip area and performs the routing on the bisect line [2]. Hence, the routing problem becomes an assignment problem and can be modeled with a network-flow model which can be solved hundreds of times faster than equivalent MIP model due to the special LP structure of the network-flow models. Two main drawbacks of this approach are:

- Optimization occurs only on the bisect line, resulting suboptimal solution. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

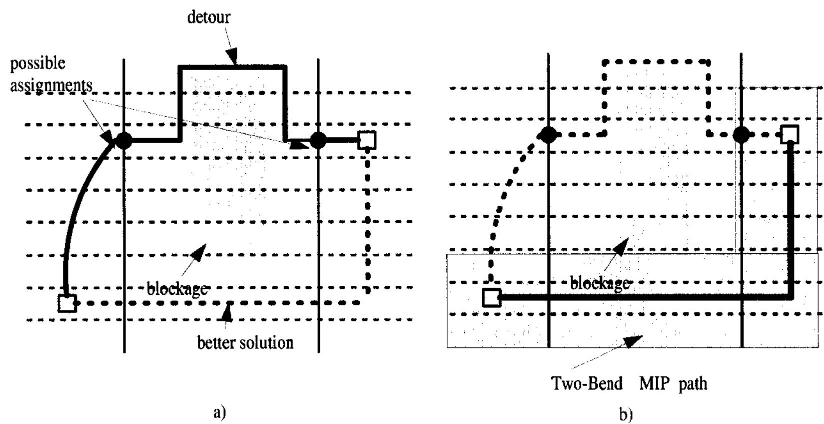


Figure 3.23: a) Suboptimal result of a divide and concur method, fails to avoid blockage b) Pre-Routing, MIP routing, helps optimize congestion globally

can cause overflows and/or unnecessary detours as shown in Figure 3.23 (a). It fails to avoid the blockage.

- In general, overflow is prevented by setting the arc capacity to the routing segment capacity. Thus, there is no congestion optimization, resulting in over-utilized and under-utilized routing segments. Figure 3.24 (a) shows a possible sub-optimal routing assignment by a hierarchical network-flow based router. Although the solution is feasible and there is no overflow, it is suboptimal in terms of routing segment utilization.

The two-bend MIP based pre-routing mostly overcomes the first problem by optimizing the congestion globally as illustrated in Figure 3.23 (b). A new iterative congestion optimization method using zero-slack values addresses the second problem. Congestion is optimized throughout the bisect line rather than focusing on overflow prevention. A simple example is illustrated in Figure 3.24 (b). Via minimization is also addressed in the network-flow router by assigning higher cost to bends.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

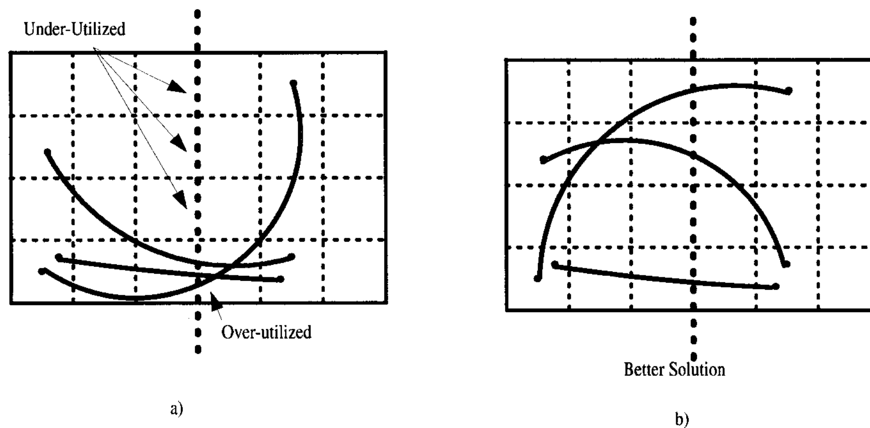


Figure 3.24: a) Suboptimal result of network-flow router, even when there is no overflow, b) Better solution

3.7.1 Network-Flow Routing Model

The network model hierarchically bisects the routing area and builds a network model for each bisect line as shown in Figure 3.25 (a) and (b), respectively. The wires crossing the bisect line can be routed by assigning the wire to a routing segment on the bisect line.

The network is constructed as follows:

- The *source node s* is placed as the starting point of the network-flow. The supply value is set to the number of nets involved in the network.
- A *routing segment node* for each routing segment is inserted into the network. The routing segment node is connected to the source node with the capacity of the routing segment. The flow value of these arcs represents the number of tracks occupied by the wires.
- A *wire node* for each wire crossing the bisect line is inserted. The wire node is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

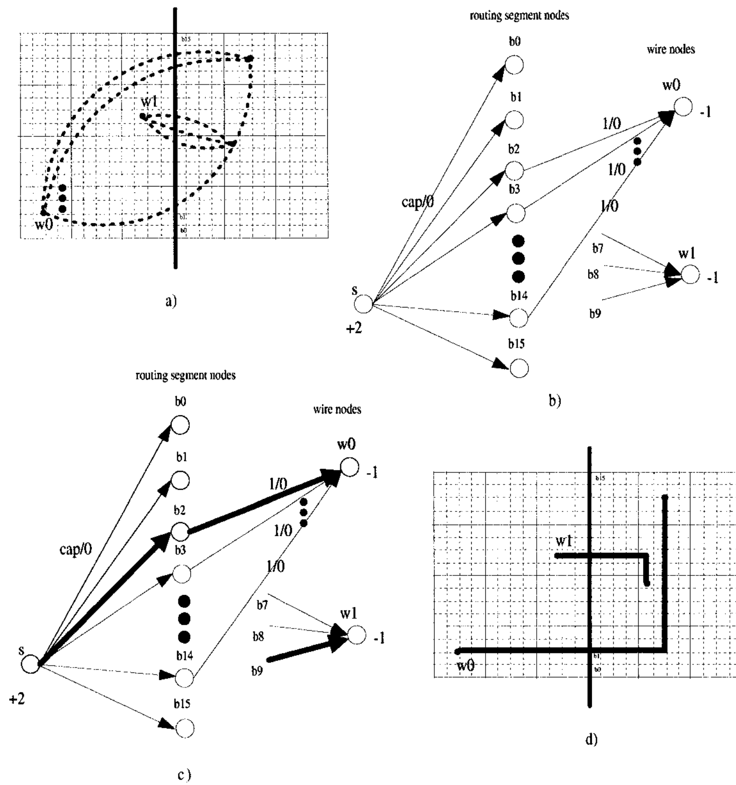


Figure 3.25: a) Bisecting, b) Network-flow routing model

connected to the routing segment nodes such that those routing segments are the candidate crossing points for that wire. These candidate locations are dictated by the two-bend MIP pre-routing and the tile size used in the MIP routing as illustrated by grayed tiles in Figure 3.26 (a). By doing so, the network-flow router follows the rough path dictated by the output of the MIP router as shown in grayed path in Figure 3.26 (c). It is worth to mention that the tile size in the MIP stage should be kept several times larger than network-flow stage to provide the enough flexibility in the network-flow routing. This is due to the design that network-flow router's input should be the routing in a coarser scale

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the MIP stage. The capacity of the arcs set to 1 since each wire occupies a singe track in the routing segment. The demand value of the wire node is also set to 1 to allow the wire to be routed.

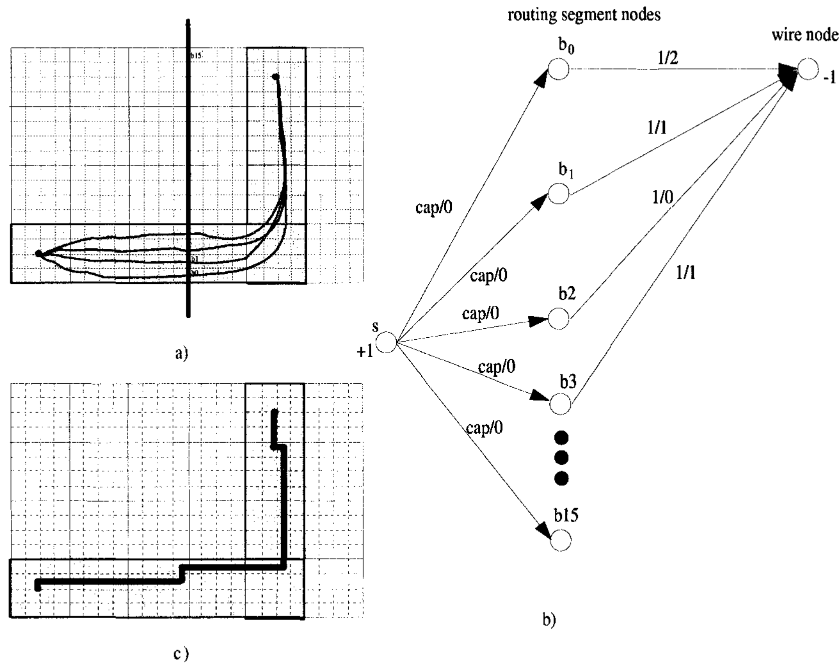


Figure 3.26: a) Two-bend guide for network routing, b) Network-flow routing model c) After network-flow routing

The advantage of using MIP pre-routing is illustrated in Figure 3.23. Without the MIP router, network-flow can fail to avoid the blockage in the middle in Figure 3.23 (a). In Figure 3.21 (b), MIP guides the network-flow router through less congested area.

The network-flow construction is similar to the Spatnekar's algorithm [2]. However, our method determines the possible assignments of the soft-edges by the output of the MIP pre-router, while the conventional method extends the possible assign-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ments until the network becomes feasible. A network-flow routing without the pre-router can result in severe suboptimal solutions as illustrated in Figure 3.23 and Figure 3.24. Spatnekar also introduced slideable-Steiner points by inserting a dummy node to represent the location of a Steiner point to determine which side of the bisect line the Steiner point is at. A gain value is assigned to each input arc of the dummy node as shown in Figure 2.31 to ensure only one location of the Steiner point is chosen in the solution. Although slideable Steiner points exploits the flexibility of the RT topology to improve congestion, their network-model with slideable Steiner points can not be solved by traditional network solvers. It can only be solved by a MIP or Fleischer-Wayne algorithm [49] which is slower than network solvers. Therefore, their formulation is more in line with the multi-commodity flow formulation [44]. The difficulties with the conventional multi-commodity flow formulation are its limitation to handle large problem sizes and the lack of efficient methods to solve large problems. One key advantage of the network-flow model over the multi-commodity models is that it can be solved hundreds of times faster [52]. Hence, we can afford an exact solution, there is no need of approximation algorithms such as randomized rounding [44] in order to achieve faster run times. As explained in the following sections, our method optimizes congestion by tightening the routing segment capacities iteratively, while the traditional method settles with no overflow on the routing segments.

3.7.2 Via Minimization

Two pins connected by a routing segment defines a bounding box. The minimum number of vias are inserted if the wire is routed within the bounding box. Routing outside the bounding box requires additional vias. Via minimization can be achieved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

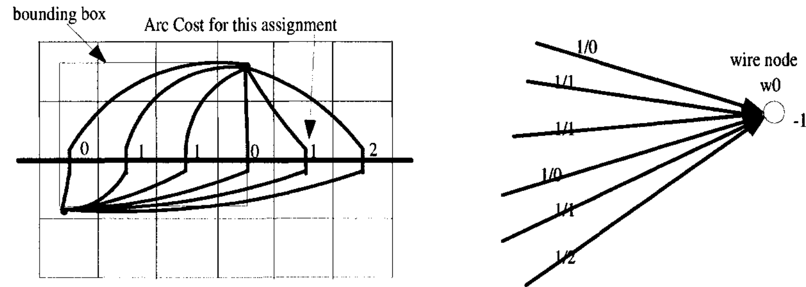


Figure 3.27: Via minimization, a) Possible assignment, b) The cost values in the network model for via minimization

by assigning a cost value to the arcs in the network model. The cost value increases for arcs that are outside the bounding box. This cost setting encourages the network to choose straight paths if possible. Figure 3.27 illustrates the increasing cost associated with arcs that are outside the bounding box.

3.7.3 Blockage Handling

The number of tracks occupied by the blockages and existing routing including clock and power is calculated as exactly in the previous section (MIP pre-routing). Since these blockages reduce the capacity of the routing segments and induce extra flow, we set the demand at routing segment nodes to b as calculated in Section 3.6.2. By doing so, a constant flow is introduced into the arcs connecting the start node to the routing segment nodes, effectively reducing the routing segment's capacity. The modified network-flow model is shown in Figure 3.28 with blockages included.

One can directly reduce the arc capacity instead of adding the demand values, but it is necessary to separate the blockage flow and actual flow for the purpose of network optimization method introduced in the next section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

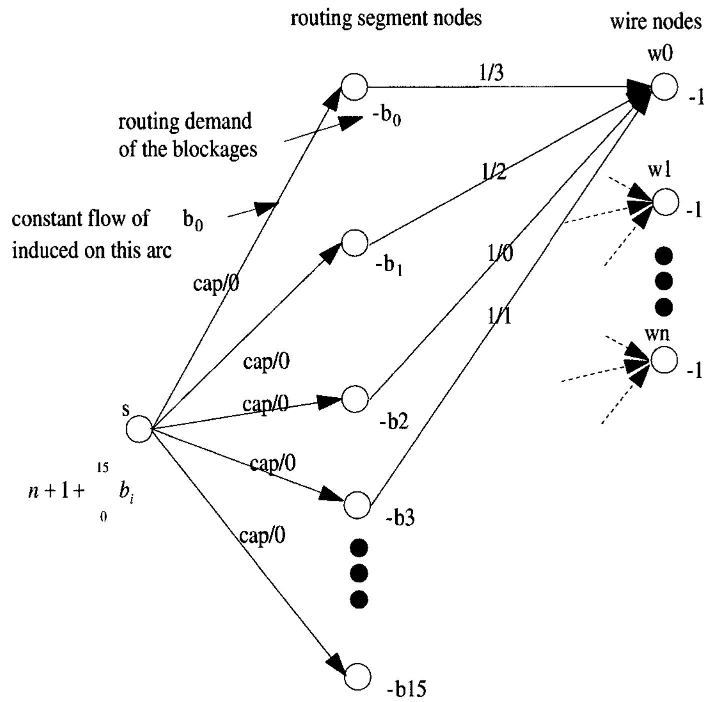


Figure 3.28: Network-flow routing model w/blockage handling, a constant flow is introduced to the routing-segment arcs by the demand on routing segment nodes

3.7.4 Network Optimization

A network model as described above can only aim to have no overflow on routing segments with the same disadvantages of the MIP routing explained in the previous section. It fails to distribute the congestion more homogeneously resulting in under-utilized and over-utilized routing segments as illustrated in Figure 3.24.

The new network optimization method aims to minimize the flow on each routing segment in a similar iterative fashion used in the MIP pre-routing. This can be achieved by slowly tightening the bound of the arcs connecting from the source node to the routing segment nodes. At each iteration, a binary search is performed for the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

arc bounds. When the final solution is reached, zero-slack values are again used to determine which routing segment is fully utilized. Similar to that in the two-bend MIP pre-routing, the capacity on a zero-slack arc is tentatively reduced slightly to further examine the feasibility. If the network is still feasible, the routing segment can be optimized further. Otherwise, lower and upper bounds are fixed for that routing segment and optimization continues with the other routing segments. This process repeats until all the arc boundaries are fixed.

Both our algorithm and the algorithm in [2] use network flow as a model for GR. However, significant differences exist between two approaches:

- The possible assignment of wire to routing segments is determined by the MIP router in the algorithm rather than being determined by the bounding box of the wire in [2]. The algorithm tends to produce a more globally optimum assignment in terms of congestion.

- The implementation of slideable Steiner point in [2] results in the use of pseudo nodes in the network flow model. Such a model is not suitable for conventional network solvers such as ILOG's solver [53]. It can only be solved by a MIP solver or by Fleischer-Wayne algorithm [49]. The network-flow model is simpler and can be solved more effectively by conventional network solvers resulting in a much shorter run-time.

- The formulation in [2] aims at a feasible solution whereas our algorithm uses the binary search and manipulation of LP slack values to further optimize congestion on every routing segment even if the maximum feasible congestion can not be further reduced.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1  OptimizeNet(NET)
2  up = INF
3  down = 0
4  do
5      do
6          foreach unmarked arc in the Network
7              Set arc capacaity to the (up + down)/2
8              if Network is infeasable
9                  then
10                         down = (up + down)/2
11                 else
12                         up = (up + down)/2
13     while up − down > 0
14     foreach zero-slack arc in the Network
15         Set arc capaciticy to the Objective Value - 1
16         if Network is infeasable
17             Mark the arc as done
18 while #arcs unmarked  > 0
```

Figure 3.29: Network-flow Congestion Optimization

- The algorithm handles blockage by adding "demand" values on the associated arcs in the network flow model. This allows us to incorporate LP slack manipulation to further optimize congestion with a minimum impact on timing. The model in [2] handles blockage by setting capacity to the associated arcs. If the network is infeasible, additional arcs are added to the network until it is feasible. Althought Spatnekar's model considers the congestion during network building process utilizing min-cut and arc expansion, congestion is not considered in the network solving process. However by incorporating LP slack manipulation and layer assignment tuning (discussed in the next section), handling route detours due to blockages in our algorithm takes into account both congestion and timing.

Realize that the congestion minimization is the primary objective while the via

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

minimization is the secondary objective. Too aggressive congestion optimization will render via minimization ineffective. Hence, the algorithm should stop when current congestion value reaches at a certain percentage of the full routing segment capacity. This value can be preset depending on the designer's preference.

3.8 QP Layer Assignment Tuning

3.8.1 Introduction

The layer assignment algorithm used in the simultaneous performance-driven buffered RT construction with layer assignment in Section 3.2 is done without the knowledge of congestion and routing. Hence it is suboptimal and can cause overflows. But, it still greatly improves the final routing quality. When both the MIP routing and the network routing is completed, more accurate congestion information is available and layer assignment can be fine tuned.

This section explains the construction of the QP model and the interference metric used to measure congestion and cross-talk. After solving the both the MIP model and the network-flow model to obtain the result of global routes, it is possible that overflow exists. For any routing segment with overflow, a layer assignment tuning step is performed reduce and eliminate the amount of overflow. The layer assignment tuning is carried out using a quadratic programming (QP) model to determine which nets should be moved to which layer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

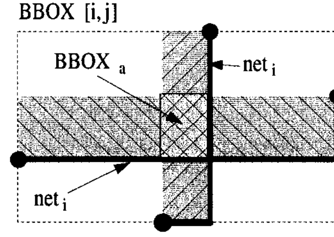


Figure 3.30: Potential Cross-talk

3.8.2 QP Model

The QP model incorporates the existing routing, blockage and interference between nets. Given two nets shown in Figure 2.9, interference between the two nets is defined as the ratio of the intersection area (marked $BBOX_a$) to the bounding box area (marked $BBOX[i,j]$). The interference between two nets is a measure of potential cross-talk and congestion. Nets with high interference are likely to have more cross talk and routing conflict, and, therefore, should be routed on different routing layers if possible. In order to take the existing routing and blockages into account, we define a congestion metric for use in the QP model as the ratio of the total flow to the total flow capacity. Notice that total flow already includes the blockages as virtual flows as explained in Section 3.6.2 and 3.7.3. Let b_i^l be congestion in the bounding box of a net i when the net is assigned to layer l. Thus, the congestion can be calculated from the routing segment flows inside the bounding box of the net as:

$$b_i^l = \frac{\sum_{k \in BBOX(i)} flow_k^l}{\sum_{k \in BBOX(i)} cap_k^l} \tag{3.10}$$

where $flow_k^l$ is the current flow value of segment k at layer l and cap_k^l is the routing capacity of segment k at layer l. Let $w_{i,j}$ be interference metric: w_i^j measures the ratio

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of intersecting area to the bounding box of net i and net j. It is simply calculated as:

$$w_{i,j} = \frac{area(BBOX_a)}{area(BBOX[i,j])} \tag{3.11}$$

The QP layer assignment model can now be constructed as:

$$\begin{aligned} &\text{minimize} \quad \textstyle\sum_i \sum_l b_i^l x_i^l + \sum_i \sum_j \sum_l w_{i,j} x_i^l x_j^l \\ &\text{subject to:} \qquad\qquad (3.12) \\ &\qquad \textstyle\sum_l x_i^j = 1 \text{ for each net } i \\ &\qquad x_i^j \in \{0,1\} \qquad\qquad (3.13) \end{aligned}$$

where x_i^l set to one if net i is assigned to layer l.

Constrains in the QP model enable each wire to be assigned only to one layer. The first term of the objective function in Equation 3.12 minimizes the total congestion. Hence, it attempts to assign nets to less congested layers. The second term minimizes the total interference between nets. Therefore, resulting assignment puts high interference nets into different layers.

Since solving the QP model is hard because of the size of the quadratic coefficients, only non-critical nets from congested areas can be afforded to be included in the QP model. Moreover, if all the nets were included in the QP model, the result would invalidate the timing-driven layer assignment performed during RT construction. The QP model is not designed to be timing-driven since incorporating delay into the QP model would be impractical, inaccurate, and costly. The purpose of the QP layer assignment is to clean-up the overflows with a minimum impact on the performance-driven layer assignment and the net delay.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

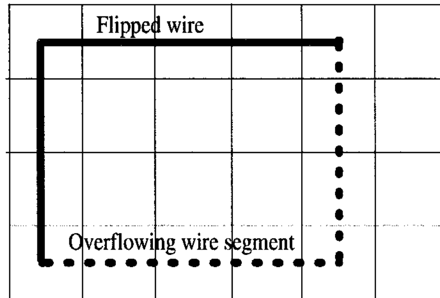


Figure 3.31: L Flipping

3.9 Post Processing Heuristics

3.9.1 Introduction

One of the disadvantages of the network-flow based router is that, even though they perform better at global level, they are not able to focus on one net at a time, causing overflows in local routing areas. Most of those overflows can easily be cleared by a series of post-processing heuristics.

In this section, we use four post-processing schema: edge flipping, detour, via pad elimination and maze router.

3.9.2 Edge Flipping

Sometimes an overflow can easily be cleared by simply flipping an L shaped edge as illustrated in Figure 3.31. Naturally, the flipped L shape path is checked for any overflows. Since flipping will not change the RT length, we can assume delay is not changed. Still a small amount of change in the delay can be expected due to the different sheet resistance and capacitances of the layer-pair assigned to the L shaped

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

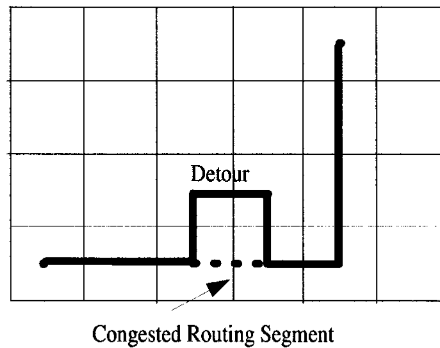


Figure 3.32: Detour

edge (horizontal length of the L shape becomes vertical and vice versa).

3.9.3 Detour

If a straight wire is causing overflows on a small portion of the wire segment, the overflowing segment can detour around congested area. This is achieved by placing a U shaped path around the congested area as shown in Figure 3.32 . Note that detour will increase the overall delay of the net.

3.9.4 Via Pad Elimination

The layer assignment algorithm can assign one wire to a certain layer while the connecting segments of the wire can be assigned to a non-neighboring layer. The connection between segments is achieved by inserting a via between the inner layers of the edge as shown in Figure 3.33

If the layers between assigned layers contains blockages or highly congested as shown in Figure 3.33 (a), Via insertion may not be possible. The involved wires have to be reassigned to different layers or neighboring layers to eliminate the via pad

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

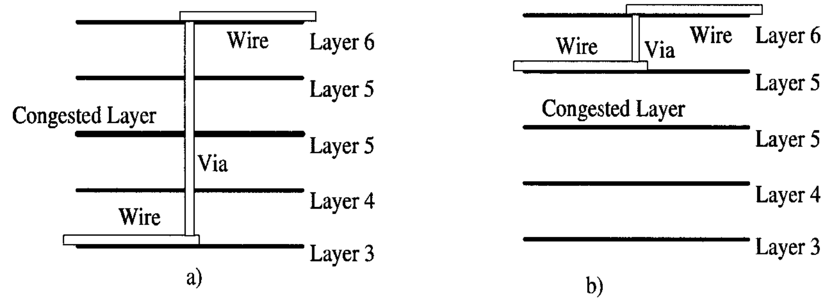


Figure 3.33: a) Via pad violation on Layer 5 b) the elimination of the violation

violation as shown in Figure 3.33 (b). Our heuristic walks through each wire and detects those via pad violations and reassigns wires to different layers such that there is no via pad violation and no overflow is incurred in the new layers.

3.9.5 Maze Router

If everything fails, we turn to a sequential router, maze router, to clear up the remaining overflows. A wire segment with overflow is ripped with a slightly larger than the wire segment's length as illustrated in Figure 3.34. The maze router is then applied to reroute the ripped wire segments.

The maze router algorithm simply finds the shortest path between two points by starting a wave-front from the source node. When the wave-front hits the destination point, shortest path is traced back connecting the two points. The cost of moving from one tile to the neighboring tile is set to 1 as the distance. However, a higher cost is incurred for layer change to minimize the via count. The maze router used in the post-processing is illustrated in Figure 3.35.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

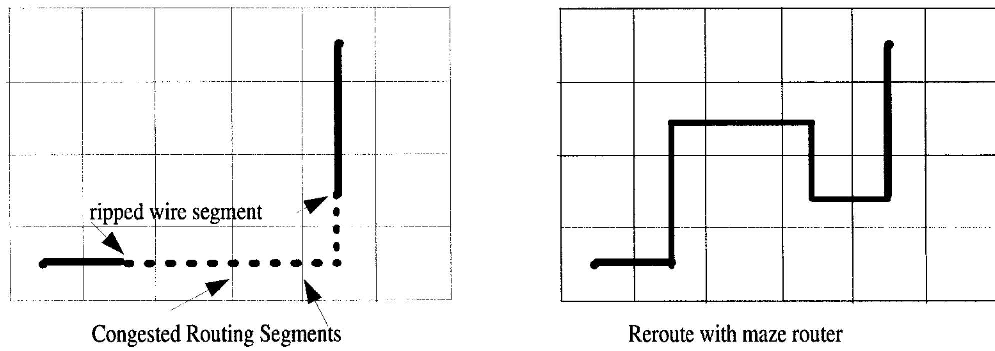


Figure 3.34: Ripping scheme for the maze router

1 $MazeRouter(from, to)$
2 $path.push(from)$
3 **while** $path \neq \phi$
4         $n = path.pop()$
5         $if(n == to)EXIT$
6   **foreach** successor s of n
7             $cost = CostFromStart + TravelCost$
8   **if** $n \in path$ as m
9   **if** $Cost(n) < Cost(m)$
10                     $path.Remove(m)$
11                     $path.push(n)$
12   **else**
13                     $path.push(n)$
14 #No Path Exists "from" to "to"

Figure 3.35: Maze Routing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.10 Conclusions

We use a complete GR algorithm to address the various issues related to performance and congestion. Most conventional methods ignore the layer assignment and buffer insertion while constructing the RT. Ignoring the impact of the layer assignment and buffer insertion as well as the inaccurate delay modeling can result in suboptimal RTs. We simultaneously applied these algorithms to obtain a better overall RT topology. Our method also uses the layer assignment to improve delay, congestion and crosstalk avoidance. A more accurate delay calculation method based on AWE is used to obtain the delay violations at sinks. This delay calculation engine is used in our performance-driven algorithms to adequately account the DSM effects.

Poor congestion optimization of hierarchical methods can result in detours, significantly increasing the delay and the use of routing resources. We dealt with this problem by a pre-router based on a simplified 2-Bend MIP routing model. This loose routing helps the network-router avoid blockages and congested areas which are not in the focus of the current hierarch level. Traditional network-flow routing models only focus on eliminating the overflows. They fail to optimize congestion and results in over-utilized and under-utilized routing areas. We use a new congestion optimization method based on slack values of both the MIP and the network-flow routing to determine which routing area can be optimized further. By doing so, congestion can be distributed homogeneously through the routing area. Our method also takes into account the blockages and existing routing for real word applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Experimental Results

We implemented our algorithm in C++. It accepts the file formats from both *cct* (a commercial router) and *Lef/Def* (OpenEDA Library Exchange Format/Design Exchange Format). A custom technology file is also used for setting the various parameters used in the algorithm. The technology file contains the following information:

- Lef/Def file names
- cct structure, library placement, net list and design file names
- timing data file
- die size
- tile size for MIP routing
- tile size for network-flow routing
- order of the AWE delay calculation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- lump, the length of the maximum segment used for delay calculation
- driver parameters (output resistance and capacitance)
- sink parameters (load capacitance)
- buffer parameters (input capacitance, output resistance and capacitance)
- selection of routing layers
- overrides of routing layer parameters (direction, pitch, unit resistance, capacitance and inductance)
- maze router parameters (direction change cost, layer change cost, step cost and wrong direction cost)
- any additional blockages
- selection of the nets from netlist

ILOG's CPLEX optimization suite [52] is used as the MIP, Network and QP solver. A Linux workstation with 2GHz Pentium 4 and 1GB memory is used as the compute platform. Both real world and randomly generated layouts are used as test cases.

The following sections present the preliminary experimental results. Section 4.1 compares the results of the simultaneous buffered RT construction with layer assignment and the traditional sequential RT construction methods. Section 4.2 presents the congestion achieved with the optimization method.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1 Simultaneous Buffered Routing-Tree Construction with Layer Assignment Results

Our algorithm was applied to construct RTs for a subset of the global nets in a 64-bit microprocessor core. The subset has a total of 13204 unconnected nets. In order to understand the behavior of the RT construction algorithm, different types of multiple-pin nets, i.e. 2-pin nets and 3-pin nets, were extracted from the unrouted nets to form different categories for comparison. There are a total of 9938 2-pin nets and 1339 3-pin nets. Since there are not enough nets with more than 3 pins in the unrouted set, 1000 randomly generated 4 to 6-pin nets were used for comparison. For the nets in the microprocessor core, the timing constraints were extracted from a static timing database. For randomly generated nets, the timing constraints were generated based on timing of the corresponding minimum length ST plus a random perturbation. The routing uses five upper metal layers.

RT constructions were performed on each set of routes. The unrouted nets of the microprocessor core as a whole were also considered as a test case. Since the CSRT method [14] is a typical sequental algorithm and is widely used for RT construction, we compare the results from the concurrent algorithm to those from CSRT. Because CSRT doesn't include buffer insertion and layer assignment, for the sake of fairness, the buffer insertion algorithm is applied to the nets with delay violation at the end. An initial layer assignment is performed based on estimation of timing criticality of the nets. This is a typical flow in a highly hierarchical custom design. The initial layer assignment is often performed manually.

The results were compared based on the following metrics:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- $\#B$: number of buffers inserted.
- $\#DV$: number of nets with delay violation.
- $maxDV$: maximum delay violation.
- L : total length of the RTs.
- CPU : CPU time.

The results in Table 4.1 show that the number of delay violations are consistently better in each test case for our algorithm. The number of buffers required for achieving better delay violations are also consistently less for our algorithm, ranging from 9% less buffers for μP core to around 29% for 6-pin nets. The RT algorithm also achieved around 1.5% shorter overall routing tree length. For 2-pin nets, our RT algorithm is reduced to the traditional method since there is only one possible connection for 2-pins nets. Our algorithm performs better overall as the pin count increases. This is due to fact that critical pins are optimized early in the routing process and less critical pins are connected to better locations on the existing RT later in the routing process.

Table 4.2 shows how the wires are distributed among layers for the μP core test case. Our algorithm yields more homegenous layer utilization. This can potentially yield better congestion optimization for the global routing. Since the actual routing has not been performed at this stage, wire lengths are used as a congestion metric instead of overflows on routing tiles.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.1: Sequential and simultaneous routing tree construction results

		#B	#DV	maxDV (ps)	L (m)	CPU (s)
9938/2-pins	CSRT	2339	277	205	22.6	27
	Our Alg.	2339	277	205	22.6	40
1339/3-pins	CSRT	1129	184	208	6.5	1.9
	Our Alg.	979	181	201	6.4	2.8
1000/4-pins	CSRT	685	63	348	16.5	2.2
	Our Alg.	490	53	214	16.5	3.8
1000/5-pins	CSRT	783	53	267	19.7	3.2
	Our Alg.	699	40	244	19.5	4.1
1000/6-pins	CSRT	1041	56	470	23.1	7.3
	Our Alg.	735	41	309	22.5	8.2
13204 nets	CSRT	3690	535	221	32.1	69
μP core	Our Alg.	3335	533	201	31.7	159

4.2 GR Results

A total of 20 test cases are randomly generated ranging from 1000 two-pin nets to 25000 five-pin nets in addition to a subset of the nets in a 64-bit microprocessor core. The timing constraints for the 64-bit microprocessor core were extracted from a static timing database. We obtained the timing requirements for the randomly generated

Table 4.2: Wire distribution

	$M3$	$M4$	$M5$	$M6$	$M7$
CSRT	12%	28%	23%	27%	10%
Our Alg.	17%	22%	24%	22%	15%

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

test cases by randomly deviating the actual delay of the minimal length ST by ±40%. Metal layers 3-7 were used for the routing except for first twelve cases where only 3 routing layers were used. The global routing is performed on each test case both by our GR algorithm and the commercial CCT router from Cadence. Detailed routing from both our algorithm and CCT is obtained using the same DR for consistency.

CCT GR is an industry leading global router for complex macro cell based designs. It employs the conventional sequential rip-up and re-route strategy. It has been widely used for global routing of large and high performance chips such as microprocessors. CCT GR does layer assignment and its primary objective is to achieve short wire length. It has a single dial for congestion control. In our experiments, the congestion control is set in the middle to balance congestion and overall wire length. CCT GR doesn't handle repeaters. Therefore, repeaters were inserted after routing is completed using the same repeater insertion algorithm used in the RT construction algorithm.

For each test case, the following metrics were obtained for comparison:

- Total wire length.
- Number of delay violations.
- Maximum delay violation.
- Number of repeaters used.

Table 4.3 lists the comparison results in all metrics collected. The first column shows the test cases ranging from 1000 2-pin nets (1k2) to 25000 5-pin nets (25k5). The last entry in the column is a subset of global routes from a 64-bit microprocessor design. For each testcase, the results from CCT GR (marked "cct") and from our

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GR algorithm (marked "gr") are listed for comparison. The meaning of the rest of the columns is as follows:

- **the column marked "#N":** the total number of nets in the best case;
- **the column marked "maxDV":** the maximum delay violation (negative timing slack) that still exist after the DR is completed;
- **the column marked "L":** the total route length;
- **the column marked "#B":** the total number of repeaters used in a given test case; and
- **the column marked "#DV":** the number of delay violation that still exist after the DR is completed.

For randomly generated test cases, our algorithm achieved, on average, 29% reduction in number of delay violations, 20% reduction in maximum delay violation, and 40% reduction in number of repeaters compared to CCT. Similarly, for the microprocessor case, the our algorithm achieved 20% reduction in number of delay violations, 8% reduction in maximum delay violation, and 18% reduction in number of repeaters compared to CCT. The wire length tends to be longer that those produced by CCT while delay and power consumption (i.e the number of repeaters used) are consistently better than those produced by CCT. This is because the our GR algorithm can spread the routes for those nets which have a lot of positive timing slack while keeping the timing critical nets short.

We also compared the congestion distribution among routing layers as illustrated in Table 4.5 and Figure 4.1. The conguestion data in Table 4.5 is calculated using the extracted nets routed through the tile boundary. Our algorithm generated more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.3: Comparison of our method with cct

Test		#N	L (m)	#DV	maxDV (ps)	#B
	cct	1000	6.564	21	54.204	254
1k2	gr	1000	6.612	15	33.023	156
	cct	1000	10.962	15	48.464	210
1k3	gr	1000	13.466	10	34.160	134
	cct	1000	14.491	7	66.170	209
1k4	gr	1000	19.063	7	57.781.	129
	cct	1000	17.771	11	14.548	157
1k5	gr	1000	24.896	6	15.070	97
	cct	5000	33.738	122	62.133	1225
5k2	gr	5000	33.745	73	41.642.	789
	cct	5000	53.903	53	47.564	1202
5k3	gr	5000	65.830	31	34.821.	711
	cct	5000	72.098	39	52.121	923
5k4	gr	5000	95.783	35	22.826.	539
	cct	5000	88.954	27	44.741	942
5k5	gr	5000	125.051	23	41.011.	553

unified congestion distribution reducing congested hot-spots. This is due to two reasons:

- Our algorithm tends to find global optimal routes with good congestion planning through the stage-I MIP routing.

- QP-based layer assignment allows further adjustment of layer assignment, taking into account routing artifacts not foreseen during RT construction.

Less congestion can often allow wire spreading to achieve less coupling noise and better

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

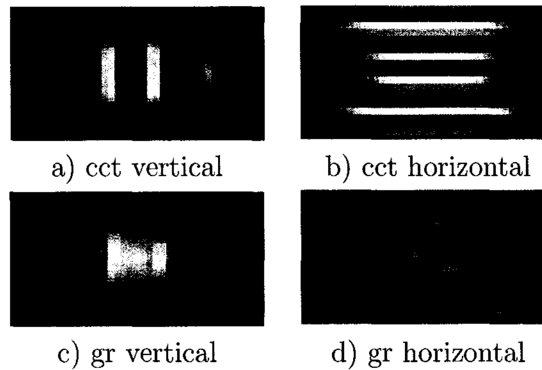


Figure 4.1: Congestion Distribution

DFM. The CPU run-time is not compared in Table 4.3 because two algorithms were running on different computing platforms. For the GR algorithm running on a 2GHz Pentium 4 computer with 1GB memory, the runtime ranges from 17 seconds for the 1k2 test case to 150 minutes for the 25k5 test case, and 17 minutes for the 64-bit microprocessor case.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.4: Comparison of our method with cct (cont.)

Test		#N	L (m)	#DV	maxDV (ps)	#B
	cct	10000	66.420	281	66.637	2448
10k2	gr	10000	66.436	169	56.459.	1544
	cct	10000	108.547	54	48.207	919
10k3	gr	10000	131.802	47	40.001.	494
	cct	10000	145.577	45	48.888	784
10k4	gr	10000	187.936	21	15.657.	405
	cct	10000	177.581	25	26.375	569
10k5	gr	10000	246.418	28	25.598.	251
	cct	25000	166.592	346	61.979	2972
25k2	gr	25000	166.582	190	52.131.	1520
	cct	25000	273.891	171	61.882	2588
25k3	gr	25000	330.370	139	59.771.	1400
	cct	24643	370.186	88	55.980	2111
25k4	gr	24701	471.958	71	42.059.	1133
	cct	23731	456.510	80	21.113	1678
25k5	gr	23884	615.289	54	27.307.	920
cpu	cct	11889	33.902	595	231.953	3733
core	gr	12144	33.617	473	212.015.	3065

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.5: Congestion (number of flows) Distribution between layers

Test		ML3	ML4	ML5	ML6	ML7
1k2	cct	11	33k	34k	N/A	N/A
	gr	19k	32k	13k	N/A	N/A
5k3	cct	48	283k	278k	N/A	N/A
	gr	178k	332k	147k	N/A	N/A
10k4	cct	2k	746k	685k	13k	64k
	gr	273k	584k	435k	358k	227k
25k5	cct	791k	1448k	1010k	865k	626k
	gr	894k	1901k	1445k	1181k	729k
cpu core	cct	2903k	2318k	2527k	2421k	919k
	gr	2900k	2305k	2532k	2435k	914k

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Concluding Remarks and Future Work

5.1 Concluding Remarks

With the ever shrinking feature sizes along the DSM generations, the performance of a VLSI chip is no longer determined by gates. Interconnect delay now dominates the gate delay. The complexity of the today's large VLSI chips also aggregate the interconnect problem. Furthermore, the interconnect optimization becoming more difficult due to the large number of parameters impacting the interconnects. GR tools must use every available resource to achieve successful routing in terms of design convergence. Three most effective methods to obtain the timing closure are the RT construction, layer assignment and buffer insertion. Most existing work ignores layer assignment, buffer insertion or both during the RT construction. This can yield sub-optimal results. We combined these algorithms to achieve the best overall result. The experiments show that the number of violations can be approximately 40%

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

better if these algorithm's are applied simultaneously compared to that with these algorithms applied separately. If one or both of the algorithms are ignored, the difference on the number of violations is much greater. This will result in overuse of routing resources further degrading the final routing quality. The new multi-objective based RT construction technique considering performance, power and congestion consistently produces shorter tree length, less number of buffers and less number of delay violations. The higher fanout of a net, the better improvement the algorithm can provide.

However, the benefit of the proposed RT construction method seems to diminishing as the number of pins increases. In some cases, excessive amount of buffers were inserted resulting in increase in power consumption. The run-time is also a concern for nets with large number of pins. This is mostly due to the fact that the layer assignment and buffer insertion is still applied to all the already constructed nets during the construction of nets with large number of pins. Hence, the long run-times can be reduced further by not applying the simultaneous scheme after majority of the nets are constructed.

Hierarchical routing schemes are used to cope with the longer runtime associated with multi-commodity routing models. Unfortunately, this method fails to avoid congested areas since it only focus on a certain part of the routing area at a time. We use a 2-bend MIP based pre-routing stage to optimize the congestion globally. This loose routing greatly improved the final routing quality as it demonstrated with the congestion histogram in Chapter 4. In general, Network-flow based routers focus on eliminating overflows. Thus, some routing areas are under-utilized as long as there is no overflow. We eliminated this problem by iteratively tightening the upper bound of the routing capacity until only those nets that can't be assigned elsewhere are left in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the routing area. This results in more homogenous distribution of the congestion with the side benefit of reducing the crosstalk. The network flow formulation is key to allow such and optimization. This is one of the key differences between the conventional multi-commodity flow formulation and our formulation.

In MIP routing, if the routing area is sparsely congested, MIP optimization engine sometimes fails to reach the optimal solution within an acceptable run-time even though the very similar problem was solved in the previous iteration in a few milliseconds. This issue will also be investigated with the help of ILOG CPLEX representative.

The results from the test cases including a subset of the routes on a commercial 64-bit microprocessor core show that our method outperforms commercial CCT router. On average, we achieved 29% less delay violations, 40% less repeater usage on the resulting routing-trees, 20% less maximum delay violation and better congestion distribution.

5.2 Future Work

The proposed algorithm can be further improved by addressing the following shortcomings:

- Handling restricted buffer locations: The algorithm assumes that buffers can be inserted anywhere in the layout. This is not a realistic assumption. In general, only certain areas are reserved for buffers called buffer farms. The buffer insertion algorithm should take the restricted buffer locations into account.

- Handling buses: The nets of a bus should be routed together, since they should

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be in the same length to obtain the same signal arrival time at each bit of the bus.

- Many pins are not in regular square shape: We assume that the connection point is in the middle of the pin. This sometimes hurt the routing especially with very long rectangular pins packed together side by side.

- Further reduction of run-time of the RT construction algorithm: As seen from the flow chart in Figure 3.17 of the RT construction algorithm, the main loop adds a sink at each iteration to the partially constructed tree of each net, and the layer assignment and the buffer insertion algorithms are then applied to all nets at each iteration. Thus, if there are high-pin-count nets, the loop does not terminate until those high-pin-count nets are completed. This dramatically increases the run-time due to the fact that layer assignment and buffer insertion algorithms are being run on already completed nets unnecessarily during that time. In fact, even one net with a very large number of pins will cause the algorithm keep iterating until this net is completed. Further run-time improvement can be achieved with a clever heuristic by stopping the simultaneous algorithm and then starting the sequential algorithm for the rest of the partially constructed high-pin-count nets.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] www.src.com. International technology roadmap for semiconductors. Technical report, Semiconductor Research Corporation, 2003.

[2] Jiang Hu; S.S Sapatnekar. A timing-constrained simultaneous global routing algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2002.

[3] L.P.P.P. Van Ginneken. Buffer placement in distributed rc-tree networks for minimal elmore delay. *Circuits and Systems, 1990., IEEE International Symposium on*, 1990.

[4] Jiang Hu; C.J. Alpert; S.T. Quay; G. Gandham. Buffer insertion with adaptive blockage avoidance. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2003.

[5] J. Lillis; Chung-Kuan Cheng; T.-T.Y. Lin. Optimal wire sizing and buffer insertion for low power and a generalized delay model. *Solid-State Circuits, IEEE Journal of*, 1996.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[6] C.C.N. Chu; D.F. Wong. A quadratic programming approach to simultaneous buffer insertion/sizing and wire sizing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1999.

[7] C.D. Cho; M. Sarrafzadeh. Four-bend top-down global routing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1998.

[8] M.C. Yildiz; P.H. Madden. Preferred direction steiner trees. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2002.

[9] Patrick H. Madden Ameya R. Agnihotri. Congestion reduction in traditional and new routing architectures. *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, 2003.

[10] M. Tang; K. Eshraghian; D. Habibi. Knowledge-based genetic algorithm for layer assignment. *Computer Science Conference, 2001. ACSC 2001. Proceedings. 24th Australasian*, Computer Science Conference, 2001. ACSC 2001. Proceedings. 24th Australasian.

[11] P. Saxena; C.L. Liu. Optimization of the maximum delay of global interconnects during layer assignment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2001.

[12] J. Griffith; G. Robins; J.S. Salowe; Tongtong Zhang. Closing the gap: near-optimal steiner trees in polynomial time. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1994.

[13] Jingyu Xu; Xianlong Hong; Tong Jing; Yici Cai. An efficient hierarchical timing-driven steiner tree algorithm for global routing. *Design Automation Conference,*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings., 2002.

[14] K.D. Boese; A.B. Kahng; B.A. McCoy; G. Robins. Near-optimal critical sink routing tree constructions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1995.

[15] N. Tsujii; K. Baba; S. Tsukiyama. An interconnect topology optimization by a tree transformation. *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000.*, 2000.

[16] Jaewon Oh; Iksoo Pyo; M. Pedram. Constructing lower and upper bounded delay routing trees using linear programming. *Design Automation Conference Proceedings 1996, 33rd*, 1996.

[17] A.B. Kahng; Bao Liu. Q-tree: a new iterative improvement approach for buffered interconnect optimization. *VLSI, 2003. Proceedings. IEEE Computer Society Annual Symposium on*, 2003.

[18] R.T. Hadsell; P.H. Madden. Improved global routing through congestion estimation. *Design Automation Conference, 2003. Proceedings*, 2003.

[19] Haiyun Bao; Xianlong Hong; Yici Cai. A new global routing algorithm independent of net ordering. *Design Automation Conference, 1999. Proceedings of the ASP-DAC '99*, 1999.

[20] S. Muddu; E. Sarto; M. Hofmann; A. Bashteen. Repeater and interconnect strategies for high-performance physical designs. *Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on*, 1998.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[21] C. Chiang; N. Shenoy. Challenges on global rrouting correlation. *Proceedings. 4th International Conference on ASIC, 2001*, 2001.

[22] Eric Nequist Lou Scheffer. Why interconnect prediction doesn't work. *Proceedings of the 2000 international workshop on System-level interconnect prediction*, 2000.

[23] Jiang Hu; S.S. Sapatnekar. Algorithms for non-hanan-based optimization for vlsi interconnect under a higher-order awe model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2000.

[24] L. Behjat; A. Vannelli; A. Kennings. Congestion based mathematical programming models for global routing. *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, 2002.

[25] A.B. Kahng; G. Robins. A new class of iterative steiner tree heuristics with good performance. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1992.

[26] C.D. Cho; S. Raje; M. Sarrafzadeh; M. Sriram; S.M. Kang. Crosstalk-minimum layer assignment. *Custom Integrated Circuits Conference, 1993., Proceedings of the IEEE 1993*, Custom Integrated Circuits Conference, 1993., Proceedings of the IEEE 1993.

[27] Henrik Esbensen. A macro-cell global router based on two genetic algorithms. *Proceedings of the conference on European design automation*, 1994.

[28] K. Su. The nominal delay and rise times of lumped delay networks. *E Circuits and Systems, IEEE Transactions on*, 1969.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[29] W.C. Elmore. Transient response of damped linear network with particular regard wideband amplifiers. *J. Applied Physics*, 1948.

[30] M. Hrkic; J. Lillis. Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost, congestion, and blockages. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2003.

[31] T. Deguchi; T. Koide; S. Wakabayashi. Timing-driven hierarchical global routing with wire-sizing and buffer-insertion for vlsi with multi-routing-layer. *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000*, 2000.

[32] Li-Da Huang; Minghorng Lai; D.F. Wong; Youxin Gao. Maze routing with buffer insertion under transition time constraints. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2003.

[33] Hai Zhou; D.F. Wong; I-Min Liu; A. Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2000.

[34] Haitian Hu; D.T. Blaauw; V. Zolotov; K. Gala; Min Zhao; R. Panda; S.S. Sapatnekar. Table look-up based compact modeling for on-chip interconnect timing and noise analysis. *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, 2003.

[35] C.L. Ratzlaff; L.T. Pillage. Rice: rapid interconnect circuit evaluation using awe. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1994.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[36] N. Gopal; D.P. Neikirk; L.T. Pillage. Evaluating rc-interconnect using moment-matching approximations. *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers, 1991 IEEE International Conference on*, 1991.

[37] L. Pillegi. Coping with rc(l) interconnect design headaches. *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers, 1995 IEEE/ACM International Conference on*, 1995.

[38] Y.I. Ismail. Efficient model order reduction via multi-node moment matching. *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, 2002.

[39] J.Lillis; C.K.Cheng; T-T.Y.Lin; C-Y.Ho. New performance driven routing techniques with explict area/delay tradeoff and simultaneous wire sizing. *33th Design Automation Conference Proceedings*, 1996.

[40] M .Borah; R.M. Owens; M.J. Irwin. An edge-based heuristic for steiner routing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1994.

[41] J. Chong. Challanges and opportunities for design innovations in nanomenter technologies. Technical report, Semiconductor Research Corporation, 1997.

[42] D.Wang; E.S.Kuh. A new general connectivity model and its applications to timing-driven steiner tree routing. *Electronics, Circuits and Systems, 1998 IEEE International Conference on*, 1998.

[43] P. Alti; B. Bhaumik. Neural global router. *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, 2000.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[44] C. Albrecht. Global routing by new approximation algorithms for multicommodity flow. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2001.

[45] Harald Räcke Marcin Bienkowski, Miroslaw Korzeniowski. A practical algorithm for constructing oblivious routing schemes. *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, 2003.

[46] R.C. Carden; Li Jianmin; Chung-Kuan Cheng. A global router with a theoretical bound on the optimal solution. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1996.

[47] C.K. Koh; P.H. Madden. Manhattan or non-manhattan? a study of alternative vlsi routing architectures. *Great Lakes Symposium on VLSI*, 2000.

[48] N. Garg; J.Konemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *Foundations of Computer Science, 1998. Proceedings.39th Annual Symposium on*, 1998.

[49] Lisa K. Fleischer; Kevin D. Wayne. Fast and simple approximation schemes for generalized flow. *online*, 2002.

[50] L.K. Fleischer. Approximationg fractional multicommodity flow independent of the number of commodities. *40th Annu. Symp. Foundations of Computer Science*, 1999.

[51] P. Saxena; B. Halpin. Modeling repeaters explicity within analytical placement. *Design Automation Conference, 2003. Proceedings*, 2004.

[52] Ilog. Ilog cplex user manual. *book*, 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[53] Ilog cplex optimizer. www.ilog.com.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

Abbreviations and Acronyms

CMOS complementary metal oxide silicon

VLSI very large scale integration

DSM deep sub-micron

AWE asymptotic waveform evaluation

MIP mixed integer problem

QP quadratic programming

LP linear programming

GR global routing or global router

DR detailed routing or detailed router

RT routing-tree

ST Steiner-tree

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SMM single-point moment matching

MMM multi-point moment matching

RICE rapid interconnect circuit evaluator

NIG network interference graph

BINO buffer insertion non-Hanan optimization

CS critical sink

MRST minimum length rectalinear Steiner-tree

CSRT critical sink routing-tree

SART steiner AWE routing-tree

BPG buffer planning graph

GA genetic algorithm

DV delay violation

RAT required arrival time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B

Technology File Format

```
#lef/def I/O
read lef er.lef
write lef test.lef
read def strapped.def
read def routed.def
write def test.def

#cct I/O
read str test-1000-2.str
read lib test-1000-2.lib
read pla test-1000-2.pla
read net test-1000-2.net
read dsn test-1000-2.dsn
read timing test-1000-2.tim
read tim test-1000-2.tim
read rte test-1000-2.cct.rte

#various paramaters
set diearea 0 -12834 13898 0
set gcell 100 100
set gtile 10 10
set order 1
set lump 100
set driver 40 0
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
set load 0.1
set buffer 0.1 40 0

#report congestion on existing routing
report cctcong test-1000-2.cct.cong.rpt

#layer selection
unselect layer METAL1
unselect layer METAL2
select layer METAL3
select layer METAL4
select layer METAL5
select layer METAL6
select layer METAL7

#layer paramaters
set layer METAL3 res 0.5625
set layer METAL4 res 0.38387
set layer METAL5 res 0.236752
set layer METAL6 res 0.1064
set layer METAL7 res 0.041334
set layer METAL3 cap 0.00024
set layer METAL4 cap 0.000249482
set layer METAL5 cap 0.00024998
set layer METAL6 cap 0.00026654
set layer METAL7 cap 0.000363712


#maze router settings
set maze dir_change_cost 1
set maze layer_change_cost 1
set maze step_cost 3
set maze wrong_dir_cost 4

#net selection
unselect net shift
unselect net nshift
unselect net VDD
unselect net GND
unselect net core_clk
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#misc. opertaions for reporting
include test-1000-2.blk
init
gen_rnd_tim
write timing test-1000-2.tim
report congcct test-1000-2.gr.cong.rpt
report ccttim test-1000-2.cct.tim.rpt

#global routing
groute

#output files
write test-1000-2.blk
write rte test-1000-2.gr.rte
write grte test.grte
write flw test.flw

exit
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C

Source Code

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
#ifndef CBUFFER_H
#define CBUFFER_H

#include "global.h"
#include "cnode.h"

class CBuffer : public CNode
{
protected:
double m_load;
public:
double load() const { return m_load; };
    void load(const double load) {m_load=load;};
virtual double cap() const { return m_load + m_cap; };
public:
CBuffer(const CPoint point,const double load);
virtual ~CBuffer();
};

#endif
#ifndef CCCTCOMP_H
#define CCCTCOMP_H

#include "global.h"

class CCctComp
{
private:
QString m_name;
QString m_image;
CPoint m_placement;
public://get
QString name() const { return m_name; };
QString image() const { return m_image; };
CPoint placement() const { return m_placement; };
public://set
void place(const double x, const double y) { m_placement = CPoint(x,y);
};
public:
CCctComp(const QString name, const QString image);
virtual ~CCctComp();
};

#endif
#ifndef CCCTIMAGE_H
#define CCCTIMAGE_H

#include "global.h"

class CCctPin;
class CCctKeepout;

class CCctImage
{
private:
QString m_name;
CPoly m_poly;bool m_hasPoly;
CRect m_rect; bool m_hasRect;
QDict<CCctPin> m_pins;
QPtrList<CCctKeepout> m_keepouts;
public://get
QString name() const { return m_name; };
CCctPin* pin(const QString name) const { return m_pins[name]; };
QDict<CCctPin> pins() const { return m_pins; };
QPtrList<CCctKeepout> keepouts() const { return m_keepouts; };
public://set
void setPoly(const CPoly poly) { m_hasPoly=true; m_poly = poly; };
void setRect(const CRect rect) { m_hasRect=true; m_rect = rect; };
void addPin(CCctPin* pin);
void addKeepout(CCctKeepout* keepout) { m_keepouts.append(keepout); };
public:
CCctImage(const QString name);
virtual ~CCctImage();
};

#endif
#ifndef CCCTKEEPOUT_H
#define CCCTKEEPOUT_H

#include "global.h"

class CCctKeepout
{
private:
QString m_layer;
CRect m_rect;
public:
QString layer() const { return m_layer;};
CRect rect() const { return m_rect; };
public:
CCctKeepout(const QString layer, const CRect rect);
virtual ~CCctKeepout();
};

#endif
#ifndef CCCTLAYER_H
#define CCCTLAYER_H

#include "global.h"

class CCctLayer
{
private:
QString m_name;
layerType m_type;
layerDirectionType m_dir;bool m_hasDirection;
double m_width;bool m_hasWidth;
double m_clear;bool m_hasClear;
int m_index;
public://get
bool hasDirection() const { return m_hasDirection; };
bool hasWidth() const { return m_hasWidth; };
bool hasPitch() const { return m_hasWidth && m_hasClear; };
public://get
layerDirectionType dir() const { return m_dir; };
double pitch() const { return m_width + m_clear; };
double width() const { return m_width; };
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
int index() const { return m_index; };
public://set
void setType(const layerType type) { m_type=type;};
void setDirection(const layerDirectionType dir) { m_dir =
dir;m_hasDirection=true;};
void setWidth(const double width) { m_width = width;m_hasWidth=true;};
void setClear(const double clear) { m_clear= clear;m_hasClear=true;};
public://get
QString name() const { return m_name; };
public:
CCctLayer(const QString name, const int index);
~CCctLayer();
};

#endif
#ifndef CCCTNET_H
#define CCCTNET_H

#include "global.h"

class CCctWire;

class CCctNet
{
private:
QString m_name;
QStringList m_pins;
QPtrList<CCctWire> m_wires;
public://get
QString name() const { return m_name; };
QStringList pins() const { return m_pins; };
QPtrList<CCctWire> wires() const { return m_wires; };
public://set
void addPin(const QString pin) { m_pins.append(pin); };
void addWire(CCctWire* wire) { m_wires.append(wire); };
public:
CCctNet(const QString name);
virtual ~CCctNet();
};

#endif
#ifndef CCCTPATH_H
#define CCCTPATH_H

#include "global.h"

class CCctPath
{
private:
QString m_layer;
int m_width;
int m_x1,m_y1,m_x2,m_y2;
QString  m_pin1;
QString m_pin2;
public:
bool intersects(CCctPath p);
static QPtrList<CCctPath> nonInPaths(const CCctPath p1, const CCctPath
p2);
QString layer() const { return m_layer; };
        int Width() const { return m_width; };
void pin1(const QString pin1) { m_pin1 = pin1; };
void pin2(const QString pin2) { m_pin2 = pin2; };
QString pin1() const { return m_pin1; };
QString pin2() const { return m_pin2; };
int x1() const { return m_x1; };
int y1() const { return m_y1; };
int x2() const { return m_x2; };
int y2() const { return m_y2; };
int width() const { return m_x2-m_x1; };
int height() const { return m_y2-m_y1; };
int left() const { return ::min(m_x1,m_x2); };
int right() const { return ::max(m_x1,m_x2); };
int bottom() const { return ::min(m_y1,m_y2); };
int top() const { return ::max(m_y1,m_y2); };
public:
CCctPath(const QString layer, const int width, const int x1, const int
y1,
const int x1, const int y2);
virtual ~CCctPath();
};
#endif
#ifndef CCCTPIN_H
#define CCCTPIN_H


#include "global.h"

class CCctPin
{
private:
QString m_name;
QValueList< QPair<QString, CRect> > m_ports;
public:
QString name() const { return m_name; };
QValueList< QPair<QString, CRect> > ports() const { return m_ports; };
public:
void addPort(const QString layer, const CRect rect)
{ m_ports.append( QPair<QString, CRect>::QPair(layer,rect) ); };
public:
CCctPin(const QString name);
virtual ~CCctPin();
};

#endif
#ifndef CCCTWIRE_H
#define CCCTWIRE_H

#include "global.h"

class CCctPath;

class CCctWire
{
private:
QString m_pin1;
QString m_pin2;
QPtrList<CCctPath> m_paths;
public:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void addPath(CCctPath* path) { m_paths.append(path); };
void pins(const QString pin1,const QString pin2);
public:
QPtrList<CCctPath> paths() const { return m_paths; };
public:
CCctWire();
virtual ~CCctWire();
};

#endif
#ifndef COMPLEX_H
#define COMPLEX_H

class complex
{
private:
  double realp;
  double imagp;
public:
  const complex & operator = (const complex & value);
  const complex & operator += (const complex & value);
  double get_real() const { return realp; }
  double get_imag() const { return imagp; }
  complex get_conj() const { return complex(realp, -imagp); }
private:
  friend complex operator * (const complex &, const complex &);
  friend complex operator / (const complex &, const complex &);
  friend complex operator + (const complex & A, const complex & B)
    { return complex (A.realp + B.realp, A.imagp + B.imagp); }
  friend complex operator - (const complex & A, const complex & B)
    { return complex (A.realp - B.realp, A.imagp - B.imagp); }
  friend complex operator - (const complex & A)
    { return complex (-A.realp, -A.imagp); }
  friend complex exp(const complex &);
  friend complex pow(const complex &, int);
  friend double fabs(const complex & A)
    { return sqrt(A.realp*A.realp + A.imagp*A.imagp); }
  friend double arg(const complex &);
  friend complex sqrt(const complex &);
  friend complex cbrt(const complex &);
public:
  complex(double rp = 0.0, double ip = 0.0): realp(rp), imagp(ip) {}
  complex(const complex & c): realp(c.realp), imagp(c.imagp) {}
  virtual ~complex() {}
};
#endif
#ifndef CDEF_H
#define CDEF_H

#include "global.h"
#include <defrReader.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefgcell.h"

class CDefComp;
class CDefPin;
class CDefBlock;
class CDefNet;
class CDefPath;

//def interface
class CDef
{
private:
QString m_versionStr;double m_version;bool m_hasVersion;
bool m_caseSensitive; bool m_hasCaseSensitive;
  QString m_divider;bool m_hasDivider;
QString m_busBit;bool m_hasBusBit;
QString m_designName;bool m_hasDesign;
    CRect m_dieArea;bool m_hasDieArea;
double m_units; bool m_hasUnits;
CDefGCell m_gCell;bool m_hasGCell;
QDict<CDefComp> m_comps;
QDict<CDefNet> m_nets;
QDict<CDefNet> m_snets;
QDict<CDefPin> m_pins;
QPtrList<CDefBlock> m_blocks;
public: //set
void busBit(const QString busBit) { m_hasBusBit=true;m_busBit=busBit; };
void design(const QString design) {
m_hasDesign=true;m_designName=design; };
void divider(const QString divider) {
m_hasDivider=true;m_divider=divider; };
void version(const double version) {
m_hasVersion=true;m_version=version; };
void versionStr(const QString version) {
m_hasVersion=true;m_versionStr=version; };
void dieArea(const double xl, const double yl, const double xh, const
double yh)
{m_hasDieArea=true;m_dieArea=CRect(xl,yl,xh-xl,yh-yl);
};
void caseSensitive(const bool caseSensitive)
{ m_hasCaseSensitive=true;m_caseSensitive=caseSensitive; };
void units(const double units) { m_hasUnits=true;m_units=units; };
void gCell(const CDefGCell gCell) { m_hasGCell=true;m_gCell=gCell; };
// void addBlockage(CDefBlockage*
pBlockage){m_blockages.append(pBlockage);};
public: //get
bool hasBusBit() const { return m_hasBusBit; };
bool hasDesign() const { return m_hasDesign; };
bool hasDivider() const { return m_hasDivider; };
bool hasVersion() const { return m_hasVersion; };
bool hasDieArea() const { return m_hasDieArea; };
bool hasCaseSensitive() const { return m_hasCaseSensitive; };
bool hasUnits() const { return m_hasUnits; };
bool hasGCell() const { return m_hasGCell; };
public: //reader callbacks
friend int defrBlockageStartCbk
( defrCallbackType_e type, int numBlockages, defiUserData data
);
friend int defrBlockageCbk
( defrCallbackType_e type, defiBlockage* blockage, defiUserData
data );
friend int defrBlockageEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrBusBitCbk
( defrCallbackType_e type,const char* busBit, defiUserData data
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
);
friend int defrComponentStartCbk
( defrCallbackType_e type,int numComps, defiUserData data );
friend int defrComponentCbk
( defrCallbackType_e type, defiComponent* comp, defiUserData
data );
friend int defrComponentEndCbk
( defrCallbackType_e type,void* ptr, defiUserData data );
friend int defrPathCbk
( defrCallbackType_e type, defiPath* path, defiUserData data );
friend int defrDesignStartCbk
( defrCallbackType_e type, const char* design, defiUserData data
);
friend int defrDesignEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrDieAreaCbk
( defrCallbackType_e type, defiBox* box, defiUserData data );
friend int defrDividerCbk
( defrCallbackType_e type, const char* divider, defiUserData
data );
friend int defrComponentExtCbk
( defrCallbackType_e type,  const char* compExt, defiUserData
data );
friend int defrGroupExtCbk
( defrCallbackType_e type, const char* groupExt, defiUserData
data );
friend int defrNetExtCbk
( defrCallbackType_e type, const char* netExt, defiUserData data
);
friend int defrNetConnectionExtCbk
( defrCallbackType_e type, const char* netConnExt, defiUserData
data );
friend int defrPinExtCbk
( defrCallbackType_e type, const char* pinExt, defiUserData
datak );
friend int defrScanChainExtCbk
( defrCallbackType_e type, const char* scanChain, defiUserData
data );
friend int defrViaExtCbk
( defrCallbackType_e type, const char* viaExt, defiUserData data
);
friend int defrFillStartCbk
( defrCallbackType_e type, int numFills, defiUserData data );
friend int defrFillCbk
( defrCallbackType_e type, defiFill* fill, defiUserData data );
friend int defrFillEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrGcellGridCbk
( defrCallbackType_e type, defiGcellGrid* grid, defiUserData
data );
friend int defrGroupsStartCbk
( defrCallbackType_e type, int numGroups, defiUserData data );
friend int defrGroupNameCbk
( defrCallbackType_e type, const char* group, defiUserData data
);
friend int defrGroupMemberCbk
( defrCallbackType_e type, const char* groupMember, defiUserData
data );
friend int defrGroupCbk
( defrCallbackType_e type, defiGroup* group, defiUserData data
);
friend int defrGroupsEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrHistoryCbk
( defrCallbackType_e type, const char* history, defiUserData
data );
friend int defrCaseSensitiveCbk
( defrCallbackType_e type, int caseSensitive, defiUserData data
);
friend int defrNetStartCbk
( defrCallbackType_e type, int numNets, defiUserData data );
friend int defrNetCbk
( defrCallbackType_e type, defiNet* net, defiUserData data );
friend int defrNetEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrStartPinsCbk
( defrCallbackType_e type,int numPins, defiUserData data );
friend int defrPinCbk
( defrCallbackType_e type, defiPin* pin, defiUserData data );
  friend int defrPinEndCbk
( defrCallbackType_e type, void* ptr,defiUserData data );
friend int defrPinPropStartCbk
( defrCallbackType_e type, int numPinProps, defiUserData data );
friend int defrPinPropCbk
( defrCallbackType_e type, defiPinProp* pinProp, defiUserData
data );
friend int defrPinPropEndCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrPropDefStartCbk
( defrCallbackType_e type, void* ptr, defiUserData data );
friend int defrPropCbk
( defrCallbackType_e type,defiProp* prop, defiUserData data );
friend int defrPropDefEndCbk
( defrCallbackType_e type,void* ptr,defiUserData data );
friend int defrRegionStartCbk
( defrCallbackType_e type, int numRegions, defiUserData data );
friend int defrRegionCbk
( defrCallbackType_e type, defiRegion* region, defiUserData data
);
friend int defrRegionEndCbk
( defrCallbackType_e type,void* ptr, defiUserData data );
friend int defrRowCbk
( defrCallbackType_e type, defiRow* row, defiUserData data );
friend int defrScanchainsStartCbk
( defrCallbackType_e type, int numScanChains, defiUserData data
);
friend int defrScanchainCbk
( defrCallbackType_e type, defiScanchain* scanChain,
defiUserData data );
friend int defrScanchainsEndCbk
( defrCallbackType_e type,void* ptr, defiUserData data);
friend int defrSlotStartCbk
( defrCallbackType_e type, int numSlots, defiUserData data );
friend int defrSlotCbk
( defrCallbackType_e type, defiSlot* slot, defiUserData data );
friend int defrSlotEndCbk
( defrCallbackType_e type, void*  ptr, defiUserData data );
friend int defrSNetStartCbk
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
( defrCallbackType_e type,int numSNets,defiUserData data );
friend int defrSNetCbk
( defrCallbackType_e type, defiNet* sNet, defiUserData data );
friend int defrSNetEndCbk
( defrCallbackType_e type,void* ptr, defiUserData data );
friend int defrTechNameCbk
( defrCallbackType_e type,const char* tech, defiUserData data );
friend int defrTrackCbk
( defrCallbackType_e type, defiTrack* track, defiUserData data
);
friend int defrUnitsCbk
( defrCallbackType_e type,double units, defiUserData data );
friend int defrVersionCbk
( defrCallbackType_e type,double version, defiUserData data );
friend int defrVersionStrCbk
( defrCallbackType_e type,const char* version, defiUserData data
);
friend int defrViaStartCbk
( defrCallbackType_e type,int numVias, defiUserData data );
friend int defrViaCbk
( defrCallbackType_e type,defiVia* via, defiUserData data );
friend int defrViaEndCbk
( defrCallbackType_e type,void* ptr, defiUserData data );

public://writer callbacks
friend int defwBlockageCbk(defwCallbackType_e type, defiUserData data );
friend int defwBusBitCbk(defwCallbackType_e type, defiUserData data );
friend int defwCaseSensitiveCbk(defwCallbackType_e type, defiUserData
data );
friend int defwComponentCbk(defwCallbackType_e type, defiUserData data
);
friend int defwDesignCbk(defwCallbackType_e type, defiUserData data );
friend int defwDesignEndCbk(defwCallbackType_e type, defiUserData data
);
friend int defwDieAreaCbk(defwCallbackType_e type, defiUserData data );
friend int defwDividerCbk(defwCallbackType_e type, defiUserData data );
friend int defwExtCbk(defwCallbackType_e type, defiUserData data );
friend int defwGcellGridCbk(defwCallbackType_e type, defiUserData data
);
friend int defwGroupCbk(defwCallbackType_e type, defiUserData data );
friend int defwHistoryCbk(defwCallbackType_e type, defiUserData data );
friend int defwNetCbk(defwCallbackType_e type, defiUserData data );
friend int defwPinCbk(defwCallbackType_e type, defiUserData data );
friend int defwPinPropCbk(defwCallbackType_e type, defiUserData data );
friend int defwPropDefCbk(defwCallbackType_e type, defiUserData data );
friend int defwRegionCbk(defwCallbackType_e type, defiUserData data );
friend int defwRowCbk(defwCallbackType_e type, defiUserData data );
friend int defwSNetCbk(defwCallbackType_e type, defiUserData data );
friend int defwScanchainCbk(defwCallbackType_e type, defiUserData data
);
friend int defwTechCbk(defwCallbackType_e type, defiUserData data );
friend int defwTrackCbk(defwCallbackType_e type, defiUserData data );
friend int defwUnitsCbk(defwCallbackType_e type, defiUserData data );
friend int defwVersionCbk(defwCallbackType_e type, defiUserData data );
friend int defwViaCbk(defwCallbackType_e type, defiUserData data );
private://helpers
QPtrList<CDefPath> m_paths;

public://get
CRect dieArea() const { return m_dieArea; };
double units() const { if(m_hasUnits) return m_units; else return
1000;};
CDefGCell gCell() const { return m_gCell; };
QPtrList<CDefBlock> blocks() const { return m_blocks; };
QDict<CDefPin> pins() const { return m_pins; };
QDict<CDefComp> comps() const { return m_comps; };
QDict<CDefNet> nets() const { return m_nets; };
QDict<CDefNet> snets() const { return m_snets; };
CDefComp* comp(const QString name) const { return m_comps[name]; };
CDefNet* net(const QString name) const { return m_nets[name]; };
CDefNet* snet(const QString name) const { return m_snets[name]; };
CDefPin* pin(const QString name) const { return m_pins[name]; };
public: //io
bool load(const QString fileName);
bool save(const QString fileName);
public:
CDef();
virtual ~CDef();
};
#endif
#ifndef CDEFBLOCK_H
#define CDEFBLOCK_H

#include "global.h"
class defiBlockage;

//def blockage
class CDefBlock
{
private:
QString m_layerName;bool m_hasLayer;
QString m_layerComponentName;
QString m_placementComponentName;
QValueList<CRect> m_rects;
bool m_hasPlacement;
bool m_hasComponent;
bool m_hasSlots;
bool m_hasFills;
bool m_hasPushdown;
public://get
bool hasLayer() const { return m_hasLayer; };
bool hasPlacement() const { return m_hasPlacement; };
bool hasComponent() const { return m_hasComponent; };
bool hasSlots() const { return m_hasSlots; };
bool hasFills() const { return m_hasFills; };
bool hasPushdown() const { return m_hasPushdown; };
int numRectangles() const { return m_rects.count(); };
QValueList<CRect> rectangles() const { return m_rects; };
QString layerName() const { return m_layerName; };
QString layerComponentName() const { return m_layerComponentName; };
QString placementComponentName() const { return
m_placementComponentName; };
public://set
void layer(const QString layer) { m_layerName=layer;m_hasLayer=true;};
void addRect(const CRect r) { m_rects.append( r ); };
void setFills(const bool fills) { m_hasFills=fills; };
public: //io
void defr(defiBlockage* blockage);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void defw();
public:
CDefBlock();
CDefBlock(defiBlockage* blockage);
virtual ~CDefBlock();
};

#endif
#ifndef CDEFCOMP_H
#define CDEFCOMP_H

#include "global.h"

class defiComponent;

//component
class CDefComp
{
private:
QString m_id;
QString m_name;
CPoint m_placement;
placementStatusType m_placementStatus;
placementOrientType m_placementOrient;
public: //set
void IdAndName(const QString id, const QString name) {m_id = id;
m_name=name;};
void setPlacementStatus(const placementStatusType placementStatus )
{m_placementStatus=placementStatus;};
void setPlacementLocation(const double placementX,
const double placementY,
const placementOrientType placementOrient)
{m_placement.x(placementX);
m_placement.y(placementY);m_placementOrient=placementOrient;};
public://get
QString id() const { return m_id; };
QString name() const { return m_name; };
placementStatusType placementStatus() const { return m_placementStatus;
};
placementOrientType placementOrient() const { return m_placementOrient;
};
bool isUnplaced() const { return m_placementStatus==UNPLACED; };
bool isPlaced() const { return m_placementStatus==PLACED; };
bool isFixed() const { return m_placementStatus==FIXED; };
bool isCover() const { return m_placementStatus==COVER; };
double placementX() const { return m_placement.x(); };
double placementY() const { return m_placement.y(); };
public://io
void defr(defiComponent* comp);
void defw() const;
public:
CDefComp(const QString id, const QString name);
CDefComp(defiComponent* comp);
virtual ~CDefComp();
};

#endif
#ifndef CDEFGCELL_H
#define CDEFGCELL_H

#include "global.h"

class defiGcellGrid;

class CDefGCell
{
private:
int m_x;
int m_numCols;
double m_xStep;
int m_y;
int m_numRows;
double m_yStep;
public://get
double x() const { return m_x; };
double y() const { return m_y; };
double xStep() const { return m_xStep; };
double yStep() const { return m_yStep; };
int numCols() const { return m_numCols; };
int numRows() const { return m_numRows; };
public://io
void defr(defiGcellGrid* defiGcellGrid);
void defw();
public:
CDefGCell();
virtual ~CDefGCell();
};

#endif
#ifndef CDEFNET_H
#define CDEFNET_H

#include "global.h"
class defiNet;
class CNetPin;
class CDefPath;
class CDefWire;

//net
class CDefNet
{
private:
QString m_name;
QString m_netStat;
useType m_use;bool m_hasUse;
QDict<CNetPin> m_pins;
QPtrVector<CDefPath> m_paths;
QPtrVector<CDefWire> m_wires;
public://set
void setName(const QString name) { m_name=name; };
void setUse(const useType use) { m_hasUse=true; m_use=use; };
void addPin(const QString instance, const QString name);
void addPath(CDefPath* path);
public://get
QString name() const { return m_name; };
useType use() const { return m_use; };
int numPins() const { return m_pins.count(); };
QDict<CNetPin> pins() const { return m_pins; };
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
CNetPin* pin(const QString pin) { return m_pins[pin]; };
int numPaths() const { return m_paths.size(); };
CDefPath* path(const int i) const { return m_paths[i]; };
int numWires() const { return m_wires.size(); };
CDefWire* wire(const int i) const { return m_wires[i]; };
bool isRouted() const { return m_netStat=="ROUTED"; };
bool isFixed() const { return m_netStat=="FIXED"; };
bool isCover() const { return m_netStat=="COVER"; };
bool hasUse() const { return m_hasUse; };
public: //io
void defr(defiNet* net);
void defw() const;
public:
CDefNet(const QString name);
CDefNet(defiNet* net, QPtrList<CDefPath> paths);
virtual ~CDefNet();
};
#endif
#ifndef CDEFPATH_H
#define CDEFPATH_H

#include "global.h"
class defiPath;

//wiring of a net
class CDefPath
{
private:
QValueList< QPair<pathType, void*> > m_path;
public:
QValueList< QPair<pathType, void*> > items() const { return m_path; };
public://io
void defr(defiPath* path);
void defw() const;
public:
CDefPath(defiPath* path);
virtual ~CDefPath();
};

#endif
#ifndef CDEFPIN_H
#define CDEFPIN_H


#include "global.h"
class defiPin;

//pin
class CDefPin
{
private:
QString m_pinName;
QString m_netName;
QString m_layer;bool m_hasLayer;
CRect m_bound;
CPoint m_placement;bool m_hasPlacement;
useType m_use; bool m_hasUse;
pinDirectionType m_direction;bool m_hasDirection;
placementStatusType m_placementStatus;
placementOrientType m_orient;
public://set
void Setup(const QString pinName, const QString netName)
{m_pinName = pinName; m_netName=netName; };
void setDirection(const pinDirectionType dir)
{m_hasDirection=true;m_direction=dir; };
void setUse(const useType use) {m_hasUse=true;m_use=use;};
void setLayer(const QString layer,const double xl, const double yl,
const double xh, const double yh)
{m_hasLayer=true;m_layer=layer; m_bound =
CRect(xl,yl,xh-xl,yh-yl);};
void setPlacement(const placementStatusType placementStatus,
const double x, const double y, const
placementOrientType orient)
{m_hasPlacement=true;m_placementStatus=placementStatus;
m_placement.x(x);m_placement.y(y);m_orient=orient;};
public://get
QString pinName() const { return m_pinName; };
QString netName() const { return m_netName; };
QString layer() const { return m_layer; }
double placementX() const { return m_placement.x(); };
double placementY() const { return m_placement.y(); };
  useType use() const { return m_use; };
pinDirectionType direction() const { return m_direction; };
placementOrientType orient() const { return m_orient; };
void bounds(double* xl, double* yl, double* xh, double* yh) const
{*xl
=m_bound.left();*yl=m_bound.bottom();*xh=m_bound.right();*yh=m_bound.top();};
public://get
bool hasDirection() const { return m_hasDirection; };
bool hasUse() const { return m_hasUse; };
bool hasPlacement() const { return m_hasPlacement; };
bool isUnplaced() const { return m_placementStatus == UNPLACED; };
bool isPlaced() const { return m_placementStatus == PLACED; };
bool isCover() const { return m_placementStatus == COVER;};
bool isFixed() const { return m_placementStatus == FIXED; };
bool hasLayer() const { return m_hasLayer; };
public://io
void defr(defiPin* pin);
void defw() const;
public:
CDefPin(defiPin* pin);
virtual ~CDefPin();
};
#endif
#ifndef CDEFWIRE_H
#define CDEFWIRE_H

#include "global.h"
class CDefPath;
class defiWire;

//wiring of a net
class CDefWire
{
private:
wireType m_wireType;
QPtrVector< CDefPath > m_paths;
public://io
```

```
void defr(defiWire* wire);
void defw() const;
public:
CDefWire(defiWire* wire);
virtual ~CDefWire();
};

#endif
#ifndef CDSN_H
#define CDSN_H

#include "global.h"

class CDsn
{
private:
QValueList< QPair<QString,CRect> > m_wiring;
public://get
QValueList< QPair<QString,CRect> > wiring() const { return m_wiring; };
public:
bool load(const QString fileName);
bool save(const QString fileName);
public:
CDsn();
virtual ~CDsn();
};

#endif
#ifndef CEDGE_H
#define CEDGE_H

#include <qvaluevector.h>
#include "clayerpair.h"

class CNode;

class CEdge
{
private:
CNode* m_up;
CNode* m_down;
CLayerPair  m_layer;
double m_res;
QValueVector<double> m_current;
public:
void layer(const CLayerPair layer) { m_layer = layer; };
public:
CNode* up() const { return m_up; };
CNode* down() const { return m_down; };
CLayerPair layer() const { return m_layer; };
double res() const { return m_res; };
double manhDist() const;
double xDist() const;
double yDist() const;
double current(const int i) const { return m_current[i]; };
double induc() const { return 0.0; };
public:
void disconnect() const;
void connect() const;
void computeR();
void resizeCurrent(const int size) { m_current.resize(size); };
void current(const double cur, const int i) { m_current[i]=cur; };
void increCurrent(const double cur, const int i) { m_current[i] +=
cur;};
public:
CEdge(CNode* up, CNode* down, const CLayerPair layer);
CEdge();
CEdge(const CEdge& e);
virtual ~CEdge();
};

#endif
#ifndef CGROUTE_H
#define CGROUTE_H

#include "global.h"
#include <ilcplex/cplex.h>
#include "cpathcut.h"
class CLayer;
class CNet;
class CNode;
class gwire;
class gpath;

class CGRoute
{
private:
bool m_initd;
CRect m_dieArea;bool m_hasDieArea;
CSize m_gCell;bool m_hasGCell;
QSize m_gTile;bool m_hasGTile;
int m_order;bool m_hasOrder;
double m_lump;bool m_hasLump;
double m_driverRes;bool m_hasDriver;
double m_driverCap;
double m_bufferLoad;bool m_hasBuffer;
double m_bufferRes;
double m_bufferCap;
double m_load;bool m_hasLoad;
QPtrVector<CLayer> m_layers;
QValueList< QPair<QString, QString> > m_cellBlocks;bool m_hasBlocks;
QValueList< QPair<QString, QString> > m_tileBlocks;
QStringList m_unselectNets;
QDict<CNet> m_nets;

//maze router
int m_layerChangeCost;
int m_dirChangeCost;
int m_stepCost;
int m_wrongDirCost;

public://set
void dieArea(const double x1, const double y1, const double x2, const
double y2)
{ m_dieArea=CRect(x1,y1,x2-x1,y2-y1);m_hasDieArea = true;};
void gCell(const double width, const double height)
{ m_gCell.width(width);m_gCell.height(height);m_hasGCell=
true;};
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void gTile(const int width, const int height)
 { m_gTile.setWidth(width);m_gTile.setHeight(height);m_hasGTile=
true;};
void order(const int order) { m_order=order;m_hasOrder=order; };
void lump(const double lump) { m_lump=lump;m_hasLump=true;};
void driver(const double res,const double cap)
{ m_driverRes=res;m_driverCap=cap;m_hasDriver=true;};
void buffer(const double load,const double res,const double cap)

{m_bufferLoad=load;m_bufferRes=res;m_bufferCap=cap;m_hasBuffer=true;};
void load(const double load) {m_load=load;m_hasLoad=true;};
bool selectLayer(const QString layer);
bool unselectLayer(const QString layer);
void addCellBlock(const QString layer, const QString blk)
{m_cellBlocks.append(QPair<QString,QString>::QPair(layer,blk));
m_hasBlocks = true;};
void addTileBlock(const QString layer, const QString blk)
{m_tileBlocks.append(QPair<QString,QString>::QPair(layer,blk));
m_hasBlocks = true;};
void unselectNet(const QString net) { m_unselectNets.append(net); };
//maze
void dirChangeCost(const int cost) { m_dirChangeCost=cost;};
void layerChangeCost(const int cost) { m_layerChangeCost = cost;};
void stepCost(const int cost) { m_stepCost = cost; };
void wrongDirCost(const int cost) { m_wrongDirCost = cost; };
public://get
CSize gCell() const { return m_gCell; };
CRect dieArea() const { return m_dieArea; };
CLayer* layer(const QString name) const;
int layerInd(const QString name) const;
int numCellOverflows() const;
int numTileOverflows() const;
int numPadViolations() const;
QPoint gpoint(const CPoint p) const;
QRect grect(const CPoint p) const;
QRect grect(const QPoint p) const;
QRect grect(const int x, const int y) const;
QPoint tpoint(const CPoint p) const;
QPtrList<gwire> wires() const;
QDict<CNet> nets() { return m_nets; };
public:
bool init();
bool initFormLefDef();
bool initFromCct();
bool gen_rnd_tim();
bool route();
void slabi();
void randtim();
void bubbleSort(QValueList<CNet*>& nets);
bool iterate(CNet* net, bool mode);
bool insertBuffer(QValueList<CNet*> bubbleList);
bool insertBuffer(CNet* net, bool moder=false);
     void removeBuffers(QValueList<CNet*> bubbleList);
void removeBuffers();
double awe(CNet* net, bool log=false);
double awe(CNode* node, bool log);
double solveDelay(double momone, complex resid[MAX_ORDER+1],
complex pole[MAX_ORDER+1], int ord);
public://routing stuff
void init(CNet* net);
void go();
double* gomain(QPtrList<gwire>& wires,double* fixes, QValueList<
QPair<int,double> >* posFixes);
void CPXbuildgo(CPXENVptr& env, CPXLPptr& lp,
QPtrList<gwire> wires, double* fixes) const;
double* CPXoptgo(CPXENVptr& env, CPXLPptr& lp) const;
int ripupwires(double* pi, QPtrList<gwire>& wires);
void lawires();
void lawires(QPtrList<gwire> wires);
QValueVector< QValueVector<double> >
getBlockagesgo( QValueVector< QPair<double,double> >& ratio,
QPtrList<gwire> wires) const;
QMap< QPair<int,int>, double> getInterferencego(QPtrList<gwire> wires)
const;
void net();
void netmain();
QValueList<QPair<layerDirectionType,int> > cuts() const;
QValueList<CPathCut>
CPXbuildnet(CPXENVptr& env, CPXNETptr& net,
CLayer* layer, const int track) const;
int* CPXoptnet(CPXENVptr& env, CPXNETptr& net,
const int numEdges, const int cellCap) const;
int CPXoptnet(CPXENVptr& env, CPXNETptr& net,
const int numEdges, const int start, int* val, bool* done)
const;
int ripuppaths();
void lapaths();
void lapaths(QValueList<QPair<gwire*,gpath*> > paths);
QValueVector< QValueVector<double> >
getBlockagesnet(QValueVector< QPair<double,double> >& ratio,
QValueList<QPair<gwire*,gpath*> > paths) const;
QMap< QPair<int,int>, double>
getInterferencenet(QValueList<QPair<gwire*,gpath*> > paths)
const;
void pp();
void pp1();
void pp2();
void pp3();
void pp4();
void maze();
void maze(gwire* wire);
public://io
bool saveBlk(const QString fileName);
bool saveFlow(const QString fileName);
bool rteRead(const QString fileName);
bool rteWrite(const QString fileName) const;
bool grteWrite(const QString fileName) const;
bool savePic(const QString fileName, const QString netName) const;
//bool rptCctTim(const QString fileName) const;
public:
CGRoute();
virtual ~CGRoute();
};

#endif


#ifndef CLAYER_H
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#define CLAYER_H

#include "global.h"
#include <qpainter.h>

class CLayer
{
private:
QString m_name;
bool m_selected;
int m_index;
layerDirectionType m_dir;bool m_hasDirection;
double m_pitch;bool m_hasPitch;
double m_width;bool m_hasWidth;
double m_res;bool m_hasRes;
double m_cap;bool m_hasCap;
int m_numcTracks;
int m_numtTracks;
int m_numcEdges;
int m_numtEdges;
int m_cwidth;
int m_cheight;
int m_twidth;
int m_theight;
int m_cellCap;
int m_tileCap;
CRect m_dieArea;
CSize m_gCell;
QSize m_gTile;
QValueVector< QValueVector<int> > m_cellBlock;
QValueVector< QValueVector<int> > m_cellFlow;
QValueVector< QValueVector<int> > m_tileBlock;
QValueVector< QValueVector<int> > m_tileFlow;
double m_len;
double m_bUsage;
double m_nUsage;
CLayer* m_next;
CLayer* m_sameNext;
CLayer* m_prev;
CLayer* m_samePrev;
public://get
bool hasDir() const { return m_hasDirection; };
bool hasPitch() const { return m_hasPitch; };
bool hasWidth() const { return m_hasWidth; };
bool hasRes() const { return m_hasRes; };
bool hasCap() const { return m_hasCap; };
public://get
QString name() const { return m_name; };
bool selected() const { return m_selected; };
int index() const { return m_index; };
layerDirectionType dir() const {return m_dir; };
double pitch() const {return m_pitch; };
double width() const {return m_width; };
double res() const {return m_res; };
double cap() const {return m_cap; };
double len() const { return m_len; };
double bUsage() const { return m_bUsage; };
double nUsage() const { return m_nUsage; };
double usage() const { return m_nUsage/(m_len-m_bUsage); };
CLayer* next(const bool same=false)
{ if(same) return m_sameNext; else return m_next; };
CLayer* prev(const bool same=false)
{ if(same) return m_samePrev; else return m_prev; };
CSize gCell() const { return m_gCell; };
CRect dieArea() const { return m_dieArea; };
int cellCap() const { return m_cellCap; };
int tileCap() const { return m_tileCap; };
int numcTracks() const { return m_numcTracks; };
int numcEdges() const { return m_numcEdges; };
int numtTracks() const { return m_numtTracks; };
int numtEdges() const { return m_numtEdges; }
int cwidth() const { return m_cwidth; };
int cheiht() const { return m_cheight; };
int cellBlock(const int track,const int edge) const
{ return m_cellBlock[track][edge]; };
int tileBlock(const int track,const int edge) const
{ return m_tileBlock[track][edge]; };
int cellFlow(const int track,const int edge) const
{ return m_cellFlow[track][edge]; };
int tileFlow(const int track,const int edge) const
{ return m_tileFlow[track][edge]; };
bool cellOverflows(const int track, const int edge) const
{ return m_cellFlow[track][edge] + m_cellBlock[track][edge] >
m_cellCap; };
bool cellFull(const int track, const int edge) const
{ return m_cellFlow[track][edge] + m_cellBlock[track][edge] >=
m_cellCap; };
 bool cellBlocked(const int track, const int edge) const
{ return m_cellBlock[track][edge] >= m_cellCap; };
bool tileOverflows(const int track, const int edge) const
{ return m_tileFlow[track][edge] + m_tileBlock[track][edge] >
m_tileCap; };
QPair<QRect,QRect> divideAtTrack(const  int  track) const;
public://set
void dir(const layerDirectionType dir) {m_dir=dir;m_hasDirection=true;};
void pitch(const double pitch) {m_pitch=pitch;m_hasPitch=true;};
void width(const double width) {m_width=width;m_hasWidth=true;};
void res(const double res) { m_res=res;m_hasRes=true;};
void cap(const double cap) { m_cap=cap;m_hasCap=true;};
void len(const double len) { m_len=len;};
void bUsage(const double usg) { m_bUsage=usg;};
void nUsage(const double usg) { m_nUsage=usg;};
void decnUsage(const double usg) {m_nUsage-=usg;};
void incnUsage(const double usg) {m_nUsage+=usg;};
void next(const bool same, CLayer* next)
{ if(same) m_sameNext = next; else m_next=next; };
void prev(const bool same, CLayer* prev)
{ if(same) m_samePrev = prev; else m_prev=prev; };
void cellBlock(const int track, const int edge, const int blk)
{ m_cellBlock[track][edge] = blk; };
void tileBlock(const int track, const int edge, const int blk)
{ m_tileBlock[track][edge] = blk; };
void cellFlow(const int track, const int edge, const int flw)
{ m_cellFlow[track][edge] = flw; };
void tileFlow(const int track, const int edge, const int flw)
{ m_tileFlow[track][edge] = flw; };
void incCellFlow(const int track, const int edge)
{ m_cellFlow[track][edge]++; };
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
void incTileFlow(const int track, const int edge)
{ m_tileFlow[track][edge]++; };
void decCellFlow(const int track, const int edge)
{ m_cellFlow[track][edge]--; };
void decTileFlow(const int track, const int edge)
{ m_tileFlow[track][edge]--; };
public://util
void init(const CRect dieArea, const CSize gCell, const QSize gTile);
void init(const QValueList<CRect> rects);
void paint(QPainter* p);
public:
friend bool operator==(const CLayer& l1, const CLayer& l2)
{ return l1.m_index==l2.m_index; };
friend bool operator<(const CLayer& l1, const CLayer& l2)
{ return l1.m_index<l2.m_index; };
friend bool operator>(const CLayer& l1, const CLayer& l2)
{ return l1.m_index>l2.m_index; };
friend bool operator<=(const CLayer& l1, const CLayer& l2)
{ return l1.m_index<=l2.m_index; };
friend bool operator>=(const CLayer& l1, const CLayer& l2)
{ return l1.m_index>=l2.m_index; };
public:
CLayer(const QString name, const int index,const bool selected);
virtual ~CLayer();
};

#endif
#ifndef CLAYERPAIR_H
#define CLAYERPAIR_H

#include <stdlib.h>
#include <qpair.h>
class CLayer;

class CLayerPair : public QPair<CLayer*,CLayer*>
{
public:
CLayer* ll() const;
CLayer* ul() const;
CLayer* vl() const;
CLayer* hl() const;
CLayerPair& operator++(int);
CLayerPair& operator--(int);
bool isNull() const { return first==NULL && second==NULL; };
friend bool operator==(const CLayerPair& lp1, const CLayerPair& lp2);
public:
CLayerPair(CLayer* l1=NULL, CLayer* l2=NULL);
CLayerPair(const CLayerPair& layerPair);
virtual ~CLayerPair();
};

#endif
#ifndef CLEF_H
#define CLEF_H

#include "global.h"
#include <lefrReader.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
```

```
class CLefLayer;
class CLefMacro;
class CLefVia;

class CLef
{
private:
double m_version; QString m_versionStr; bool m_hasVersion;
bool m_caseSensitive; bool m_hasCaseSensitive;
QString m_busBitChars;bool m_hasBusBitChars;
QString m_dividerChar;bool m_hasDividerChar;
QDict<CLefLayer> m_layers;
QDict<CLefMacro> m_macros;
QDict<CLefVia> m_vias;
public: //set
void version(const double version)
{m_hasVersion=true;m_version=version;};
void versionStr(const QString version)
{ m_hasVersion=true; m_versionStr=version; };
void caseSensitive(const bool caseSensitive)
{m_hasCaseSensitive=true; m_caseSensitive=caseSensitive; };
void busBitChars(const QString busBit)
{m_hasBusBitChars=true;m_busBitChars=busBit ; };
void dividerChar(const QString divider)
{m_hasDividerChar=true;m_dividerChar=divider; };
public: //get
bool hasVersion() const { return m_hasVersion; };
bool hasCaseSensitive() const { return m_hasCaseSensitive; };
bool hasBusBitChars() const { return m_hasBusBitChars; };
bool hasDividerChar() const { return m_hasDividerChar; };
public: //reader callbacks
friend int lefrAntennaInoutCbk(lefrCallbackType_e type,
double inout, lefiUserData data);
friend int lefrAntennaInputCbk(lefrCallbackType_e type,
double input, lefiUserData data);
friend int lefrAntennaOutputCbk(lefrCallbackType_e type,
double output, lefiUserData data);
friend int lefrBusBitCharsCbk(lefrCallbackType_e type,
const char* busBits, lefiUserData data);
friend int lefrClearanceMeasureCbk(lefrCallbackType_e type,
const char* clearance, lefiUserData data);
friend int lefrDividerCharCbk(lefrCallbackType_e type,
const char* divider, lefiUserData data);
friend int lefrLibraryEndCbk(lefrCallbackType_e type,
void* ptr, lefiUserData data);
friend int lefrLayerCbk(lefrCallbackType_e type,
lefiLayer* layer, lefiUserData data);
friend int lefrMacroBeginCbk(lefrCallbackType_e type,
const char* macro, lefiUserData data);
friend int lefrMacroCbk(lefrCallbackType_e type,
lefiMacro* macro, lefiUserData data);
friend int lefrMacroClassTypeCbk(lefrCallbackType_e type,
const char* macroClass, lefiUserData data);
friend int lefrObstructionCbk(lefrCallbackType_e type,
 lefiObstruction* obs, lefiUserData data);
friend int lefrPinCbk(lefrCallbackType_e type,
lefiPin* pin, lefiUserData data);
friend int lefrManufacturingCbk(lefrCallbackType_e type,
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
double grid,lefiUserData data);
friend int lefrCaseSensitiveCbk(lefrCallbackType_e type,
int caseSensitive,lefiUserData data);
friend int lefrNoWireExtensionCbk(lefrCallbackType_e type,
const char* wireExtension,lefiUserData data);
friend int lefrNonDefaultCbk(lefrCallbackType_e type,
lefiNonDefault* def, lefiUserData data);
friend int lefrPropBeginCbk(lefrCallbackType_e type,
void* ptr,lefiUserData data);
friend int lefrPropCbk(lefrCallbackType_e type,
lefiProp* prop,lefiUserData data);
friend int lefrPropEndCbk(lefrCallbackType_e type,
void* ptr,lefiUserData data);
friend int lefrSpacingBeginCbk(lefrCallbackType_e type,
void* ptr,lefiUserData data);
friend int lefrSpacingCbk(lefrCallbackType_e type,
lefiSpacing* spacing, lefiUserData data);
friend int lefrSpacingEndCbk(lefrCallbackType_e type,
void* ptr,lefiUserData data);
friend int lefrSiteCbk(lefrCallbackType_e type,
lefiSite* site, lefiUserData data);
friend int lefrUnitsCbk(lefrCallbackType_e type,
lefiUnits* unit, lefiUserData data);
friend int lefrUseMinSpacingCbk(lefrCallbackType_e type,
lefiUseMinSpacing* minspacing, lefiUserData data);
friend int lefrVersionCbk(lefrCallbackType_e type,
 double version, lefiUserData data);
friend int lefrVersionStrCbk(lefrCallbackType_e type,
const char* version,lefiUserData data);
friend int lefrViaCbk(lefrCallbackType_e type,
lefiVia* via, lefiUserData data);
friend int lefrViaRuleCbk(lefrCallbackType_e type,
lefiViaRule* viarule, lefiUserData data);
public://writer callbacks
friend int lefwAntennaCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwBusBitCharsCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwClearanceMeasureCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwDividerCharCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwExtCbk(lefwCallbackType_e type, lefiUserData data);
//later
friend int lefwEndLibCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwLayerCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwMacroCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwManufacturingGridCbk(lefwCallbackType_e type,
lefiUserData data);
  friend int lefwCaseSensitiveCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwNonDefaultCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwNoWireExtensionCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwPropDefCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwSiteCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwSpacingCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwUnitsCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwUseMinSpacingCbk(lefwCallbackType_e type, lefiUserData
data);
friend int lefwVersionCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwViaCbk(lefwCallbackType_e type, lefiUserData data);
friend int lefwViaRuleCbk(lefwCallbackType_e type, lefiUserData data);

private: //helpers
QString m_currMacro;
unsigned int m_layerIndex;

public://get
CLefLayer* layer(const QString name) const;
QPtrVector<CLefLayer> routingLayers() const;
CLefMacro* macro(const QString name) const { return m_macros[name]; };
CLefVia* via(const QString name) const { return m_vias[name]; };

public: //utility functions
bool load(const QString fileName);
bool save(const QString fileName);

public:
CLef();
virtual ~CLef();
};

#endif
#ifndef CLEFGEOM_H
#define CLEFGEOM_H

#include "global.h"
class lefiGeometries;

//gemotries
class CLefGeom
{
protected:
QValueList< QPair<geomType, void*> > m_items;
public://set
void addLayer(const QString layer);
void addRect(const QString layer, const CRect rect);
public: //get
int numItems() const { return m_items.count(); };
QValueList< QPair<geomType, void*> > items() const
{ return m_items;};
public://io
void lefr(lefiGeometries* lefiGeo);
public:
CLefGeom();
CLefGeom(lefiGeometries* lefiGeo);
virtual ~CLefGeom();
};


#endif
#ifndef CLEFLAYER_H
#define CLEFLAYER_H

#include "global.h"
class lefiLayer;
```

```
//layer
class CLefLayer
{
private:
unsigned int m_index;
QString m_name;
layerType m_type; bool m_hasType;
layerDirectionType m_direction; bool m_hasDirection;
double m_pitch; bool m_hasPitch;
double m_offset;bool m_hasOffset;
double m_width; bool m_hasWidth;
double m_resistance;bool m_hasResistance;
double m_capacitance; bool m_hasCapacitance;
double m_edgeCap; bool m_hasEdgeCap;
public: //set
void setName(const QString name)
{ m_name=name; };
void setType(const layerType type)
{ m_hasType=true;m_type=type; };
void setPitch(const double pitch)
{ m_hasPitch=true;m_pitch=pitch;};
void setOffset(const double offset)
{ m_hasOffset=true;m_offset=offset; };
void setWidth(const double width)
{ m_hasWidth=true;m_width=width; };
void setDirection(const layerDirectionType direction)
{ m_hasDirection=true;m_direction=direction; };
void setResistance(const double resistance)
{ m_hasResistance=true;m_resistance=resistance; };
void setCapacitance(const double capacitance)
{ m_hasCapacitance=true;m_capacitance=capacitance; };
void setEdgeCap(const double edgeCap)
{ m_hasEdgeCap=true;m_edgeCap=edgeCap; };
public: //get
unsigned int index() const { return m_index; };
QString name() const { return m_name; };
double width() const { return m_width; };
double pitch() const { return m_pitch; };
double offset() const { return m_offset; };
double resistance() const { return m_resistance; };
double unitRes() const;
double capacitance() const { return m_capacitance; };
double edgeCap() const { return m_edgeCap; };
double unitCap() const;
layerDirectionType direction() const { return m_direction; };
layerType type() const { return m_type; };
bool hasType() const { return m_hasType; };
bool hasPitch() const { return m_hasPitch; };
bool hasOffset() const { return m_hasOffset; };
bool hasWidth() const { return m_hasWidth; };
bool hasDirection() const { return m_hasDirection; };
bool hasResistance() const { return m_hasResistance; };
bool hasCapacitance() const { return m_hasCapacitance; };
bool hasEdgeCap() const { return m_hasEdgeCap; };
public://io
void lefr(lefiLayer* layer);
void lefw() const;
void lefwCut() const;
void lefwMasOvr() const;
void lefwRout() const;
public:
CLefLayer(const QString name, const int index);
CLefLayer(lefiLayer* layer, const int index);
virtual ~CLefLayer();
};

#endif
#ifndef CLEFMACRO_H
#define CLEFMACRO_H

#include "global.h"

class lefiMacro;
class CLefPin;
class CLefObs;

//macro
class CLefMacro
{
private:
QString m_name;
CSize m_size;bool m_hasSize;
CPoint m_origin; bool m_hasOrigin;
QDict<CLefPin> m_pins;
QPtrList<CLefObs> m_obs;
public://set
void setName(const QString name) { m_name=name; };
void setSize(const double width, const double height)
{m_hasSize=true; m_size.width(width);m_size.height(height);};
void setOrigin(const double x, const double y)
{m_hasOrigin=true; m_origin.x(x);m_origin.y(y);};
void addPin(CLefPin* pin);
void addObs(CLefObs* obs);
public: //get
QString name() const { return m_name; };
double sizeX() const { return m_size.width(); };
double sizeY() const { return m_size.height(); };
double originX() const { return m_origin.x(); };
double originY() const { return m_origin.y(); };
QDict<CLefPin> pins() const {return m_pins;};
CLefPin* pin(const QString pin) const { return m_pins[pin]; };
QPtrList<CLefObs> obs() { return m_obs;};
public://get
bool hasSize() const { return m_hasSize; };
bool hasOrigin() const { return m_hasOrigin; };
public://io
void lefr(lefiMacro* macro);
void lefw() const;
public:
CLefMacro(const QString name);
virtual ~CLefMacro();
};


#endif
#ifndef CLEFOBS_H
#define CLEFOBS_H
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#include "global.h"
#include "clefgeom.h"

class lefiGeometries;

//macro obstruction
class CLefObs : public CLefGeom
{
public://io
void lefw() const;
public:
CLefObs();
CLefObs(lefiGeometries* lefiGeo);
virtual ~CLefObs();
};

#endif
#ifndef CLEFPIN_H
#define CLEFPIN_H

#include "global.h"

class lefiPin;
class CLefPort;

//macro pin
class CLefPin
{
private:
QString m_name;
pinDirectionType m_direction;bool m_hasDirection;
useType m_use;bool m_hasUse;
QPtrList<CLefPort> m_ports;
public: //set
void setName(const QString name) {m_name=name;};
void setDirection(const pinDirectionType direction)
{m_hasDirection=true;m_direction=direction; };
void setUse(const useType use) { m_hasUse=true; m_use=use; };
void addPort(CLefPort* port) { m_ports.append( port ); };
public: //get
QString name() const { return m_name; };
int numPorts() const { return m_ports.count(); };
QPtrList<CLefPort> ports() const { return m_ports; };
pinDirectionType direction() const { return m_direction; };
useType use() const { return m_use; };
public://get
bool hasDirection() const { return m_hasDirection; };
bool hasUse() const { return m_hasUse; };
public://io
void lefr(lefiPin* pin);
void lefw();
public:
CLefPin(const QString name);
CLefPin(lefiPin* pin);
virtual ~CLefPin();
};

#endif
#ifndef CLEFPORT_H
#define CLEFPORT_H

#include "global.h"
#include "clefgeom.h"
class lefiGeometries;

//pin ports
class CLefPort : public CLefGeom
{
public://io
void lefw() const;
public:
CLefPort();
CLefPort(lefiGeometries* lefiGeo);
virtual ~CLefPort();
};

#endif
#ifndef CLEFVIA_H
#define CLEFVIA_H

#include "global.h"

class lefiVia;

class CLefVia
{
private:
QString m_name;
QValueList< QPair<QString, QValueList<CRect> > > m_data;
public:
QValueList< QPair<QString, QValueList<CRect> > > data() const
{ return m_data; };
public://io
void lefr(lefiVia* lefiVia);
void lefw();
public:
CLefVia(lefiVia* lefiVia);
virtual ~CLefVia();
};

#endif
#ifndef CLIB_H
#define CLIB_H

#include "global.h"

class CCctImage;

class CLib
{
private:
QDict<CCctImage> m_images;
public://get
CCctImage* image(const QString image) const
{ return m_images[image]; };
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public:
bool load(const QString fileName);
bool save(const QString fileName);
public:
CLib();
virtual ~CLib();
};

#endif

#ifndef CMAZE_H
#define CMAZE_H

#include "global.h"
#include "coord.h"
#include "cmtile.h"

class CMNode;
class CLayer;
class gwire;

class CMaze
{
private:
gwire* m_wire;
CTile m_from;
CTile m_to;
QPtrVector<CLayer> m_layers;
int m_layerChangeCost;
public:
void layerChangeCost(const int cost)
{ m_layerChangeCost = cost;};
public:
bool route();
int costEstimate(CMTile* from);
int travelCost(CMTile *fromNode, CMTile *toNode);
bool isLast(CMTile* tile);
QPtrList<CMTile> succ(CMTile* node);
CMTile* find(QPtrList<CMTile> set,CMTile* node);
public:
CMaze(CTile from, CTile to, QPtrVector<CLayer> layers, gwire* wire);
~CMaze();
};

#endif


#ifndef CMNODE_H
#define CMNODE_H


class CMNode
{
public:
CMNode();
~CMNode();
};

#endif
```

```
#ifndef CMTILE_H
#define CMTILE_H

#include "ctile.h"

class CMTile : public CTile
{
private:
CMTile* m_parent;
int m_costFromStart;
int m_costToGoal;
public:
void parent(CMTile* parent) {m_parent=parent;};
void costFromStart(const int cost) {m_costFromStart = cost;};
void costToGoal(const int cost) {m_costToGoal=cost;};
public:
CMTile* parent() { return m_parent; };
int costFromStart() const { return m_costFromStart; };
int costToGoal() const { return m_costToGoal; };
public:
CMTile();
CMTile(const CMTile& tile);
CMTile(CTile& tile, CMTile* parent=NULL);
CMTile(CLayer* layer, int x, int y);
~CMTile();
};

#include <qptrlist.h>

class CSortedPtrList :public QPtrList<CMTile>
{
protected:
virtual int
compareItems(QPtrCollection::Item item1, QPtrCollection::Item
item2);
// virtual int
compareItems(QPtrCollection::Item item1, QPtrCollection::Item
item2 );
public:
CSortedPtrList();
~CSortedPtrList();
};

#endif
#ifndef CNET_H
#define CNET_H

#include "global.h"
#include "cedge.h"
#include "clayerpair.h"
class gwire;
class CNode;
class CBuffer;
class CCctPath;

class CNet : public QString, public QPtrList<gwire>
{
private:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CNode* m_root;
QPtrList<CNode> m_nodes;
double m_maxDV;
CLayerPair m_layer;
public://get
CLayerPair layer() const { return m_layer; };
CNode* root() const { return m_root; };
CNode* pin(QString pin) const;
QPtrList<CNode>& nodes() { return m_nodes; };
QValueList<CEdge> edges() const;
double maxDV() const { return m_maxDV; };
     double DVImp() const;
double length() const;
int numNodes() const { return m_nodes.count(); };
int numBuffers() const;
public://set
void layer(CLayerPair layer);
void root(CNode* root);
void makeRoot(const double res, const double cap);
void addSink(CNode* sink) { m_nodes.append( sink ); };
void removeSink(CNode* sink) { m_nodes.remove( sink ); };
void removeNode(CNode* node) { m_nodes.remove( node ); };
void maxDV(const double maxDV) { m_maxDV=maxDV; };
     void saveDV();
public://util
void joinNodeAtEdge(CNode* node, const CEdge edge);
void joinNodeAtPoint(CNode* node, const CEdge edge, const CPoint p);
void disjoinNodeFromEdge(CNode* node, const CEdge edge);
void connectNodes(CNode* up, CNode* down);
void segment(const double lump);
void desegment();
bool insertBufferAtEdge(const CEdge edge,
const double load, const double res, const double cap);
  bool insertBufferAtNode(const CEdge edge,
const double load, const double res, const double cap);
void removeBufferFromEdge(const CEdge edge);
     void removeBufferFromNode(const CEdge edge);
void removeBuffers();
     void removeBuffer(CNode* n, bool perm);
     void reInsertBuffer(CNode* n,  const double load, const double res,
const double cap);
void computeCapAndR();
void buildTree(CNode* rootN, QString root, int x, int y,
const QPtrList<CCctPath> T, QPtrList<CCctPath>* P, QTextStream&
f);
public://io
void rteWrite(QTextStream& f);
void grteWrite(QTextStream& f);
void paint(QPainter* p);
    void dump(QString msg);
public:
CNet(const QString name);
virtual ~CNet();
};

#endif
#ifndef CNETLIST_H
#define CNETLIST_H
```

```
#include "global.h"

class CCctNet;

class CNetList
{
private:
QDict<CCctNet> m_nets;
public:
QDict<CCctNet> nets() const
{ return m_nets; };
CCctNet* net(const QString name) const
{ return m_nets[name]; };
public://io
bool load(QString fileName);
bool save(QString fileName);
public:
CNetList();
virtual ~CNetList();
};

#endif
#ifndef CNETPIN_H
#define CNETPIN_H

#include "global.h"

//net pin
class CNetPin
{
private:
QString m_instance;
QString m_name;
int m_synthesized;
public: //get
QString instance() const
{ return m_instance; };
QString name() const { return m_name; };
public: //io
void defw() const;
public:
CNetPin(const QString instance,const QString name,const int
synthesized);
virtual ~CNetPin();
};

#endif
#ifndef CNODE_H
#define CNODE_H

#include "global.h"
#include "cedge.h"

class CNode : public CPoint
{
protected:
nodeType m_type;
CEdge m_edge;
CNode* m_parent;
```

```
QPtrList<CNode> m_children;
double m_cap;
QValueVector<double> m_moment;
double m_delay;
public://get
nodeType type() const { return m_type; };
CEdge& edge() { return m_edge; };
CNode* parent() const { return m_parent; };
QPtrList<CNode>& children() { return m_children; };
virtual double cap() const { return m_cap; };
virtual QString layer() const { return QString(); };
virtual QString name() const { return ::nodeTypeStr(m_type);};
double delay() const { return m_delay; };
double moment(const int i) const { return m_moment[i]; };
public://set
void type(const nodeType type) { m_type = type; };
void edge(CEdge edge) { m_edge = edge; };
void parent(CNode* parent) { m_parent=parent; };
void addChild(CNode* child) { m_children.append(child); };
void removeChild(CNode* child) { m_children.remove(child); };
void delay(const double delay) { m_delay = delay; };
public://util
void resizeMoment(const int size) { m_moment.resize(size); };
void computeCapAndR();
void resetSubtree(const int ord);
void computeCurrent(const int ord);
void computeMoment(const int ord);
void moment(double mom, int i) { m_moment[i] = mom; };
public:
CNode(const nodeType type, const CPoint point);
virtual ~CNode();
};

#endif

#ifndef COORD_H
#define COORD_H

class Rectangle;
class Point;

class Coord
{
private:
int m_x;
int m_y;
int m_z;
public:
int x() const { return m_x; };
int y() const { return m_y; };
int z() const { return m_z; };
void setCoord(int x, int y, int z);
bool intersects(const Coord *c, Rectangle *r) const;
Point *center() const;
public:
bool operator<(const Coord &c) const;
bool operator==(const Coord &c) const;
public:
Coord(int x=0, int y=0, int z=0);
~Coord();
};

#endif

#ifndef CPATHCUT_H
#define CPATHCUT_H

#include "global.h"
class gpath;
class gwire;

class CPathCut : public QPair< QPair<int,int>,QPtrList<gpath> >
//lb,ub,path list
{
private:
gwire* m_wire;
public:
void lb(const int lb)
{ first.first = ::min(first.first,lb); };
void ub(const int ub)
{ first.second = ::max(first.second,ub); };
void append(gpath* path)
{ second.append(path); };
public:
gwire* wire() const { return m_wire; };
int lb() const { return first.first; };
int ub() const { return first.second; };
QPtrList<gpath> paths() const { return second; };
public:
CPathCut(gwire* wire, const  int lb, const int ub, gpath* path);
CPathCut();
virtual ~CPathCut();
};

#endif
/***************************************************************************
                          cpla.h  -  description
                             -------------------
    begin                : Thu Oct 9 2003
    copyright            : (C) 2003 by Cengiz ALKAN
    email                : alkan@shavano.engr.colostate.edu
 ***************************************************************************/
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef CPLA_H
#define CPLA_H

#include "global.h"

class CCctComp;

class CPla
{
private:
QDict<CCctComp> m_comps;
public://get
QDict<CCctComp> comps() const { return m_comps; };
CCctComp* comp(const QString comp) const { return m_comps[comp]; };
public://io
bool  load(const QString fileName);
bool save(const QString fileName);
public:
CPla();
virtual ~CPla();
};

#endif

#ifndef CPOINT_H
#define CPOINT_H

#include <qtextstream.h>
#include <cmath>

//point class
class CPoint
{
private:
double m_x;
double m_y;
public: //set
void x(const double x) { m_x=x; };
void y(const double y) { m_y=y; };
public: //get
double x() const { return m_x; };
double y() const { return m_y; };
public://utils
static double manhDist(const CPoint& p1, const CPoint& p2)
{ return fabs(p1.x()-p2.x())+fabs(p1.y()-p2.y()); };
static double xDist(const CPoint& p1, const CPoint& p2)
{ return  fabs(p1.x()-p2.x()); };
static double yDist(const CPoint& p1, const CPoint& p2)
{ return  fabs(p1.y()-p2.y()); };
friend bool operator==(const CPoint& p1, const CPoint& p2);
public:
CPoint(const double x=0.0, const double y=0.0);
CPoint(const CPoint& p);
virtual ~CPoint();
friend QTextStream& operator<<(QTextStream& dmp, CPoint& p);
};

#endif
#ifndef CPOLY_H
#define CPOLY_H

#include <qvaluelist.h>
#include "cpoint.h"
#include "crect.h"
//polygon item geom type
class CPoly : public QValueList<CPoint>
{
public:
void append(const double x, const double y)
{ push_back(CPoint(x,y));};
public:
CRect boundingRect() const;
public:
CPoly();
virtual ~CPoly();
};

#endif
#ifndef CPTRLIST_H
#define CPTRLIST_H

#include <qptrlist.h>

template<class type>
class CPtrList : public QPtrList<type>
{
public:
CPtrList(type* from, type* to,
const bool properFrom, const bool properTo)
{
QPtrList<type>::setAutoDelete(false);
type *begin, *end;
type *_from,*_to;
begin = end = _from = _to = NULL;

if(from==NULL) _from=to; else _from=from;
if(to==NULL) _to=from; else _to=to;

if(_from==NULL || _to==NULL) return;

if(properFrom) begin = _from;
else if(*from<*to) begin = _from->next();
else if(*from>*to) begin = _from->prev();

if(properTo) end = _to;
else if(*from<*to) end = _to->prev();
else if(*from>*to) end = _to->next();

if(*_from<*_to)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for(type* i=begin;i!=NULL && *i<=*end;i=i->next()) append(i);
else if(*_from>*_to)
for(type* i=begin;i!=NULL && *i>=*end;i=i->prev())
append(i);
};
CPtrList():QPtrList<type>::QPtrList()
{};
virtual ~CPtrList()
{
QPtrList<type>::clear();
};
};
#endif
#ifndef CRECT_H
#define CRECT_H


#include <qrect.h>
#include "cpoint.h"

//rectangle utility class for double numbers
class CRect
{
private:
double m_left;
double m_bottom;
double m_right;
double m_top;
public: //set
void left(const double left) {m_left = left; };
void right(const double right) { m_right=right; };
void bottom( const double bottom) { m_bottom=bottom;};
void top(const double top) {m_top=top; };
public://get
double left() const { return m_left; };
double bottom() const { return m_bottom; };
double right() const { return m_right; };
double top() const { return m_top; };
double width() const { return m_right-m_left; };
double height() const { return m_top-m_bottom; };
CPoint bottomLeft() const { return CPoint(m_left,m_bottom); };
CPoint bottomRight() const { return CPoint(m_right,m_bottom); };
CPoint topLeft() const { return CPoint(m_left,m_top); };
CPoint topRight() const { return CPoint(m_right,m_top); };
double area() const { return width()*height(); };
CPoint center() const
{ return CPoint( (m_left+m_right)/2.0, (m_bottom+m_top)/2.0 );};
public://util
    QRect toQRect() const;
CRect orientDefPin(const CPoint placement, const int orient) const;
CRect orientLefPin(const CRect rec, const int orient) const;
CRect scale(const double s) const;
CRect translate(const double dx, const double dy) const;
CRect translate(CPoint p) const;
bool intersects(const CRect rect) const;
CRect intersect(const CRect rect) const;
CRect unite(const CRect rect) const;
CRect normalize() const;
public://operators
CRect operator/(const double s) const { return scale(1.0/s); };
CRect operator*(const double s) const { return scale( s ); };
CRect operator+(CPoint p) const { return translate(p); };
CRect operator&(CRect r) const { return intersect(r); };
CRect operator|(CRect r) const { return unite(r); };
public:
CRect(const double left=0.0, const double bottom=0.0,
const double width=0.0, const double height=0.0);
CRect(const CPoint bottomLeft, const CPoint topRight);
CRect(const QRect r);
virtual ~CRect();
};


#endif
#ifndef CRTE_H
#define CRTE_H

#include "global.h"

class CCctNet;

class CRte
{
public:
int m_res;
QDict<CCctNet> m_nets;
public:
CCctNet* net(const  QString net) const
{ return m_nets[net]; };
QDict<CCctNet> nets() const { return m_nets; };
int res() const { return m_res;  };
public:
bool load(const QString fileName);
bool save(const QString fileName);
public:
CRte();
virtual ~CRte();
};

#endif
#ifndef CSINK_H
#define CSINK_H

#include "global.h"
#include "cnode.h"

class CSink : public QString, public CNode
{
private:
QString m_layer;
double m_load;
double m_rat;
    double m_sDV;
public:
void rat(const double rat) { m_rat=rat;};
    void saveDV() { m_sDV = m_delay-m_rat; };
public:
double load() const { return m_load; };
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
double rat() const { return m_rat; };
virtual double cap() const { return m_load+m_cap; };
virtual QString layer() const { return m_layer; };
virtual QString name() const { return *this; };
double dv() const { return m_delay-m_rat; };
    double DVImp() const;
public:
CSink( const QString name,
const QString layer,
const CPoint point,
const double load,
const double rat);
virtual ~CSink();
};

#endif
#ifndef CSIZE_H
#define CSIZE_H

//size utility class
class CSize
{
private:
double m_width;
double m_height;
public://set
void width(const double width) { m_width=width; };
void height(const double height) { m_height=height; };
public: //get
double width() const { return m_width; };
double height() const { return m_height; };
public:
CSize(const double width=0.0, const double height=0.0 );
virtual ~CSize();
};

#endif
#ifndef CSOURCE_H
#define CSOURCE_H

#include "global.h"
#include "cnode.h"

class CSource : public QString, public CNode
{
private:
QString m_layer;
double m_outRes;
double m_outCap;
public:
double outRes() const { return m_outRes; };
double outCap() const { return m_outCap; };
virtual double cap() const { return m_cap + m_outCap; };
virtual QString layer() const { return m_layer; };
virtual QString name() const { return *this; };
public:
CSource(const QString name,
const QString layer,
const CPoint point,
const double res,
const double cap);
virtual ~CSource();
};

#endif
#ifndef CSTR_H
#define CSTR_H

#include "global.h"

class CCctLayer;
class CCctKeepout;

class CStr
{
private:
CPoly m_diePoly;bool m_hasDiePoly;
QDict<CCctLayer> m_layers;
QPtrList<CCctKeepout> m_keepouts;
public://get
bool hasDiePoly() const { return m_hasDiePoly; };
public://get
CPoly diePoly() const { return m_diePoly; };
CCctLayer* layer(const QString name) const;
QPtrList<CCctKeepout> keepouts() const
{ return m_keepouts; };
QDict<CCctLayer> layers() const { return m_layers; };
public://io
bool load(const QString fileName);
bool save(const QString fileName);
public:
CStr();
virtual ~CStr();
};

#endif
#ifndef CSUBROOT_H
#define CSUBROOT_H

#include "global.h"
#include "cnode.h"

class CSubRoot : public CNode
{
protected:
double m_outRes;
double m_outCap;
    bool m_done;
public:
    bool done() const { return m_done; };
    void done(const bool done) { m_done = done; };
double outRes() const { return m_outRes; };
    void outRes(const double outRes) { m_outRes = outRes; };
double outCap() const { return m_outCap; };
    void outCap(const double outCap) { m_outCap = outCap; };
virtual double cap() const
{ return m_outCap + m_cap; };
public:
```

```
CSubRoot(const CPoint point,
const double res, const double cap);
virtual ~CSubRoot();
};

#endif
#ifndef CTILE_H
#define CTILE_H

#include <qpoint.h>
#include <qsize.h>
#include <qrect.h>
class CLayer;

class CTile : public QPoint, public QSize
{
private:
CLayer* m_layer;
public:
CLayer* layer() const { return m_layer; };
QRect rect() const
{ return QRect(x()*width(),y()*height(),width(),height()); };
public:
CTile(CLayer* layer, const int x,
const int y, const QSize size);
CTile(const CTile& tile);
CTile();
virtual ~CTile();
};

class CTileEdge
{
private:
CLayer* m_layer;
int m_track;
int m_edge;
QSize m_size;
public:
CLayer* layer() const { return m_layer; };
int track() const { return m_track; };
int edge() const { return m_edge; };
public:
bool operator==(CTile& tile);
public:
CTileEdge(CLayer* layer, const int track,
const int edge, const QSize tileSize);
CTileEdge(const CTileEdge& tileEdge);
CTileEdge();
virtual ~CTileEdge();
};
#endif
#ifndef CTIM_H
#define CTIM_H

#include "global.h"

class CTim
{
private:
QDict< QValueList< QPair< QPair<QString,QString>, double > > >
m_riseTime;
QDict< QValueList< QPair< QPair<QString,QString>, double > > >
m_fallTime;
public://get
pinDirectionType dir(const QString net, const QString comp) const;
double rat(const QString net, const QString comp) const;
public://io
bool load(const QString fileName);
bool save(const QString fileName);
public:
CTim();
virtual ~CTim();
};
#endif
#ifndef CTIMING_H
#define CTIMING_H

#include "global.h"

class CTiming
{
private:
QDict< QValueList< QPair<QString, double> > > m_rats;
public://set
void net(const QString net);
void source(const QString net, const QString source);
void sink(const QString net, const QString sink, const double rat);
public://get
double rat(const QString net, const QString sink) const;
pinDirectionType dir(const QString net, const QString pin) const;
public://io
bool load(const QString fileName);
bool save(const QString fileName);
public:
CTiming();
virtual ~CTiming();
};
#endif
#ifndef CUTIL_H
#define CUTIL_H

#include "global.h"
class CLefLayer;
class CCctLayer;
#define MAXNUMLAYERS 20
#define MAXNUMROWS 1000
#define MAXNUMCOLS 1000
class CUtil
{
public:
bool lefdefCongWrite(const QString name);
bool cctCongWrite(const QString name);
bool conglefdefWrite(const QString name);
bool congcctWrite(const QString name);
bool lengthWrite(const QString name);
bool cctTimWrite(const QString name);
private:
void addRect(int ***map, QString layerName,
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CRect r, QPtrVector<CLefLayer> layers);
void addRect(int ***map, QString layerName,
CRect r, QPtrVector<CCctLayer> layers);
public:
CUtil();
virtual ~CUtil();
};

#endif
#ifndef CVALUELIST_H
#define CVALUELIST_H

#include <qvaluelist.h>

template<class type>
class CValueList : public QValueList<type>
{
public:
CValueList(const type from, const type to,
const bool properFrom, const bool properTo)
{
type begin,end;
if(properFrom) begin = from;
else if(from<to) begin = from +1;
else if(from>to) begin = from -1;
else begin = from;
if(properTo) end = to;
else if(from<to) end = to -1;
else if(from>to) end = to +1;
else end = to;

if(from<to) for(type i=begin;i<=end;i = i + 1) append(i);
else if(from>to) for(type i=begin;i>=end;i = i -1) append(i);
};
//edge mode for traversing edges
CValueList(const type from, const type to)
{
if(from<to) for(type i=from+1;i<=to;i++) append(i);
else if(from>to) for(type i=from;i>to;i--) append(i);
};
virtual ~CValueList()
{
QValueList<type>::clear();
};
};

#endif
#ifndef GLOBAL_H
#define GLOBAL_H


#include <iostream>
//#include <stdlib>
//#include <stdio.h>
//#include <math.h>
#include <assert.h>

#include <qfile.h>
#include <qtextstream.h>
```

```
#include <qstring.h>
#include <qregexp.h>
#include <qstringlist.h>
#include <qvaluelist.h>
#include <qvaluevector.h>
#include <qvaluestack.h>
#include <qmap.h>
#include <qdict.h>
#include <qptrvector.h>
#include <qptrlist.h>
#include <qpair.h>

//utility classes
#include "cpoint.h"
#include "csize.h"
#include "crect.h"
#include "cpoly.h"
#include "ccomplex.h"
#include "cvaluelist.h"
#include "cptrlist.h"

//lef/def definitions for enumeration constants
#include "defiComponent.hpp"
#include "lefiMacro.hpp"
#include "defiPath.hpp"

enum layerType { CUT, MASTERSLICE,  OVERLAP, ROUTING, UNKNOWNLAYERTYPE};
enum layerDirectionType { VERTICAL,HORIZONTAL, UNKNOWNLAYERDIRECTION};
enum geomType { GEOMLAYER=lefiGeomLayerE,
   GEOMLAYERMINSPACING=lefiGeomLayerMinSpacingE,
   GEOMLAYERRULEWIDTH=lefiGeomLayerRuleWidthE,
GEOMWIDTH=lefiGeomWidthE,
   GEOMPATH=lefiGeomPathE,
   GEOMPATHITER=lefiGeomPathIterE,
   GEOMRECT=lefiGeomRectE,
   GEOMRECTITER=lefiGeomRectIterE,
   GEOMPLYGON=lefiGeomPolygonE,
   GEOMPOLYGONITER=lefiGeomPolygonIterE,
   GEOMVIA=lefiGeomViaE,
   GEOMVIAITER=lefiGeomViaIterE,
   GEOMCLASS=lefiGeomClassE,
   GEOMEND=lefiGeomEnd,
UNKNOWNGEOM=lefiGeomUnknown};
enum pinDirectionType { IN, OUT, INOUT, FEEDTHRU, UNKNOWNPINDIR };
enum useType { ANALOG, CLOCK, GROUND, POWER, RESET, SCAN, SIGNAL, TIEOFF,
UNKNOWNUSE};

enum placementStatusType {  UNPLACED=DEFI_COMPONENT_UNPLACED,
  PLACED=DEFI_COMPONENT_PLACED,
FIXED=DEFI_COMPONENT_FIXED,
COVER=DEFI_COMPONENT_COVER,
UNKNOWNPLACEMENTSTATUS };

enum placementOrientType { N=0,W=1,S=2,E=3,FN=4,FW=5,FS=6,FE=7, UNKNOWNORIENT};
enum pathType { PATHDONE=DEFIPATH_DONE,
PATHLAYER=DEFIPATH_LAYER,
PATHVIA=DEFIPATH_VIA,
PATHVIAROTATION=DEFIPATH_VIAROTATION,
PATHWIDTH=DEFIPATH_WIDTH,
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
PATHPOINT=DEFIPATH_POINT,
PATHFLUSHPOINT=DEFIPATH_FLUSHPOINT,
PATHTAPER=DEFIPATH_TAPER,
PATHSHAPE=DEFIPATH_SHAPE,
PATHTAPERRULE=DEFIPATH_TAPERRULE,
PATHVIADATA=DEFIPATH_VIADATA,
UNKNOWNPATH};


enum wireType { WIRECOVER, WIREFIXED, WIREROUTED, WIRENOSHIELD, UNKNOWNWIRETYPE
};

enum nodeType {DRIVER,ROOT,SINK,BUFFER,SUBROOT,STEINER,AWE};


layerType layer( const QString layer );
QString layer(const layerType layer);
layerDirectionType layerDirection(const QString layerDirection);
QString layerDirection(const layerDirectionType layerDirection);
geomType geom(const int geomType);
pinDirectionType pinDirection(const QString pinDirection);
QString pinDirection(const pinDirectionType pinDirection);
useType use(const QString use);
QString use(const useType use);
QString nodeTypeStr(const nodeType type);

placementStatusType placementStatus(const int placementStatus );
QString placementStatus(const placementStatusType placementStatus);
placementOrientType placementOrient(const int placementOrient );
int placementOrient(const placementOrientType placementOrient );
pathType path(const  int type);
QString pathStr(const pathType type);
wireType wire(const QString type);
QString wire(const wireType type);



//constants
#define INF 1e9
#define IINF 100000
#define MAX_ORDER 4
#define TINY 1.0e-7
#define SMALL 0.01
#define PI 3.141592653589793
#define RES 100.0

//utility functions
QValueList<QRect> clipRect(QRect r1, const QRect r2);
void gauss(int, complex sys[MAX_ORDER+1][MAX_ORDER+1], complex *, complex *);
bool cubic(complex *);
bool quartic(complex *);
bool computePoles(complex*, complex*, int);
void polish(complex*, complex*, int &, bool &);

// template functions
//maximum of two
template <class type>
type max(type x, type y){if(x>y) return x; else return y;}

template<class type>
type max(QValueList<type> x)
{
if(x.isEmpty()) return -1;
type max = x.first();
QValueListIterator<type> it;
for(it=x.begin();it!=x.end();it++)
if(*it>max) max = *it;
return max;
}

//minimum of  two
template <class type>
type min(type x, type y){if(x<y) return x; else return y;}

template<class type>
type min(QValueList<type> x)
{
if(x.isEmpty()) return -1;
type min = x.first();
QValueListIterator<type> it;
for(it=x.begin();it!=x.end();it++)
if(*it<min) min = *it;
return min;
}

//median value of three
template <class type>
type median(type x, type y, type z)
{
   if (x < y) {if (x > z ) return x;else{if (y > z)
return z; else return y; }}
   else{if (y > z) return y;
else{if (x > z) return z; else return x;}}
}

#endif



#ifndef GPATH_H
#define GPATH_H

#include "global.h"
#include "clayerpair.h"
#include "ctile.h"
class CLayer;

class gpath : public QRect
{
private:
CLayerPair m_layer;
QRect m_from;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
QRect m_to;
bool m_routed;
public:
CLayerPair layer() const { return m_layer; };
QRect from() const { return m_from; };
QRect to() const { return m_to; };
layerDirectionType dir() const
{  if(width()>height()) return HORIZONTAL;
else if(height()>width()) return VERTICAL;
else return UNKNOWNLAYERDIRECTION; };
bool routed() const { return m_routed; };
bool cutsAndBounds(CLayer* layer,const int track,int& lb,int& ub) const;
public:
QValueList<CTile> tiles(QSize tileSize = QSize(1,1)) const;
QValueList<CTileEdge> edges(QSize tileSize = QSize(1,1)) const;
public:
void from(const QRect from)
{
m_from=from;
QRect r = m_from | m_to;
setLeft(r.left());setRight(r.right());
setTop(r.top());setBottom(r.bottom());
};
void to(const QRect to)
{
m_to = to;
QRect r = m_from | m_to;
setLeft(r.left());setRight(r.right());
setTop(r.top());setBottom(r.bottom());
};
void layer(const CLayerPair layer) { m_layer = layer; };
void routed(const bool routed) { m_routed = routed; };
void incTileFlows(const QSize tileSize) const;
void decTileFlows(const QSize tileSize) const;
void incCellFlows() const;
void decCellFlows() const;
public:
void rteWrite(QTextStream& f);
void grteWrite(QTextStream& f);
void paint(QPainter* p);
public:
gpath(const CLayerPair layer, const QRect from,
const QRect to, const bool routed=false);
gpath(const gpath& path);
gpath();
virtual ~gpath();
};

#endif
```

```
#ifndef GWIRE_H
#define GWIRE_H

#include "global.h"
#include "ctile.h"
#include "cpathcut.h"
#include "clayerpair.h"
class CNode;
class gpath;

class gwire : public QPtrList<gpath>
{
private:
CNode* m_from;
CNode* m_to;
bool m_routed;
public:
bool routed() const { return m_routed; };
CNode* from() const { return m_from; };
CNode* to() const { return m_to; };
public:
QValueList<CTile> tiles(const QSize tileSize = QSize(1,1) ) const;
QValueList<CTileEdge> edges(const QSize tileSize= QSize(1,1) ) const;
bool cutPathsAndBounds(CLayer* layer, const int track,
QValueList<CPathCut>& cutpaths);
QRect farfrom(gpath* path, gpath& farfrom);
QRect farto(gpath* path, gpath& farto);
public:
void routed(const bool routed) { m_routed=routed; };
// void append(gpath* path);
// void prepend(gpath* path);
gwire& operator+=(gwire& wire);
void movePath(gpath* path, CLayerPair layer=CLayerPair(),
const int dx=0, const int dy=0);
void incTileFlows(const QSize tileSize) const;
void decTileFlows(const QSize tileSize) const;
void incCellFlows() const;
void decCellFlows() const;
void rewire(QPtrList<gpath> paths, CLayer* layer,
const int track, const int edge);
void mergeStraightPaths();
void mergeExpandUnroutedPaths(const QSize gTile);
bool pp1(gpath* path);
bool pp1(gpath* path, CLayerPair layerp);
  bool pp2(gpath* path);
bool pp2(gpath* path, CLayerPair layerp);
void pp3();
void pp3(gpath* prev, gpath* next);
void pp4();
void pp4from(CPtrList<CLayer> list);
void pp4to(CPtrList<CLayer> list);
public:
void rteWrite(QTextStream& f);
void grteWrite(QTextStream& f);
int numPadViolations();
void paint(QPainter* p);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
public:
gwire(CNode* from, CNode* to);
gwire(const gwire& wire);
gwire();
virtual ~gwire();
};

#endif
#ifndef OPENSET_H
#define OPENSET_H

#include "global.h"


template <class K, class T>
class OpenSet
{
private:
QValueVector<T> theVector;
QMap<K,T> theMap;
public:
void insert(const K k, T t)
{
theVector.push_bask(t);
push_heap(theVector.begin(), theVector.end());
theMap[k] = t;
};
void erase(const K);
void update()
{
make_heap(theVector.begin(), theVector.end());
};
T pop_best()
{
T retVal = theVector.front();
pop_heap(theVector.begin(), theVector.end());
theVector.pop_back();
theMap.erase(retVal->coord());
return retVal;
};
T find(const K k)
{
if(theMap.find(k) != theMap.end())
return theMap[k];
return NULL;
};
int size() const { theVector.size(); };
public:
OpenSet();
~OpenSet();
};

#endif
#ifndef UTIL_H
#define UTIL_H

#include "global.h"

void init();
void dispose();
bool load(const QString fileName);
bool parseCmd(const QString cmd);

#endif

#include "cbuffer.h"

CBuffer::CBuffer(const CPoint point, const double load)
:CNode(BUFFER,point)
,m_load(load)
{

}
CBuffer::~CBuffer()
{

}

#include "ccctcomp.h"

CCctComp::CCctComp(const QString name, const QString image)
:m_name(name)
,m_image(image)
{
}
CCctComp::~CCctComp()
{
}
#include "ccctimage.h"
#include "ccctpin.h"
#include "ccctkeepout.h"

CCctImage::CCctImage(const QString name)
:m_name(name)
,m_hasPoly(false)
,m_hasRect(false)
{
m_pins.setAutoDelete(true);
m_keepouts.setAutoDelete(true);
}
CCctImage::~CCctImage()
{
m_pins.clear();
m_keepouts.clear();
}

void CCctImage::addPin(CCctPin* pin)
{
m_pins.insert(pin->name(),pin);
}
#include "ccctkeepout.h"

CCctKeepout::CCctKeepout(const QString layer, const CRect rect)
:m_layer(layer)
,m_rect(rect)
{
}
CCctKeepout::~CCctKeepout(){
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
#include "ccctlayer.h"

CCctLayer::CCctLayer(const QString name, const int index)
:m_name(name)
,m_hasDirection(false)
,m_hasWidth(false)
,m_hasClear(false)
,m_index(index)
{
}
CCctLayer::~CCctLayer()
{
}
#include "ccctnet.h"
#include "ccctwire.h"

CCctNet::CCctNet(const QString name)
:m_name(name)
{
m_wires.setAutoDelete(true);
}
CCctNet::~CCctNet()
{
m_pins.clear();
m_wires.clear();
}
#include "ccctpath.h"

CCctPath::CCctPath(const QString layer, const int width,
const int x1, const int y1, const int x2, const int y2)
:m_layer(layer)
,m_width(width)
,m_x1(x1)
,m_y1(y1)
,m_x2(x2)
,m_y2(y2)
{
// m_pin1 = QString().setNum(x1)+","+QString().setNum(y1);
// m_pin2 = QString().setNum(x2)+","+QString().setNum(y2);
m_pin1 = "";
m_pin2 = "";
}

CCctPath::~CCctPath()
{
}


bool CCctPath::intersects(CCctPath p)
{
return (height()==0 && p.height()==0 &&
       (bottom()==p.bottom()) &&
   (top()==p.top()) &&
   (::max(left(),p.left()) < ::min( right(), p.right() ))) ||
   (width()==0 && p.width()==0 &&
       (left()==p.left()) &&
   (right()==p.right()) &&
   (::max(bottom(),p.bottom()) < ::min(top(), p.top() )));
}

QPtrList<CCctPath> CCctPath::nonInPaths(const CCctPath p1, const CCctPath p2)
{
assert(p1.layer()==p2.layer());
//assert(p1.intersects(p2));

QPtrList<CCctPath> paths;
if(p1.height()==0 && p2.height()==0)
{
int y,x1,y1,x2,y2,width;
bool pin1,pin2,pin3,pin4;
QString layer,pin;
CCctPath* path;

assert(p1.y1()==p1.y2());
assert(p2.y1()==p2.y2());
assert(p1.y1()==p2.y1());
assert(p1.layer()==p2.layer());
y = p1.y1();
layer = p1.layer();
width = p1.Width();
pin1=pin2=pin3=pin4=false;

x1 = ::min( p1.left(), p2.left() );
y1 = y;
x2 = ::max( p1.left(), p2.left() );
y2 = y;
path = new CCctPath(layer,width,x1,y1,x2,y2);
if( x1==p1.x1() ) {path->pin1( p1.pin1() );pin1=true;}
else if ( x1==p1.x2() )
{path->pin1( p1.pin2() );pin2=true;}
else if ( x1==p2.x1() )
{path->pin1( p2.pin1() );pin3=true;}
else if ( x1==p2.x2() )
{path->pin1( p2.pin2() );pin4=true;}
if( x2==p1.x1() && !pin1)
{path->pin2( p1.pin1()
);pin1=true;pin2=pin3=pin4=false;}
else if ( x2==p1.x2() && !pin2)
{path->pin2( p1.pin2()
);pin2=true;pin1=pin3=pin4=false;}
else if ( x2==p2.x1() && !pin3)
{path->pin2( p2.pin1()
);pin3=true;pin1=pin2=pin4=false;}
else if ( x2==p2.x2() && !pin4)
{path->pin2( p2.pin2()
);pin4=true;pin1=pin2=pin3=false;}
if(path->pin1()!="" &&
path->pin2()!="" &&
path->pin1()==path->pin2()) path->pin2("");
paths.append( path );

pin = path->pin2();
x1 = x2;
y1 = y;
x2 = ::min( p1.right(), p2.right() );
y2 = y;
path = new CCctPath(layer,width,x1,y1,x2,y2);
```

```
path->pin1(pin);
if( x2==p1.x1() && !pin1)
{path->pin2( p1.pin1()
);pin1=true;pin2=pin3=pin4=false;}
else if ( x2==p1.x2() && !pin2)
{path->pin2( p1.pin2()
);pin2=true;pin1=pin3=pin4=false;}
else if ( x2==p2.x1() && !pin3)
{path->pin2( p2.pin1()
);pin3=true;pin1=pin2=pin4=false;}
else if ( x2==p2.x2() && !pin4)
{path->pin2( p2.pin2()
);pin4=true;pin1=pin2=pin3=false;}
if(path->pin1()!=""
&& path->pin2()!="" &&
path->pin1()==path->pin2()) path->pin2("");
paths.append( path );

pin = path->pin2();
x1 = x2;
y1 = y;
x2 = ::max( p1.right(), p2.right() );
y2 = y;
path = new CCctPath(layer,width,x1,y1,x2,y2);
path->pin1(pin);
if( x2==p1.x1() && !pin1) path->pin2( p1.pin1() );
else if ( x2==p1.x2() && !pin2) path->pin2( p1.pin2() );
else if ( x2==p2.x1() && !pin3) path->pin2( p2.pin1() );
else if ( x2==p2.x2() && !pin4) path->pin2( p2.pin2() );
if(path->pin1()!="" && path->pin2()!=""
&& path->pin1()==path->pin2()) path->pin2("");
paths.append( path );
}else if(p1.width()==0 && p2.width()==0)
{
int x,x1,y1,x2,y2,width;
bool pin1,pin2,pin3,pin4;
QString layer,pin;
CCctPath* path;

assert(p1.x1()==p1.x2());
assert(p2.x1()==p2.x2());
assert(p1.x1()==p2.x1());
assert(p1.layer()==p2.layer());
x = p1.x1();
layer = p1.layer();
width = p1.Width();
pin1=pin2=pin3=pin4=false;

y1 = ::min( p1.bottom(), p2.bottom() );
x1 = x;
y2 = ::max( p1.bottom(), p2.bottom() );
x2 = x;
path = new CCctPath(layer,width,x1,y1,x2,y2);
if( y1==p1.y1() )
{path->pin1( p1.pin1() );pin1=true;}
else if ( y1==p1.y2() )
{path->pin1( p1.pin2() );pin2=true;}
else if ( y1==p2.y1() )
{path->pin1( p2.pin1() );pin3=true;}
else if ( y1==p2.y2() )
{path->pin1( p2.pin2() );pin4=true;}
if( y2==p1.y1() && !pin1)
{path->pin2( p1.pin1()
);pin1=true;pin2=pin3=pin4=false;}
else if ( y2==p1.y2() && !pin2)
{path->pin2( p1.pin2()
);pin2=true;pin1=pin3=pin4=false;}
else if ( y2==p2.y1() && !pin3)
{path->pin2( p2.pin1()
);pin3=true;pin1=pin2=pin4=false;}
else if ( y2==p2.y2() && !pin4)
{path->pin2( p2.pin2()
);pin4=true;pin1=pin2=pin3=false;}
if(path->pin1()!="" &&
path->pin2()!="" &&
path->pin1()==path->pin2()) path->pin2("");
paths.append( path );

pin = path->pin2();
y1 = y2;
x1 = x;
y2 = ::min( p1.top(), p2.top() );
x2 = x;
path = new CCctPath(layer,width,x1,y1,x2,y2);
path->pin1(pin);
if( y2==p1.y1() && !pin1)
{path->pin2( p1.pin1()
);pin1=true;pin2=pin3=pin4=false;}
else if ( y2==p1.y2() && !pin2)
{path->pin2( p1.pin2()
);pin2=true;pin1=pin3=pin4=false;}
else if ( y2==p2.y1() && !pin3)
{path->pin2( p2.pin1()
);pin3=true;pin1=pin2=pin4=false;}
else if ( y2==p2.y2() && !pin4)
{path->pin2( p2.pin2()
);pin4=true;pin1=pin2=pin3=false;}
if(path->pin1()!="" &&
path->pin2()!="" &&
path->pin1()==path->pin2()) path->pin2("");
paths.append( path );

pin = path->pin2();
y1 = y2;
x1 = x;
y2 = ::max( p1.top(), p2.top() );
x2 = x;
path = new CCctPath(layer,width,x1,y1,x2,y2);
path->pin1(pin);
if( y2==p1.y1() && !pin1) path->pin2( p1.pin1() );
else if ( y2==p1.y2() && !pin2) path->pin2( p1.pin2() );
else if ( y2==p2.y1() && !pin3) path->pin2( p2.pin1() );
else if ( y2==p2.y2() && !pin4) path->pin2( p2.pin2() );
if(path->pin1()!="" &&
 path->pin2()!="" &&
path->pin1()==path->pin2()) path->pin2("");
paths.append( path );
}else std::cout << "ERROR on CcctPath::nonInPaths()\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
return paths;
}
#include "ccctpin.h"

CCctPin::CCctPin(const QString name)
:m_name(name)
{
}
CCctPin::~CCctPin()
{
m_ports.clear();
}
#include "ccctwire.h"
#include "ccctpath.h"

CCctWire::CCctWire()
{
m_paths.setAutoDelete(true);
}

CCctWire::~CCctWire()
{
m_paths.clear();
}

void CCctWire::pins(const QString pin1,const QString pin2)
{
m_pin1=pin1,m_pin2=pin2;
m_paths.getFirst()->pin1(pin1);
m_paths.getLast()->pin2(pin2);
}


#include "global.h"
#include "ccomplex.h"

/****************************************************************************
 * operations on class complex
 ***************************************************************************/


const complex & complex::operator=(const complex & value)
{
  if (this != & value) {
    realp = value.realp;
    imagp = value.imagp;
  }
  return *this;
}


const complex & complex::operator+=(const complex & value)
{
  realp += value.realp;
  imagp += value.imagp;
  return *this;
}
```

```
complex operator * (const complex & A, const complex & B)
{
  return complex (A.realp*B.realp - A.imagp*B.imagp,
                  A.realp*B.imagp + A.imagp*B.realp);
}


complex operator / (const complex & A, const complex & B)
{
  double modu = B.realp*B.realp + B.imagp*B.imagp;
  complex tmp(B.realp/modu, -B.imagp/modu);
  return A*tmp;
}


complex exp (const complex & A)
{
  return complex (exp(A.realp)*cos(A.imagp), exp(A.realp)*sin(A.imagp));
}


complex pow (const complex & A, int n)
{
  complex tmp = A;
  for (int i = 1; i<n; i++) tmp = tmp*A;
  return tmp;
}


double arg (const complex & A)
{
  double ang;

  if (A.realp > 0) ang = atan(A.imagp/A.realp);
  else if (A.realp < 0) {
    if (A.imagp >= 0) ang = atan(A.imagp/A.realp) + PI;
    else ang = atan(A.imagp/A.realp) - PI;
  }
  else {
    if (A.imagp > 0) ang = PI/2;
    else if (A.imagp < 0) ang = -PI/2;
    else ang = 0;
  }

  return ang;
}


complex sqrt (const complex & A)
{
  double modu = sqrt(fabs(A));
  double ang = arg(A)/2.;
  return complex(modu*cos(ang), modu*sin(ang));
}


complex cbrt (const complex & A)
{
  double p = pow(fabs(A), 1./3.);
```

```
  double ang = arg(A)/3.;
  return complex(p*cos(ang), p*sin(ang));
}


void swap(complex &x, complex &y)
{
  complex tmp;
  tmp=x;  x=y;  y=tmp;
}


/****************************************************************************
 *  solve a set of linear equations through Gauss elimination
 *  form: sys * stat = excite
 *  vector stat composed by the variables to be solved
 *  this is used for AWE, thus size is restricted by the max order
 ***************************************************************************/

void gauss(int odr,
complex sys[MAX_ORDER+1][MAX_ORDER+1],
complex stat[MAX_ORDER+1],
complex excit[MAX_ORDER+1])
{
  int i, j, k, max_pos;
  complex ctmp;
  double tmp;

  for (i = 1; i < odr; i++) {
    tmp = fabs(sys[i][i]);
    max_pos = i;

    for (j = i+1; j <= odr; j++) {
      if (fabs(sys[j][i]) > tmp) {
tmp = fabs(sys[j][i]);
max_pos = j;
      }
    }

    //make the pivot row with large diagonal value
    //in order to avoid large error

    for (j = 1; j <= odr; j++)
      swap(sys[i][j], sys[max_pos][j]);
    swap(excit[i], excit[max_pos]);

    for (j = i+1; j <= odr; j++) {
      ctmp = sys[j][i]/sys[i][i];
      for (k = i; k <= odr; k++)
sys[j][k] = sys[j][k] - ctmp*sys[i][k];
      excit[j] = excit[j] - ctmp*excit[i];
    }
  }

  stat[odr] = excit[odr]/sys[odr][odr];
  for (i = odr-1; i>0; i--) {
    ctmp = 0;
    for (j = i+1; j <= odr; j++)
      ctmp += sys[i][j]*stat[j];
    stat[i] = (excit[i]-ctmp)/sys[i][i];
  }
}


/****************************************************************************
 * function solving cubic equation,
 * input: array of size 3, cr[0] is coefficient for 0 order term
 * output: one of 3 roots
 * refer: College Algebra by M. Richardson 3rd edition, p367
 ***************************************************************************/

bool cubic(complex *cr)
{
  complex p, q, m[3];

  p = cr[1] - cr[2]*cr[2]/3;
  q = cr[0] - cr[1]*cr[2]/3 + 2*cr[2]*cr[2]*cr[2]/27;

  m[0] = cbrt(-q/2 + sqrt(q*q/4 + p*p*p/27));
  m[1] = m[0]*complex(-0.5, sqrt(3)/2);
  m[2] = m[1]*complex(-0.5, sqrt(3)/2);

  if (fabs(m[0]) < TINY*TINY ||
      fabs(m[1]) < TINY*TINY ||
      fabs(m[2]) < TINY*TINY)
    return false;

  q = cr[2];
  cr[0] = m[0] - (p/m[0] + q)/3;
  cr[1] = m[1] - (p/m[1] + q)/3;
  cr[2] = m[2] - (p/m[2] + q)/3;

  for (int i = 0; i <= 2; i++)
    if (fabs(cr[i].get_imag()) < (TINY*TINY)) cr[i] = cr[i].get_real();

  return true;
}


/****************************************************************************
 * function solving quartic equation,
 * input: array of size 4, cr[0] is coefficient for 0 order term
 * output: in complex array cr, 4 roots
 * refer: College Algebra by M. Richardson 3rd edition, p369
 ***************************************************************************/

bool quartic(complex *cr)
{
  complex ce[3];

  complex sq(cr[3]*cr[3]);
  complex g(cr[2] - 3*sq/8);
  complex h(sq*cr[3]/8 - cr[3]*cr[2]/2 + cr[1]);
  complex k(sq*cr[2]/16 - cr[3]*cr[1]/4 + cr[0] - 3*sq*sq/256);

  ce[0] = 4*g*k - h*h;
  ce[1] = -4*k;
  ce[2] = -g;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  if (!cubic(ce))
    return false;

  complex z(ce[0]);

  if (fabs(z - g) < TINY*TINY)
    return false;

  sq = sqrt(z - g);
  h = h/sq;
  complex r(sqrt((z - g) - 2*(z - h)));
  complex t(sqrt((z - g) - 2*(z + h)));
  k = cr[3]/4;

  cr[1] = 0.5*(-sq + r) - k;
  cr[2] = 0.5*(-sq - r) - k;
  cr[3] = 0.5*(sq + t) - k;
  cr[4] = 0.5*(sq - t) - k;

  return true;
}

bool computePoles(complex deno[MAX_ORDER+1],
complex pole[MAX_ORDER+1], int odr)
{
  double delta;

  switch (odr) {
  case 4:
    if (!quartic(deno)) return false;
    for (int i = 1; i <= 4; i++)
      pole[i] = deno[i];
    break;

  case 3:
    if (!cubic(deno)) return false;
    pole[1] = deno[0];
    pole[2] = deno[1];
    pole[3] = deno[2];
    break;

  case 2:
    delta = deno[1].get_real()*deno[1].get_real() - 4*deno[0].get_real();
    if (delta >= 0) {
      pole[2] = (-deno[1].get_real() + sqrt(delta))/2;
      pole[1] = (-deno[1].get_real() - sqrt(delta))/2;
    }
    else {
      pole[2] = complex(-deno[1].get_real(), sqrt(-delta))/2;
      pole[1] = pole[2].get_conj();
    }
    break;

  case 1:
    pole[1] = -deno[0];
    break;
  }
```



```
  return true;
}

void polish(complex resid[MAX_ORDER+1],
complex pole[MAX_ORDER+1], int &odr, bool &reduce)
{
  reduce = false;

  for (int i = 1; i <= odr; i++) {
    if (fabs(resid[i].get_imag()) < TINY)
      resid[i] = resid[i].get_real();
    if (fabs(pole[i].get_imag()) < TINY)
      pole[i] = pole[i].get_real();

    if (pole[i].get_real() > TINY) {
// if positive pole exists
      if (fabs(resid[i]) < TINY) {
// if its residue trivial, negelect it
resid[i] = 0;
pole[i] = 0;
      }
      else {
// if positive pole is significant, reduce order
odr--;
reduce = true;
      }
    }
  }
}
```

```
#include "cdef.h"
#include "cdefcomp.h"
#include "cdefpin.h"
#include "cdefblock.h"
#include "cdefnet.h"
#include "cdefpath.h"

CDef::CDef()
:m_hasVersion(false)
,m_hasCaseSensitive(false)
,m_hasDivider(false)
,m_hasBusBit(false)
,m_hasDesign(false)
,m_hasDieArea(false)
,m_hasUnits(false)
,m_hasGCell(false)
{
m_comps.setAutoDelete( true );
m_nets.setAutoDelete( true );
m_pins.setAutoDelete( true );
m_blocks.setAutoDelete( true );
}

CDef::~CDef()
{
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
m_comps.clear();
m_nets.clear();
m_pins.clear();
m_blocks.clear();
}

bool CDef::load(const QString fileName)
{
FILE* defFile;
defrInit();
defrSetBlockageStartCbk( defrBlockageStartCbk );
defrSetBlockageCbk( defrBlockageCbk );
defrSetBlockageEndCbk( defrBlockageEndCbk );
defrSetBusBitCbk( defrBusBitCbk );
defrSetComponentStartCbk( defrComponentStartCbk );
defrSetComponentCbk( defrComponentCbk );
defrSetComponentEndCbk( defrComponentEndCbk );
defrSetPathCbk( defrPathCbk );
defrSetDesignCbk( defrDesignStartCbk );
defrSetDesignEndCbk( defrDesignEndCbk );
defrSetDieAreaCbk( defrDieAreaCbk );
defrSetDividerCbk(defrDividerCbk );
defrSetComponentExtCbk( defrComponentExtCbk );
defrSetGroupExtCbk( defrGroupExtCbk );
defrSetNetExtCbk( defrNetExtCbk );
defrSetNetConnectionExtCbk( defrNetConnectionExtCbk );
defrSetPinExtCbk( defrPinExtCbk );
defrSetScanChainExtCbk( defrScanChainExtCbk );
defrSetViaExtCbk( defrViaExtCbk );
defrSetFillStartCbk( defrFillStartCbk );
defrSetFillCbk( defrFillCbk );
defrSetFillEndCbk( defrFillEndCbk );
defrSetGcellGridCbk( defrGcellGridCbk );
defrSetGroupsStartCbk( defrGroupsStartCbk );
defrSetGroupNameCbk( defrGroupNameCbk );
defrSetGroupMemberCbk( defrGroupMemberCbk );
defrSetGroupCbk( defrGroupCbk );
defrSetGroupsEndCbk( defrGroupsEndCbk );
defrSetHistoryCbk( defrHistoryCbk );
defrSetCaseSensitiveCbk( defrCaseSensitiveCbk );
defrSetNetStartCbk( defrNetStartCbk );
defrSetNetCbk( defrNetCbk );
defrSetNetEndCbk( defrNetEndCbk );
defrSetStartPinsCbk( defrStartPinsCbk );
defrSetPinCbk( defrPinCbk );
  defrSetPinEndCbk( defrPinEndCbk );
defrSetPinPropStartCbk( defrPinPropStartCbk );
defrSetPinPropCbk( defrPinPropCbk );
defrSetPinPropEndCbk( defrPinPropEndCbk );
defrSetPropDefStartCbk( defrPropDefStartCbk );
defrSetPropCbk( defrPropCbk );
defrSetPropDefEndCbk( defrPropDefEndCbk );
defrSetRegionStartCbk( defrRegionStartCbk );
defrSetRegionCbk( defrRegionCbk );
defrSetRegionEndCbk( defrRegionEndCbk );
defrSetRowCbk( defrRowCbk );
defrSetScanchainsStartCbk( defrScanchainsStartCbk );
defrSetScanchainCbk( defrScanchainCbk );
defrSetScanchainsEndCbk( defrScanchainsEndCbk);
defrSetSlotStartCbk( defrSlotStartCbk );
defrSetSlotCbk( defrSlotCbk );
defrSetSlotEndCbk( defrSlotEndCbk );
defrSetSNetStartCbk( defrSNetStartCbk );
defrSetSNetCbk( defrSNetCbk );
defrSetSNetEndCbk( defrSNetEndCbk );
defrSetTechnologyCbk( defrTechNameCbk );
defrSetTrackCbk( defrTrackCbk );
defrSetUnitsCbk( defrUnitsCbk );
defrSetVersionCbk( defrVersionCbk );
defrSetVersionStrCbk( defrVersionStrCbk );
defrSetViaStartCbk( defrViaStartCbk );
defrSetViaCbk( defrViaCbk );
defrSetViaEndCbk( defrViaEndCbk );
defFile = fopen(fileName,"r");
if( defFile==NULL )
{
std::cout << "ERROR : couldn't open " << fileName << "\n";
return false;
}
  defrRead(defFile,fileName, (void*)this,1);

return true;
}

bool CDef::save(const QString fileName)
{
FILE* defFile;
defFile = fopen(fileName,"w");
if( defFile==NULL)
{
std::cout << "ERROR : couldn't open  " << fileName << "\n";
return false;
}
defwInitCbk(defFile);
defwSetBlockageCbk(defwBlockageCbk );
defwSetBusBitCbk( defwBusBitCbk );
defwSetCaseSensitiveCbk( defwCaseSensitiveCbk );
defwSetComponentCbk( defwComponentCbk );
defwSetDesignCbk( defwDesignCbk );
defwSetDesignEndCbk( defwDesignEndCbk );
defwSetDieAreaCbk( defwDieAreaCbk );
defwSetDividerCbk( defwDividerCbk );
defwSetExtCbk( defwExtCbk );
defwSetGcellGridCbk( defwGcellGridCbk );
defwSetGroupCbk( defwGroupCbk );
defwSetHistoryCbk( defwHistoryCbk );
defwSetNetCbk( defwNetCbk );
defwSetPinCbk(defwPinCbk );
defwSetPinPropCbk( defwPinPropCbk );
defwSetPropDefCbk( defwPropDefCbk );
defwSetRegionCbk(defwRegionCbk);
defwSetRowCbk( defwRowCbk );
defwSetSNetCbk( defwSNetCbk );
defwSetScanchainCbk( defwScanchainCbk );
defwSetTechnologyCbk( defwTechCbk );
defwSetTrackCbk( defwTrackCbk );
defwSetUnitsCbk( defwUnitsCbk );
defwSetVersionCbk( defwVersionCbk );
```

```
defwSetViaCbk( defwViaCbk );

defwWrite(defFile,fileName, (void*)this);

return true;
}
#include <defiBlockage.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefblock.h"

CDefBlock::CDefBlock()
:m_hasLayer(false)
,m_hasPlacement(false)
,m_hasComponent(false)
,m_hasSlots(false)
,m_hasFills(false)
,m_hasPushdown(false)
{

}

CDefBlock::CDefBlock(defiBlockage* blockage)
:m_hasLayer(false)
,m_hasPlacement(false)
,m_hasComponent(false)
,m_hasSlots(false)
,m_hasFills(false)
,m_hasPushdown(false)
{
defr(blockage);
}

CDefBlock::~CDefBlock()
{
m_rects.clear();
}

void CDefBlock::defr(defiBlockage* blockage)
{
int numRects;

if( blockage->hasLayer() )
{
m_hasLayer = true;
m_layerName = blockage->layerName();
}
if( blockage->hasPlacement() )
{
m_hasPlacement=true;
m_placementComponentName = blockage->placementComponentName();
}
if( blockage->hasComponent() )
{
m_hasComponent=true;
m_layerComponentName = blockage->layerComponentName();
}
if( blockage->hasSlots() ) m_hasSlots=true;
if( blockage->hasFills() ) m_hasFills=true;
if( blockage->hasPushdown() ) m_hasPushdown=true;
numRects = blockage->numRectangles();
for(int i=0;i<numRects;i++)
m_rects.append( CRect(blockage->xl(i),blockage->yl(i),
blockage->xh(i)-blockage->xl(i),blockage->yh(i)-blockage->yl(i)) );
}

void CDefBlock::defw()
{
QValueListIterator<CRect> r;
if( hasLayer() && hasFills() )
defwBlockageLayerFills(m_layerName);
else if( hasLayer() && hasPushdown() )
defwBlockageLayerPushdown(m_layerName);
else if( hasLayer() && hasSlots() )
defwBlockageLayerSlots(m_layerName);
else if( hasLayer() && hasComponent() )
defwBlockageLayer( m_layerName, m_layerComponentName );
else if( hasLayer() )
defwBlockageLayer(m_layerName,0);
else if( hasPlacement() )
defwBlockagePlacementComponent(m_placementComponentName);
else if( hasPushdown() )
defwBlockagePlacementPushdown();
for(r=m_rects.begin();r!=m_rects.end();r++)
defwBlockageRect( int((*r).left()),
int((*r).bottom()),int((*r).right()),int((*r).top()) );
}
#include <defiComponent.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefcomp.h"

CDefComp::CDefComp(const QString id, const QString name)
:m_id(id)
,m_name(name)
{
}

CDefComp::CDefComp(defiComponent* comp)
{
defr( comp );
}

CDefComp::~CDefComp()
{
}

void CDefComp::defr(defiComponent* comp)
{
IdAndName( comp->id(), comp->name() );
setPlacementStatus(::placementStatus( comp->placementStatus() ) );
setPlacementLocation( comp->placementX(), comp->placementY(),
::placementOrient( comp->placementOrient() ) );
}

void CDefComp::defw() const
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
defwComponent(id(),name(),
0,0,0,0,0,0,0,0,0,0,0,
::placementStatus( placementStatus() ),
(int)placementX(),(int)placementY(),
::placementOrient( placementOrient() ),
0,0,0,0,0,0);
}
#include "cdefgcell.h"
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include <defiRowTrack.hpp>

CDefGCell::CDefGCell()
{
}
CDefGCell::~CDefGCell()
{
}
void CDefGCell::defr(defiGcellGrid* defiGcellGrid)
{
if(QString(defiGcellGrid->macro())=="X")
{
        m_x = defiGcellGrid->x();
m_numCols = defiGcellGrid->xNum();
m_xStep = defiGcellGrid->xStep();
}else if(QString(defiGcellGrid->macro())=="Y")
{
        m_y = defiGcellGrid->x();
m_numRows = defiGcellGrid->xNum();
m_yStep = defiGcellGrid->xStep();
}
}
void CDefGCell::defw()
{
defwGcellGrid("X",m_x,m_numCols,(int)m_xStep);
defwGcellGrid("Y",m_y,m_numRows,(int)m_yStep);
}
#include <defiNet.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefnet.h"
#include "cnetpin.h"
#include "cdefwire.h"
#include "cdefpath.h"

CDefNet::CDefNet(const QString name)
:m_name(name)
,m_hasUse(false)
{
m_pins.setAutoDelete(true);
m_paths.setAutoDelete(true);
m_wires.setAutoDelete(true);
}

CDefNet::CDefNet(defiNet* net, QPtrList<CDefPath> paths)
:m_hasUse(false)
{
m_pins.setAutoDelete(true);
m_paths.setAutoDelete(true);
m_wires.setAutoDelete(true);
defr(net);
QPtrListIterator<CDefPath> it(paths);
for(it.toFirst();it.current();++it) addPath( *it );
}

CDefNet::~CDefNet()
{
m_pins.clear();
m_paths.clear();
m_wires.clear();
}

void CDefNet::defr(defiNet* net)
{
int numConnections = net->numConnections();
int numPaths = net->numPaths();
int numWires = net->numWires();
QString pin;
// std::cout << net->name() << " : " << numPaths << " _ " << numWires <<
"\n";

setName( net->name() );
if( net->hasUse() ) setUse( ::use(net->use()) );
for(int i=0;i<numConnections;i++)
{
pin = QString(net->instance(i)) + ":" + QString(net->pin(i));
m_pins.insert( pin, new CNetPin( net->instance(i),
net->pin(i), net->pinIsSynthesized(i) ) );
}
if( net->isFixed() ) m_netStat = "FIXED";
else if( net->isCover() ) m_netStat = "COVER";
else if( net->isRouted() ) m_netStat = "ROUTED";
m_paths.resize(numPaths);
for(int i=0;i<numPaths;i++) m_paths.insert(i,new CDefPath(net->path(i))
);
m_wires.resize(numWires);
for(int i=0;i<numWires;i++) m_wires.insert(i, new CDefWire(net->wire(i))
);
}

void CDefNet::defw() const
{
QDictIterator<CNetPin> pinIt(m_pins);
int numPaths = m_paths.size();
int numWires = m_wires.size();

defwNet(name());
for(pinIt.toFirst();pinIt.current();++pinIt) (*pinIt)->defw();
if( m_hasUse ) defwNetUse( ::use(m_use) );
if( numPaths>0 )
{
defwNetPathStart("NEW");
for(int i=0;i<numPaths;i++) m_paths[i]->defw();
defwNetPathEnd();
}
if( numWires>0 )
{
for(int i=0;i<numWires;i++) m_wires[i]->defw();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
defwNetEndOneNet();
}

// add a pin
void CDefNet::addPin(const QString instance, const QString name)
{
 m_pins.insert(instance+":"+name, new CNetPin( instance, name, 0 ) );
}

void CDefNet::addPath(CDefPath* path)
{
// std::cout << "adding prev path to " << m_name << "\n";
int size = m_paths.size();
m_paths.resize( size+1 );
m_paths.insert( size, path );
}
#include <defiPath.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefpath.h"

CDefPath::CDefPath(defiPath* path)
{
defr(path);
}

CDefPath::~CDefPath()
{
while(!m_path.isEmpty())
{
switch( m_path.front().first )
{
case PATHLAYER : delete (QString*)m_path.front().second;break;
case PATHVIA : delete (QString*)m_path.front().second;break;
case PATHPOINT : delete (CPoint*)m_path.front().second;break;
case PATHWIDTH : delete (int*)m_path.front().second;break;
case PATHFLUSHPOINT : delete
(CPoint*)m_path.front().second;break;
default : std::cout << "CDefPath::~CDefPath() not implemented
case\n";
}
m_path.pop_front();
}
m_path.clear();
}

void CDefPath::defr(defiPath* path)
{
QPair< pathType, void*> defPath;
pathType pathType;
int x,y,e;

  path->initTraverse();
  while((pathType = ::pathType(path->next())) != PATHDONE)
{
  switch (pathType)
{
  case PATHLAYER:
defPath.first = pathType;
defPath.second = new QString(path->getLayer());
m_path.append( defPath );
// std::cout << " : " << ::pathStr(pathType)
<< " " << QString(path->getLayer());
break;
case PATHVIA:
defPath.first = pathType;
defPath.second = new QString( path->getVia() );
m_path.append( defPath );
// std::cout << " : " << ::pathStr(pathType)
<< " " << QString(path->getVia());
break;
  case PATHPOINT:
defPath.first = pathType;
path->getPoint(&x,&y);
defPath.second = new CPoint(x,y);
m_path.append( defPath );
// std::cout << " : " << ::pathStr(pathType)
<< " (" << x << "," <<  y << ")";
break;
  case PATHFLUSHPOINT:
defPath.first = pathType;
path->getFlushPoint(&x,&y,&e);
defPath.second = new CPoint(x,y);
m_path.append( defPath );
// std::cout << " : " << ::pathStr(pathType)
<< " (" << x << "," <<  y << ")";
break;
  case PATHWIDTH:
defPath.first = pathType;
defPath.second = new int( path->getWidth() );
m_path.append( defPath );
// std::cout << " : " << ::pathStr(pathType)
<< " (" << path->getWidth() << ")";
break;
default: std::cout << " CDefPath::defr() not implemented
case\n";
}
}
}

void CDefPath::defw() const
{
int numPoints;
QValueListConstIterator< QPair<pathType, void*> > pathIt;
CPoint point;
QValueList<CPoint> points;

for(pathIt=m_path.begin();pathIt!=m_path.end();pathIt++)
{
switch((*pathIt).first)
{
case PATHLAYER:
if( !points.isEmpty() )
{
numPoints = points.count();
const char* pointX[1000];//numPoints];
const char* pointY[1000];//numPoints];
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int c = 0;
while( !points.isEmpty() )
{
point = points.front();
points.pop_front();
pointX[c] =
QString().setNum(point.x()).latin1();
pointY[c] =
QString().setNum(point.y()).latin1();
c++;
}
defwNetPathPoint(numPoints,pointX,pointY,NULL);
}
defwNetPathLayer( *(QString*)(*pathIt).second,0,"");
break;
case PATHVIA:
if( !points.isEmpty() )
{
numPoints = points.count();
const char* pointX[1000];//numPoints];
const char* pointY[1000];//numPoints];
int c = 0;
while( !points.isEmpty() )
{
point = points.front();
points.pop_front();
pointX[c] =
QString().setNum(point.x());
pointY[c] = QString().setNum(point.y());
c++;
}
defwNetPathPoint(numPoints,pointX,pointY,NULL);
}
defwNetPathVia( *(QString*)(*pathIt).second );
break;
case PATHPOINT:
points << *(CPoint*)(*pathIt).second;
break;
default: std::cout << "CDefPath::defw() not implemented case\n";
}
}
if( !points.isEmpty() )
{
numPoints = points.count();
const char* pointX[1000];//numPoints];
const char* pointY[1000];//numPoints];
int c = 0;
while( !points.isEmpty() )
{
point = points.front();
points.pop_front();
pointX[c] = QString().setNum(point.x());
pointY[c] = QString().setNum(point.y());
c++;
}
defwNetPathPoint(numPoints,pointX,pointY,NULL);
}
}
#include "cdefpin.h"
```

```
#include <defiPinCap.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>

CDefPin::CDefPin(defiPin* pin)
:m_hasLayer(false)
,m_hasPlacement(false)
,m_hasUse(false)
,m_hasDirection(false)
{
defr(pin);
}

CDefPin::~CDefPin()
{
}

void CDefPin::defr(defiPin* pin)
{
int xl,yl,xh,yh;
placementStatusType plaStatus;

Setup( pin->pinName(), pin->netName() );
if( pin->hasDirection() ) setDirection( ::pinDirection( pin->direction()
) );
if( pin->hasUse() ) setUse( ::use( pin->use() ) );
if( pin->hasLayer() ) { pin->bounds(&xl,&yl,&xh,&yh);
setLayer( pin->layer(),xl,yl,xh,yh);}
if( pin->hasPlacement() )
{
plaStatus = UNKNOWNPLACEMENTSTATUS;
if( pin->isUnplaced() ) plaStatus = UNPLACED;
if( pin->isPlaced() ) plaStatus = PLACED;
if( pin->isCover() ) plaStatus = COVER;
if( pin->isFixed() ) plaStatus = FIXED;
setPlacement(plaStatus, pin->placementX(), pin->placementY(),
::placementOrient( pin->orient()) );
}

}

void CDefPin::defw() const
{
QString dir;
QString uses;
placementStatusType placementStatus;
placementOrientType placementOrient;
int x,y;
double xl,yl,xh,yh;
QString layers;

    xl=yl=xh=yh=0;
   if(hasDirection()) dir = ::pinDirection(direction());
if(hasUse()) uses = ::use(use());
if( hasLayer() ) layers =layer();
if( hasLayer() ) bounds( &xl,&yl,&xh,&yh);
if( hasPlacement() )
{
placementStatus = UNKNOWNPLACEMENTSTATUS;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if( isUnplaced() ) placementStatus = UNPLACED;
if( isPlaced() ) placementStatus = PLACED;
if( isCover() ) placementStatus = COVER;
if( isFixed() ) placementStatus = FIXED;
x = (int)placementX();
y = (int)placementY();
placementOrient = orient();
defwPin( pinName(), netName(),0,dir,uses,
::placementStatus(placementStatus),x,y,
::placementOrient(placementOrient),
layers,(int)xl,(int)yl,(int)xh,(int)yh);

}
else defwPin( pinName(), netName(),0,dir,uses,
0,0,0,0,
layers,(int)xl,(int)yl,(int)xh,(int)yh);
}
#include "cdef.h"
#include "cdefcomp.h"
#include "cdefpin.h"
#include "cdefblock.h"
#include "cdefnet.h"
#include "cdefpath.h"

int defrBlockageStartCbk( defrCallbackType_e type,
int numBlockages, defiUserData data )
{
assert(type==defrBlockageStartCbkType);
std::cout << "defrBlockageStartCbk " << numBlockages << data << "\n";
return 0;
}

int defrBlockageCbk( defrCallbackType_e type,
defiBlockage* blockage, defiUserData data )
{
assert(type==defrBlockageCbkType);
CDef* def = (CDef*)data;

def->m_blocks.append( new CDefBlock(blockage) );
return 0;
}

int defrBlockageEndCbk( defrCallbackType_e type,
void* ptr, defiUserData data )
{
assert(type==defrBlockageEndCbkType);
std::cout << "defrBlockageEndCbk " << ptr << data << "\n";
return 0;
}

int defrBusBitCbk( defrCallbackType_e type,
const char* busBit, defiUserData data )
{
assert(type==defrBusBitCbkType);
CDef* def = (CDef*)data;

def->busBit( busBit );
return 0;
}

int defrComponentStartCbk( defrCallbackType_e type,
int numComps, defiUserData data )
{
assert(type==defrComponentStartCbkType);
std::cout << "defrComponentStartCbk " << numComps << data << "\n";
return 0;
}

int defrComponentCbk( defrCallbackType_e type,
defiComponent* comp, defiUserData data )
{
assert(type==defrComponentCbkType);
CDef* def = (CDef*)data;

def->m_comps.insert( comp->id(), new CDefComp( comp ) );
return 0;
}

int defrComponentEndCbk( defrCallbackType_e type,void* ptr,
 defiUserData data )

{
assert(type==defrComponentEndCbkType);
std::cout << "defrComponentEndCbk " << ptr << data << "\n";
return 0;
}

int defrPathCbk( defrCallbackType_e type, defiPath* path,
 defiUserData data )
{
assert(type==defrPathCbkType);
CDef* def = (CDef*)data;

def->m_paths.append( new CDefPath( path ) );
return 0;
}

int defrDesignStartCbk( defrCallbackType_e type, const char* design,
 defiUserData data )
{
assert(type==defrDesignStartCbkType);
CDef* def = (CDef*)data;

def->design( design );
return 0;
}

int defrDesignEndCbk( defrCallbackType_e type, void* ptr,
 defiUserData data )
{
assert(type==defrDesignEndCbkType);
std::cout << "defrDesignEndCbk " << ptr << data << "\n";
return 0;
}

int defrDieAreaCbk( defrCallbackType_e type, defiBox* box,
 defiUserData data )
{
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
assert(type==defrDieAreaCbkType);
CDef* def = (CDef*)data;

def->dieArea( box->xl(), box->yl(), box->xh(), box->yh() );
return 0;
}

int defrDividerCbk( defrCallbackType_e type, const char* divider,
defiUserData data )

{
assert(type==defrDividerCbkType);
CDef* def = (CDef*)data;

def->divider( divider );
return 0;
}

int defrComponentExtCbk( defrCallbackType_e type,  const char* compExt,
 defiUserData data )

{
assert(type==defrComponentExtCbkType);
std::cout << "defrComponentExtCbk " << compExt << data << "\n";
return 0;
}
int defrGroupExtCbk( defrCallbackType_e type, const char* groupExt,
defiUserData data )

{
assert(type==defrGroupExtCbkType);
std::cout << "defrGroupExtCbk " << groupExt << data << "\n";
return 0;
}
int defrNetExtCbk( defrCallbackType_e type, const char* netExt,
 defiUserData data )

{
assert(type==defrNetExtCbkType);
std::cout << "defrNetExtCbk " << netExt << data << "\n";
return 0;
}
int defrNetConnectionExtCbk( defrCallbackType_e type, const char* netConnExt,
 defiUserData data )
{
assert(type==defrNetConnectionExtCbkType);
std::cout << "defrNetConnectionExtCbk " << netConnExt << data << "\n";
return 0;
}

int defrPinExtCbk( defrCallbackType_e type, const char* pinExt,
defiUserData data )
{
assert(type==defrPinExtCbkType);
std::cout << "defrPinExtCbk " << pinExt << data << "\n";
return 0;
}

int defrScanChainExtCbk( defrCallbackType_e type, const char* scanChainExt,
 defiUserData data )
{
assert(type==defrScanChainExtCbkType);
std::cout << "defrScanChainExtCbk " << scanChainExt << data << "\n";
return 0;
}

int defrViaExtCbk( defrCallbackType_e type, const char* viaExt,
defiUserData data )
{
assert(type==defrViaExtCbkType);
std::cout << "defrViaExtCbk " << viaExt << data << "\n";
return 0;
}

int defrFillStartCbk( defrCallbackType_e type, int numFills,
defiUserData data )
{

assert(type==defrFillStartCbkType);
std::cout << "defrFillStartCbk " << numFills << data << "\n";
return 0;
}

int defrFillCbk( defrCallbackType_e type, defiFill* fill,
defiUserData data )
{
assert(type==defrFillCbkType);
std::cout << "defrFillCbk " << fill << data << "\n";
return 0;
}

int defrFillEndCbk( defrCallbackType_e type, void* ptr,
defiUserData data )
{
assert(type==defrFillEndCbkType);
std::cout << "defrFillEndCbk " << ptr << data << "\n";
return 0;
}
int defrGcellGridCbk( defrCallbackType_e type, defiGcellGrid* grid,
 defiUserData data )
{
assert(type==defrGcellGridCbkType);
CDef* def = (CDef*)data;

def->m_gCell.defr(grid);
def->m_hasGCell = true;
return 0;
}

int defrGroupsStartCbk( defrCallbackType_e type, int numGroups,
defiUserData data )
{
assert(type==defrGroupsStartCbkType);
std::cout << "defrGroupsStartCbk " << numGroups << data << "\n";
return 0;
}

int defrGroupNameCbk( defrCallbackType_e type, const char* group,
```

```
defiUserData data )
{
assert(type==defrGroupNameCbkType);
std::cout << "defrGroupNameCbk " << group << data << "\n";
return 0;
}

int defrGroupMemberCbk( defrCallbackType_e type, const char* groupMember,
defiUserData data )
{
assert(type==defrGroupMemberCbkType);
std::cout << "defrGroupMemberCbk " << groupMember << data << "\n";
return 0;
}

int defrGroupCbk( defrCallbackType_e type, defiGroup* group,
defiUserData data )
{
assert(type==defrGroupCbkType);
std::cout << "defrGroupCbk " << group << data << "\n";
return 0;
}

int defrGroupsEndCbk( defrCallbackType_e type, void* ptr,
defiUserData data )
{
assert(type==defrGroupsEndCbkType);
std::cout << "defrGroupsEndCbk " << ptr << data << "\n";
return 0;
}

int defrHistoryCbk( defrCallbackType_e type, const char* history,
defiUserData data )
{
assert(type==defrHistoryCbkType);
std::cout << "defrHistoryCbk " << history << data << "\n";
return 0;
}

int defrCaseSensitiveCbk( defrCallbackType_e type, int caseSensitive,
defiUserData data )
{
assert(type==defrCaseSensitiveCbkType);
CDef* def = (CDef*)data;

def->caseSensitive( caseSensitive==1 );
return 0;
}

int defrNetStartCbk( defrCallbackType_e type, int numNets,
defiUserData data )
{
assert(type==defrNetStartCbkType);
// std::cout << "defrNetStartCbk " << numNets << data << "\n";
return 0;
}

int defrNetCbk( defrCallbackType_e type, defiNet* net,
defiUserData data )
{
assert(type==defrNetCbkType);
CDef* def = (CDef*)data;

// std::cout << "NET : " << net->name() << "\n";
def->m_nets.insert( net->name(), new CDefNet( net,
def->m_paths ) );
def->m_paths.clear();
return 0;
}

int defrNetEndCbk( defrCallbackType_e type, void* ptr,
defiUserData data )
{
assert(type==defrNetEndCbkType);
// std::cout << "defrNetEndCbk " << ptr << data << "\n";
return 0;
}

int defrStartPinsCbk( defrCallbackType_e type,int numPins,
 defiUserData data )
{
assert(type==defrStartPinsCbkType);
// std::cout << "defrStartPinsCbk " << numPins << data << "\n";
return 0;
}

int defrPinCbk( defrCallbackType_e type, defiPin* pin,
defiUserData data )

{
assert(type==defrPinCbkType);
CDef* def = (CDef*)data;

def->m_pins.insert( pin->pinName(), new CDefPin( pin ) );
return 0;
}

int defrPinEndCbk( defrCallbackType_e type, void* ptr,
defiUserData data )
{
assert(type==defrPinEndCbkType);
// std::cout << "defrPinEndCbk " << ptr << data << "\n";
return 0;
}

int defrPinPropStartCbk( defrCallbackType_e type, int numPinProps,
defiUserData data )
{
assert(type==defrPinPropStartCbkType);
std::cout << "defrPinPropStartCbk " << numPinProps << data << "\n";
return 0;
}

int defrPinPropCbk( defrCallbackType_e type, defiPinProp* pinProp,
defiUserData data )
{
assert(type==defrPinPropCbkType);
std::cout << "defrPinPropCbk " << pinProp << data << "\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
return 0;
}

int defrPinPropEndCbk( defrCallbackType_e type, void* ptr,
defiUserData data )
{
assert(type==defrPinPropEndCbkType);
std::cout << "defrPinPropEndCbk " << ptr << data << "\n";
return 0;
}

int defrPropDefStartCbk( defrCallbackType_e type, void* ptr,
 defiUserData data )
{
assert(type==defrPropDefStartCbkType);
std::cout << "defrPropDefStartCbk " << ptr << data << "\n";
return 0;
}

int defrPropCbk( defrCallbackType_e type,defiProp* prop,
defiUserData data )
{
assert(type==defrPropCbkType);
std::cout << "defrPropCbk " << prop << data << "\n";
return 0;
}

int defrPropDefEndCbk( defrCallbackType_e type,void* ptr,
defiUserData data )
{
assert(type==defrPropDefEndCbkType);
std::cout << "defrPropDefEndCbk " << ptr << data << "\n";
return 0;
}

int defrRegionStartCbk( defrCallbackType_e type, int numRegions,
defiUserData data )
{
assert(type==defrRegionStartCbkType);
std::cout << "defrRegionStartCbk " << numRegions << data << "\n";
return 0;
}

int defrRegionCbk( defrCallbackType_e type, defiRegion* region,
defiUserData data )
{
assert(type==defrRegionCbkType);
std::cout << "defrRegionCbk " << region << data << "\n";
return 0;
}

int defrRegionEndCbk( defrCallbackType_e type,void* ptr,
defiUserData data )
{
assert(type==defrRegionEndCbkType);
std::cout << "defrRegionEndCbk " << ptr << data << "\n";
return 0;
}

int defrRowCbk( defrCallbackType_e type, defiRow* row,
defiUserData data )
{
assert(type==defrRowCbkType);
std::cout << "defrRowCbk " << row << data << "\n";
return 0;
}

int defrScanchainsStartCbk( defrCallbackType_e type, int numScanChains,
 defiUserData data )
{
assert(type==defrScanchainsStartCbkType);
std::cout << "defrScanchainsStartCbk " << numScanChains << data << "\n";
return 0;
}

int defrScanchainCbk( defrCallbackType_e type, defiScanchain* scanChain,
defiUserData data )

{
assert(type==defrScanchainCbkType);
std::cout << "defrScanchainCbk " << scanChain << data << "\n";
return 0;
}
int defrScanchainsEndCbk( defrCallbackType_e type,void* ptr,
 defiUserData data)

{
assert(type==defrScanchainsEndCbkType);
std::cout << "defrScanchainsEndCbk " << ptr << data << "\n";
return 0;
}
int defrSlotStartCbk( defrCallbackType_e type, int numSlots,
defiUserData data )

{
assert(type==defrSlotStartCbkType);
std::cout << "defrSlotStartCbk " << numSlots << data << "\n";
return 0;
}
int defrSlotCbk( defrCallbackType_e type, defiSlot* slot,
 defiUserData data )

{
assert(type==defrSlotCbkType);
std::cout << "defrSlotCbk " << slot << data << "\n";
return 0;
}
int defrSlotEndCbk( defrCallbackType_e type, void*  ptr,
defiUserData data )
{
assert(type==defrSlotEndCbkType);
std::cout << "defrSlotEndCbk " << ptr << data << "\n";
return 0;
}

int defrSNetStartCbk( defrCallbackType_e type,int numSNets,
defiUserData data )
{
```

```
assert(type==defrSNetStartCbkType);
// std::cout << "defrSNetStartCbk " << numSNets << data << "\n";
return 0;
}

int defrSNetCbk( defrCallbackType_e type, defiNet* sNet,
defiUserData data )
{
assert(type==defrSNetCbkType);
CDef* def = (CDef*)data;

// std::cout << "SNETCBK " << sNet->name() << "\n";
def->m_snets.insert( sNet->name(),  new CDefNet( sNet,
def->m_paths ) );
def->m_paths.clear();
return 0;
}

int defrSNetEndCbk( defrCallbackType_e type,void* ptr,
 defiUserData data )
{
assert(type==defrSNetEndCbkType);
// std::cout << "defrSNetEndCbk " << data << "\n";
return 0;
}

int defrTechNameCbk( defrCallbackType_e type,const char* tech,
 defiUserData data )
{
assert(type==defrTechNameCbkType);
std::cout << "defrTechNameCbk " << tech << data << "\n";
return 0;
}

int defrTrackCbk( defrCallbackType_e type, defiTrack* track,
defiUserData data )

{
assert(type==defrTrackCbkType);
std::cout << "defrTrackCbk " << track << data << "\n";
return 0;
}
int defrUnitsCbk( defrCallbackType_e type,double units,
 defiUserData data )
{
assert(type==defrUnitsCbkType);
CDef* def = (CDef*)data;

def->units( units );
return 0;
}
int defrVersionCbk( defrCallbackType_e type,double version,
 defiUserData data )
{
assert(type==defrVersionCbkType);
CDef* def = (CDef*)data;

def->version( version );
return 0;
}

int defrVersionStrCbk( defrCallbackType_e type,const char* version,
defiUserData data )
{
assert(type==defrVersionStrCbkType);
CDef* def = (CDef*)data;

def->versionStr( version );
return 0;
}

int defrViaStartCbk( defrCallbackType_e type,int numVias,
 defiUserData data )

{
assert(type==defrViaStartCbkType);
std::cout << "defrViaStartCbk " << numVias << data << "\n";
return 0;
}
int defrViaCbk( defrCallbackType_e type,defiVia* via,
defiUserData data )

{
assert(type==defrViaCbkType);
std::cout << "defrViaCbk " << via << data << "\n";
return 0;
}
int defrViaEndCbk( defrCallbackType_e type,void* ptr,
defiUserData data )

{
assert(type==defrViaEndCbkType);
std::cout << "defrViaEndCbk " << ptr << data << "\n";
return 0;
}
#include "cdef.h"
#include "cdefcomp.h"
#include "cdefpin.h"
#include "cdefblock.h"
#include "cdefnet.h"

int defwBlockageCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwBlockageCbkType);
CDef* def = (CDef*)data;

if( def->m_blocks.isEmpty() ) return 0;
defwStartBlockages(def->m_blocks.count());
QPtrListIterator<CDefBlock> blockageIt(def->m_blocks);
for(blockageIt.toFirst();blockageIt.current();++blockageIt)
 (*blockageIt)->defw();
defwEndBlockages();
return 0;
}
int defwBusBitCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwBusBitCbkType);
CDef* def = (CDef*)data;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if( def->hasBusBit() ) defwBusBitChars( def->m_busBit );
return 0;
}

int defwCaseSensitiveCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwCaseSensitiveCbkType);
CDef* def = (CDef*)data;

if( def->hasCaseSensitive() )
{
if( def->m_caseSensitive ) defwCaseSensitive( "ON" );
else  defwCaseSensitive( "OFF" );
}
return 0;
}

int defwComponentCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwComponentCbkType);
CDef* def = (CDef*)data;

if( def->m_comps.isEmpty() ) return 0;
QDictIterator<CDefComp> compIt(def->m_comps);
defwStartComponents( compIt.count());
for(compIt.toFirst();compIt.current();++compIt) (*compIt)->defw();
defwEndComponents();
return 0;
}

int defwDesignCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwDesignCbkType);
CDef* def = (CDef*)data;

if( def->hasDesign() ) defwDesignName( def->m_designName );
return 0;
}

int defwDesignEndCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwDesignEndCbkType);
std::cout << "defwDesignEndCbk " << data << "\n";
defwEnd();
return 0;
}

int defwDieAreaCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwDieAreaCbkType);
CDef* def = (CDef*)data;

if( def->hasDieArea() )
defwDieArea(  (int)def->m_dieArea.left(),
(int)def->m_dieArea.bottom(),
(int)def->m_dieArea.right(), (int)def->m_dieArea.top() );
return 0;
}

int defwDividerCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwDividerCbkType);
CDef* def = (CDef*)data;

if( def->hasDivider() ) defwDividerChar( def->m_divider );
return 0;
}

int defwExtCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwExtCbkType);
std::cout << "defwExtCbk " << data << "\n";
return 0;
}

int defwGcellGridCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwGcellGridCbkType);
CDef* def = (CDef*)data;

if(def->hasGCell())
def->m_gCell.defw();
return 0;
}

int defwGroupCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwGroupCbkType);
std::cout << "defwGroupCbk " << data << "\n";
return 0;
}

int defwHistoryCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwHistoryCbkType);
std::cout << "defwHistoryCbk " << data << "\n";
return 0;
}

int defwNetCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwNetCbkType);
CDef* def = (CDef*)data;

if( def->m_nets.isEmpty() ) return 0;
   QDictIterator<CDefNet> netIt(def->m_nets);
defwStartNets( netIt.count() );
for(netIt.toFirst();netIt.current();++netIt) (*netIt)->defw();
defwEndNets();
return 0;
}

int defwPinCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwPinCbkType);
CDef* def = (CDef*)data;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if( def->m_pins.isEmpty() ) return 0;
QDictIterator<CDefPin> pinIt(def->m_pins);
defwStartPins( pinIt.count() );
for(pinIt.toFirst();pinIt.current();++pinIt) (*pinIt)->defw();
defwEndPins();
return 0;
}

int defwPinPropCbk(defwCallbackType_e type, defiUserData data )

{
assert(type==defwPinPropCbkType);
std::cout << "defwPinPropCbk " << data << "\n";
return 0;
}
int defwPropDefCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwPropDefCbkType);
std::cout << "defwPropDefCbk " << data << "\n";
return 0;
}

int defwRegionCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwRegionCbkType);
std::cout << "defwRegionCbk " << data << "\n";
return 0;
}
int defwRowCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwRowCbkType);
std::cout << "defwRowCbk " << data << "\n";
return 0;
}

int defwSNetCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwSNetCbkType);
std::cout << "defwSNetCbk " << data << "\n";
return 0;
}

int defwScanchainCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwScanchainCbkType);
std::cout << "defwScanchainCbk " << data << "\n";
return 0;
}

int defwTechCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwTechCbkType);
std::cout << "defwTechCbk " << data << "\n";
return 0;
}

int defwTrackCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwTrackCbkType);
std::cout << "defwTrackCbk " << data << "\n";
return 0;
}

int defwUnitsCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwUnitsCbkType);
CDef* def = (CDef*)data;

if( def->hasUnits() ) defwUnits( (int)def->m_units );
return 0;
}
int defwVersionCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwVersionCbkType);
CDef* def = (CDef*)data;

if( def->hasVersion() )
{
int ver1 = (int)floor(def->m_versionStr.toDouble());
int ver2 =
(int)(10.0*(def->m_versionStr.toDouble()-(double)ver1));
defwVersion( ver1,ver2 );
}
return 0;
}

int defwViaCbk(defwCallbackType_e type, defiUserData data )
{
assert(type==defwViaCbkType);
std::cout << "defwViaCbk " << data << "\n";
return 0;
}

#include <defiNet.hpp>
#include <defiPath.hpp>
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cdefwire.h"
#include "cdefpath.h"

CDefWire::CDefWire(defiWire* wire)
{
defr(wire);
}

CDefWire::~CDefWire()
{
m_paths.clear();
}

void CDefWire::defr(defiWire* wire)
{
m_wireType =  ::wire(wire->wireType());
int numPaths = wire->numPaths();
m_paths.resize(numPaths);
// std::cout << "numWirePaths : " << numPaths << "\n";
for(int i=0;i<numPaths;i++)
m_paths.insert(i,  new CDefPath( wire->path(i) ));
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}

void CDefWire::defw() const
{
int numPaths = m_paths.size();
if(numPaths>0)
{
defwNetPathStart(::wire(m_wireType));
for(int i=0;i<numPaths;i++) m_paths[i]->defw();
defwNetPathEnd();
}
}
#include "cdsn.h"

CDsn::CDsn()
{
}
CDsn::~CDsn()
{
m_wiring.clear();
}

bool CDsn::load(const QString fileName)
{
QFile dsnFile(fileName);
if( !dsnFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open " <<
fileName << "\n";
return false;
}
QTextStream dsn(&dsnFile);

QString layer,var;
double x1,y1,x2,y2;

while( !dsn.atEnd() )
{
dsn >> var;
var = var.replace( QRegExp("\\(|\\)"), "" );
if( var=="rect" )
{
dsn >> layer >> x1 >> y1 >> x2 >> y2;
m_wiring.append( QPair<QString,CRect>::QPair(layer,
CRect(x1,y1,x2-x1,y2-y1)));
}
}
dsnFile.close();
return true;
}

bool CDsn::save(const QString fileName)
{
std::cout << "CDsn::save() not implemented yet " << fileName << "\n";
return true;
}
#include "cedge.h"
#include "cnode.h"
#include "csource.h"
```

```
#include "csubroot.h"
#include "clayer.h"

CEdge::CEdge(CNode* up, CNode* down, const CLayerPair layer)
:m_up(up)
,m_down(down)
,m_layer(layer)
,m_res(INF)
{
}

CEdge::CEdge()
:m_up(NULL)
,m_down(NULL)
,m_res(INF)
{
}

CEdge::CEdge(const CEdge& e)
:m_up(e.m_up)
,m_down(e.m_down)
,m_layer(e.m_layer)
,m_res(e.m_res)
,m_current(e.m_current)
{
}

CEdge::~CEdge()
{
m_current.clear();
}

double CEdge::manhDist() const
{
return CPoint::manhDist(*m_down,*m_up);
}

double CEdge::xDist() const
{
return CPoint::xDist(*m_down,*m_up);
}

double CEdge::yDist() const
{
return CPoint::yDist(*m_down,*m_up);
}

void CEdge::disconnect() const
{
m_down->parent(NULL);
m_up->removeChild( m_down );
m_layer.vl()->decnUsage( yDist() );
m_layer.hl()->decnUsage( xDist() );
}

void CEdge::connect() const
{
m_down->parent( m_up );
m_up->addChild( m_down );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
m_layer.vl()->incnUsage( yDist() );
m_layer.hl()->incnUsage( xDist() );
m_down->edge( *this );
}

void CEdge::computeR()
{
if(m_down->type()==ROOT) m_res = ((CSource*)m_down)->outRes();
else if(m_down->type()==SUBROOT) m_res = ((CSubRoot*)m_down)->outRes();
else m_res = xDist() * m_layer.hl()->res() +
 yDist() * m_layer.vl()->res();
}

#include "cgroute.h"
#include "clef.h"
#include "clefvia.h"
#include "cleflayer.h"
#include "clefmacro.h"
#include "clefpin.h"
#include "clefport.h"
#include "clefobs.h"
#include "cdef.h"
#include "cdefblock.h"
#include "cdefpin.h"
#include "cdefcomp.h"
#include "cdefnet.h"
#include "cnetpin.h"
#include "cdefpath.h"
#include "cstr.h"
#include "ccctkeepout.h"
#include "ccctlayer.h"
#include "clib.h"
#include "ccctimage.h"
#include "ccctpin.h"
#include "cpla.h"
#include "ccctcomp.h"
#include "cnetlist.h"
#include "ccctnet.h"
#include "cdsn.h"
#include "ctim.h"
#include "ctiming.h"

#include "clayer.h"
#include "cnet.h"
#include "cnode.h"
#include "csource.h"
#include "csink.h"

#include "gwire.h"
#include "gpath.h"

CGRoute::CGRoute()
:m_initd(false)
,m_hasDieArea(false)
,m_hasGCell(false)
,m_hasGTile(false)
,m_hasOrder(false)
,m_hasLump(false)
,m_hasDriver(false)
,m_hasBuffer(false)
,m_hasLoad(false)
,m_hasBlocks(false)
{
m_layers.setAutoDelete(true);
m_nets.setAutoDelete(true);
}
CGRoute::~CGRoute()
{
m_nets.clear();
m_layers.clear();
m_cellBlocks.clear();
m_tileBlocks.clear();
m_unselectNets.clear();
}

bool CGRoute::init()
{
extern bool hasLefDef;
extern bool hasCct;
if(hasLefDef) {if(!initFormLefDef()) return false;}
else if( hasCct ) {if(!initFromCct()) return false;}
else{std::cout << "ERROR :    no data to groute\n";return false;}
m_initd = true;
return true;
}

bool CGRoute::initFormLefDef()
{
extern CLef lef;
extern CDef def;
extern CTim tim;

// init die area;
if(!m_hasDieArea)
{
if(!def.hasDieArea()){std::cout
<< "Error : no die area defined\n";return false;}
      m_dieArea = def.dieArea()/def.units();
m_hasDieArea = true;
std::cout << "setting die area (";
std::cout << m_dieArea.left() << " "
<< m_dieArea.bottom() << " : ";
std::cout << m_dieArea.right() << " "
<< m_dieArea.top() << ")\n";
}
//gcell
if(!m_hasGCell)
{
if(!def.hasGCell()){std::cout << "Error :
no gcell defined\n";return false;}
m_gCell.width( def.gCell().xStep()/def.units() );
m_gCell.height( def.gCell().yStep()/def.units() );
m_hasGCell = true;
std::cout << "setting gcell (";
std::cout << m_gCell.width() << " "
<< m_gCell.height() << "\n";
}
if(!m_hasGTile)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
std::cout << "Error : no gtile defined\n";
return false;
}
if(!m_hasOrder)
{
std::cout << "Warning : no order defined, setting to default
1\n";
m_order = 1;
m_hasOrder = true;
}
if(!m_hasLump)
{
std::cout << "Warning : no lump defined,
seting to default 100u\n";
m_lump = 100;
m_hasLump = true;
}
if(!m_hasDriver)
{
std::cout << "Warnign : no driver defined,
setting to defaults 0(ohm) 0(pF)\n";
m_driverRes = 0;
m_driverCap = 0;
m_hasDriver = true;
}
if(!m_hasBuffer)
{
std::cout << "Warning : no buffer defined,
setting to defaults Inf(ohm), 0(ohm) 0(pf)\n";
m_bufferLoad = INF;
m_bufferRes = 0;
m_bufferCap = 0;
m_hasBuffer = true;
}
if(!m_hasLoad)
{
std::cout << "Warning : no load defined,
setting to defult Inf(ohm)\n";
m_load = INF;
m_hasLoad = true;
}
if(m_layers.size()==0)
{
std::cout << "Error : no layers selected for routing\n";
return false;
}
int numLayers = m_layers.size();
    for(int i=0;i<numLayers;i++)
{
if( i-2>=0 ) m_layers[i]->prev(true,m_layers[i-2]);
if( i-1>=0 ) m_layers[i]->prev(false,m_layers[i-1]);
if( i+2<numLayers ) m_layers[i]->next(true,m_layers[i+2]);
if( i+1<numLayers ) m_layers[i]->next(false,m_layers[i+1]);
}

for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
if(!pLayer->hasDir())
{
CLefLayer* pLefLayer = lef.layer(pLayer->name());
if(pLefLayer==NULL  && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pLefLayer->hasDirection()  && pLayer->selected())
{
std::cout <<  "Error : selected layer routing
direction"
 << pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->dir( pLefLayer->direction() );
std::cout << "setting " << pLayer->name() << " direction
to "
<< ::layerDirection(pLayer->dir()) << "\n";
}

if(!pLayer->hasPitch())
{
CLefLayer* pLefLayer = lef.layer(pLayer->name());
if(pLefLayer==NULL && pLayer->selected())
{
std::cout << "Error : selected layer " <<
pLayer->name()
<< " couldn't be found in library\n";
return false;
}
if(!pLefLayer->hasPitch() && pLayer->selected())
{
std::cout <<  "Error : selected layer pitch" <<
pLayer->name()
 << " couldn't be found in library\n";
return false;
}
pLayer->pitch( pLefLayer->pitch() );
std::cout << "setting " << pLayer->name()
<< " pitch to " << pLayer->pitch() << "u\n";
}

if(!pLayer->hasWidth())
{
CLefLayer* pLefLayer = lef.layer(pLayer->name());
if(pLefLayer==NULL && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pLefLayer->hasWidth() && pLayer->selected())
{
std::cout <<  "Error : selected layer width"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
pLayer->width( pLefLayer->width() );
std::cout << "setting " << pLayer->name()
<< " width to " << pLayer->width() << "u\n";
}

if(!pLayer->hasRes())
{
CLefLayer* pLefLayer = lef.layer(pLayer->name());
if(pLefLayer==NULL && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pLefLayer->hasResistance()  && pLayer->selected())
{
std::cout <<  "Error : selected layer
resistance"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->res( pLefLayer->unitRes() );
std::cout << "setting " << pLayer->name()
<< " resistance to " << pLayer->res() << "ohm\n";
}

if(!pLayer->hasCap())
{
CLefLayer* pLefLayer = lef.layer(pLayer->name());
if(pLefLayer==NULL && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pLefLayer->hasCapacitance() && pLayer->selected())
{
std::cout <<  "Error : selected layer
capacitance"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->cap( pLefLayer->unitCap() );
std::cout << "setting " << pLayer->name()
<< " capacitance to " << pLayer->cap() << "u\n";
}
}

std::cout << "calculating gcell capacities\n";
for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
pLayer->init(m_dieArea,m_gCell,m_gTile);
if(!pLayer->selected()) continue;
//read blockages that red from file
if(m_hasBlocks)
{
QValueListIterator< QPair<QString,QString> > blkIt;
for(blkIt=m_cellBlocks.begin();blkIt!=m_cellBlocks.end();blkIt++)
{
if((*blkIt).first!=pLayer->name()) continue;
QString blkS = (*blkIt).second;
QStringList params = QStringList::split(" ",
blkS.simplifyWhiteSpace());
int track = params[0].toInt();
int edge = params[1].toInt();
int blk = params[2].toInt();
if(blk>pLayer->cellCap()) blk =
pLayer->cellCap();
pLayer->cellBlock(track,edge,blk);
}

for(blkIt=m_tileBlocks.begin();blkIt!=m_tileBlocks.end();
blkIt++)
{
if((*blkIt).first!=pLayer->name()) continue;
QString blkS = (*blkIt).second;
QStringList params = QStringList::split(" ",
blkS.simplifyWhiteSpace());
int track = params[0].toInt();
int edge = params[1].toInt();
int blk = params[2].toInt();
if(blk>pLayer->tileCap()) blk =
pLayer->tileCap();
pLayer->tileBlock(track,edge,blk);
}
        continue;
}
QValueList<CRect> allrects;
       //add def blockages
QPtrList<CDefBlock> blocks = def.blocks();
QPtrListIterator<CDefBlock> blockIt(blocks);
for(blockIt.toFirst();blockIt.current();++blockIt)
{
           CDefBlock* pBlock = *blockIt;
if(!pBlock->hasLayer()) continue;
if(pBlock->layerName()!=pLayer->name()) continue;
QValueList<CRect> rects = pBlock->rectangles();
QValueListIterator<CRect> rectIt;
for(rectIt=rects.begin();rectIt!=rects.end();rectIt++)
allrects += (*rectIt)/def.units();
}
// std::cout << "\n\nnum Blk " << allrects.count() << "\n";
//add def pins
QDict<CDefPin> defpins = def.pins();
QDictIterator<CDefPin> defpinIt(defpins);
for(defpinIt.toFirst();defpinIt.current();++defpinIt)
{
CDefPin* pDefPin = *defpinIt;
if(!pDefPin->hasLayer()) continue;
if(pDefPin->layer()!=pLayer->name()) continue;
double x1,y1,x2,y2;
pDefPin->bounds(&x1,&y1,&x2,&y2);
CRect r = CRect(x1,y1,(x2-x1),(y2-y1)) / def.units();
r =
r.orientDefPin(CPoint(pDefPin->placementX()/def.units(),
pDefPin->placementY()/def.units()),pDefPin->orient());
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
allrects += r;
}
// std::cout << "num def pins added " << allrects.count() << "\n";
//add comp pins and obstructions
QDict<CDefComp> comps = def.comps();
QDictIterator<CDefComp> compIt(comps);
for(compIt.toFirst();compIt.current();++compIt)
{
CDefComp* pComp = *compIt;
CLefMacro* pMacro = lef.macro(pComp->name());
if(pMacro==NULL) continue;
            //obstructions
QPtrList<CLefObs> obs = pMacro->obs();
QPtrListIterator<CLefObs> obsIt(obs);
for(obsIt.toFirst();obsIt.current();++obsIt)
{
QValueList< QPair<geomType, void*> >
items = (*obsIt)->items();
     QValueListIterator< QPair<geomType, void*> >
itemIt;
QString layerName;
CRect rect;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
              {
              switch((*itemIt).first)
                 {
                 case GEOMLAYER:
                 layerName =
*(QString*)(*itemIt).second;
break;
             case GEOMRECT:
                    rect =
*(CRect*)(*itemIt).second;
                    if( pMacro->hasOrigin() ) rect =
rect +
CPoint( pMacro->originX(), pMacro->originY() );
                    rect = rect.orientLefPin(
CRect(pComp->placementX()/def.units(),
 pComp->placementY()/def.units(),
                                            pMacro->sizeX(),pMacro->sizeY()
),
                                          pComp->placementOrient() );
if(layerName==pLayer->name())
allrects += rect;
                    break;
                        default: std::cout
<< "CGRoute::initFormLefDef() not implemented case\n";
                   }

             }
}
//pins
QDict<CLefPin> pins = pMacro->pins();
QDictIterator<CLefPin> pinIt(pins);
for(pinIt.toFirst();pinIt.current();++pinIt)
{
CLefPin* pPin = *pinIt;
QPtrList<CLefPort> ports = pPin->ports();
QPtrListIterator<CLefPort> portIt(ports);
              for(portIt.toFirst();portIt.current();++portIt)
              {
              QValueList< QPair<geomType, void*> >
items = (*portIt)->items();
                 QValueListIterator< QPair<geomType,
void*> > itemIt;
QString layer;
CRect r;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
                 {
                 switch((*itemIt).first)
                   {
                   case GEOMLAYER :
                   layer =
*(QString*)(*itemIt).second;
                      break;
                  case  GEOMRECT :
                   r =
*(CRect*)(*itemIt).second;
                      if( pMacro->hasOrigin()
)
r = r + CPoint(
pMacro->originX(), pMacro->originY() );
                      r = r.orientLefPin(
CRect(pComp->placementX()/def.units()
pComp->placementY()/def.units(),

pMacro->sizeX(),pMacro->sizeY() );
                                                          pComp->placementOrient()
);
if(layer==pLayer->name()
) allrects += r;
                      break;
         default: std::cout
<< "CGRoute::initFormLefDef() not implemented case\n";
              }
}
          }
}
}
// std::cout << "comp pins and obs "
<< allrects.count() << "\n";
//add special nets
{QDict<CDefNet> nets = def.snets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL;
CPoint oldpoint,newpoint;
bool firstpoint = true;
double x1,x2,y1,y2,with;
        with = -1;
```

```
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()/def.units()-with/2;
y1 =
oldpoint.y()/def.units();
x2 =
newpoint.x()/def.units()+with/2;
y2 =
newpoint.y()/def.units();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 =
oldpoint.x()/def.units();
y1 =
oldpoint.y()/def.units()-with/2;
x2 =
newpoint.x()/def.units();
y2 =
newpoint.y()/def.units()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();

if(newlayer->name()==pLayer->name()) allrects += r;
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()/def.units()-with/2;
y1 =
oldpoint.y()/def.units();
x2 =
newpoint.x()/def.units()+with/2;
y2 =
newpoint.y()/def.units();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 =
oldpoint.x()/def.units();
y1 =
oldpoint.y()/def.units()-with/2;
x2 =
newpoint.x()/def.units();
y2 =
newpoint.y()/def.units()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();

if(newlayer->name()==pLayer->name()) allrects += r;
break;}
case PATHWIDTH :
{with = *((int*)item);
with = with/def.units();
break;}
default:break;
}
}
}
}}
// std::cout << "special nets added " << allrects.count() << "\n";
//add nets
{QDict<CDefNet> nets = def.nets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL;
CPoint oldpoint,newpoint;
double x1,x2,y1,y2,with;
with=-1;
bool firstpoint=true;
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()/def.units()-newlayer->width()/2;
y1 =
oldpoint.y()/def.units();
x2 =
newpoint.x()/def.units()+newlayer->width()/2;
y2 =
newpoint.y()/def.units();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 =
oldpoint.x()/def.units();
y1 =
oldpoint.y()/def.units()-newlayer->width()/2;
x2 =
newpoint.x()/def.units();
y2 =
newpoint.y()/def.units()+newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();

if(newlayer->name()==pLayer->name()) allrects += r;
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()/def.units()-newlayer->width()/2;
y1 =
oldpoint.y()/def.units();
x2 =
newpoint.x()/def.units()+newlayer->width()/2;
y2 =
newpoint.y()/def.units();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 =
oldpoint.x()/def.units();
y1 =
oldpoint.y()/def.units()-newlayer->width()/2;
x2 =
newpoint.x()/def.units();
y2 =
newpoint.y()/def.units()+newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();

if(newlayer->name()==pLayer->name()) allrects += r;
break;}
case PATHWIDTH :
{with = *((int*)item);
with = with/def.units();
break;}
default:break;
```

```
}
}
}
}}
// std::cout << "ALL Num Items " <<  allrects.count() << "\n";
pLayer->init(allrects);
      //allrects process;
}

//intialize nets
QDict<CDefNet> nets = def.nets();
QDictIterator<CDefNet> netIt(nets);
QStringList selectNets;
//select all nets except POWER,GROUND,CLOCK
//and alos one pin nets
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* pDefNet = *netIt;
if( pDefNet->use()!=POWER && pDefNet->use()!=GROUND &&
pDefNet->use()!=CLOCK &&
    pDefNet->numPins()>1)
selectNets.append(pDefNet->name());
}
//remove nets that matches unselected net patterns
QValueListIterator<QString> unsNetPIt;
for(unsNetPIt=m_unselectNets.begin();unsNetPIt!=m_unselectNets.end();unsNetPIt+
+)
{
QString pattern = *unsNetPIt;
QStringList unselectNets = selectNets.grep(
QRegExp(pattern,true,true) );
QValueListIterator<QString> unsNetIt;

for(unsNetIt=unselectNets.begin();unsNetIt!=unselectNets.end();unsNetIt++)
{
QString unsNet = *unsNetIt;
std::cout << "unselect net " << unsNet << "\n";
selectNets.remove(unsNet);
}
}
//initialize selected nets
QValueListIterator<QString> selNetIt;
for(selNetIt=selectNets.begin();selNetIt!=selectNets.end();selNetIt++)
{
QString nname = *selNetIt;
CDefNet* pDefNet = def.net(nname);
CNet* pNet = new CNet(nname);
QDict<CNetPin> pins = pDefNet->pins();
QDictIterator<CNetPin>  pinIt(pins);
for(pinIt.toFirst();pinIt.current();++pinIt)
{
CNetPin* pNetPin = *pinIt;
QString pinstance = pNetPin->instance();
QString pname = pNetPin->name();
pinDirectionType dir;
double x1,y1,x2,y2;
CRect r;
QString layer;
QValueList< QPair<QString, CPoint> > pinPoints;
if(pinstance=="PIN")
{
CDefPin* pDefPin = def.pin(pname);
if(pDefPin==NULL)
{ std::cout << "Warning : " << pinstance << "-"
<< pname
<< " of " << nname << " couldn't found in def, skipping pin\n";continue;}
if(!pDefPin->hasDirection())
{ std::cout << "Warning : " << pinstance << "-"
<< pname << " of " << nname << " has no direction, assuming IN\n";}
if(!pDefPin->hasLayer())
{ std::cout << "Warning : " << pinstance << "-"
<< pname << " of " << nname << " has no layer, skipping pin\n";continue;}
if(!pDefPin->hasPlacement())
              { std::cout << "Warning : " << pinstance << "-"
<< pname << " of " << nname << " has no placement, skipping pin\n";continue;}
if(pDefPin->hasDirection())
{
if(pDefPin->direction()==OUT) dir = IN;
else dir = OUT;
} else dir = IN;
    pDefPin->bounds(&x1,&y1,&x2,&y2);
              r = CRect(x1,y1,(x2-x1),(y2-y1)) / def.units();
              r =
r.orientDefPin(CPoint(pDefPin->placementX()/def.units(),
pDefPin->placementY()/def.units()),pDefPin->orient());
layer = pDefPin->layer();
pinPoints <<
QPair<QString,CPoint>::QPair(layer,r.center());
}else
{
CDefComp* pDefComp = def.comp(pinstance);
if(pDefComp==NULL)
{
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " reference to component
couldn't found, skipping pin\n";
continue;
}
if(pDefComp->isUnplaced())
              {
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " component is not placed,
skipping pin\n";
continue;
}
CLefMacro* pLefMacro = lef.macro(
pDefComp->name() );
if( pLefMacro==NULL)
{
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " reference to macro
couldn't found, skipping pin\n";
continue;
}
if( !pLefMacro->hasSize() )
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
{
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " macro has no size,
skipping pin\n";
continue;
}
CLefPin* pLefPin = pLefMacro->pin( pname );
    if( pLefPin==NULL)
{
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " reference to pin couldn't
found, skipping pin\n";
continue;
}
if( pLefPin->numPorts()==0 )
{
std::cout << "Warning : " << pinstance
<< "-" << pname << " of " << nname;
std::cout << " pin has no ports,
skipping pin\n";
continue;
}
if( pLefPin->hasDirection() )
{
if(pLefPin->direction()==OUT ) dir=OUT;
else dir=IN;
}else
{
dir = tim.dir( nname, pinstance );
}

QPtrList<CLefPort> ports = pLefPin->ports();
QPtrListIterator<CLefPort> portIt(ports);
for(portIt.toFirst();portIt.current();++portIt)
{
QValueList< QPair<geomType, void*> >
items = (*portIt)->items();
QValueListIterator< QPair<geomType,
void*> > itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
switch((*itemIt).first)
{
case GEOMLAYER :
layer =
*(QString*)(*itemIt).second;
break;
case  GEOMRECT :
r =
*(CRect*)(*itemIt).second;
if(
pLefMacro->hasOrigin() )
r = r + CPoint(
pLefMacro->originX(), pLefMacro->originY() );
r = r.orientLefPin(
CRect(  pDefComp->placementX()/def.units(),
 pDefComp->placementY()/def.units(),

pLefMacro->sizeX(),pLefMacro->sizeY() ),pDefComp->placementOrient() );
pinPoints <<
QPair<QString,CPoint>::QPair(layer,r.center());
break;
default:
std::cout <<
"CGRoute::initFormLefDef() not implemented case\n";
}
}
}
}
if(pinPoints.isEmpty())
    {
std::cout << "Warning : " << pinstance << "-"
<< pname << " of " << nname;
std::cout << " has no ports, skipping pin\n";
continue;
}
if(dir==OUT)
{
if(pNet->root()!=NULL)
{
std::cout << "Warning : " << pinstance
<< "-"
<< pname << " of " << nname;
std::cout << " the net already have a
source,
converting output pin to input pin\n";
dir=IN;
}else
{
CSource* pSource = new
CSource(pinstance+"-"+pname,
pinPoints.front().first,
pinPoints.front().second,
m_driverRes,m_driverCap);
pNet->root(pSource);
}
}
if(dir==IN)
{
CSink* pSink = new CSink(pinstance+"-"+pname,
pinPoints.front().first,
pinPoints.front().second,
m_load, tim.rat(nname,pinstance));
pNet->addSink( pSink );
}
}
if(pNet->numNodes()==0)
{
std::cout << "Warning : " << (QString)*pNet << " has no
pins\n";
continue;
}
if(pNet->root()==NULL)
{
std::cout << "Warning : no source pin detected for " <<
```

```
(QString)*pNet;
std::cout << ", assuming one of the pins as source
pin\n";
pNet->makeRoot(m_driverRes,m_driverCap);
}
// if(pNet->numNodes()>4) continue;//willbe removed, for faster
debugging
// if(pNet->numNodes()!=3 ) continue; // filter out 3 pin nets;
m_nets.insert(nname, pNet);
pNet->layer( CLayerPair(layer("METAL3"),layer("METAL4")) );
}
return true;
}

bool CGRoute::initFromCct()
{
extern CStr str;
extern CDsn dsn;
extern CPla pla;
extern CLib lib;
extern CNetList netlist;
extern CTim tim;
extern CTiming timing;
extern bool hasTim;
extern bool hasTiming;
// init die area;
if(!m_hasDieArea)
{
if(!str.hasDiePoly()){std::cout
<< "Error : no die area defined\n";return false; }
      m_dieArea = str.diePoly().boundingRect();
m_hasDieArea = true;
std::cout << "setting die area (";
std::cout << m_dieArea.left() << " " << m_dieArea.bottom() << "
: ";
std::cout << m_dieArea.right() << " " << m_dieArea.top() <<
")\n";
}
//gcell
if(!m_hasGCell) {std::cout << "Error : no gcell defined\n";return
false;}
if(!m_hasGTile) {std::cout << "Error : no gtile defined\n";return
false;}
if(!m_hasOrder)
{
std::cout << "Warning : no order defined, setting to default
1\n";
m_order = 1;
m_hasOrder = true;
}
if(!m_hasLump)
{
std::cout << "Warning : no lump defined, seting to default
100u\n";
m_lump = 100;
m_hasLump = true;
}
if(!m_hasDriver)
{
std::cout << "Warnign : no driver defined, setting to defaults
0(ohm) 0(pF)\n";
m_driverRes = 0;
m_driverCap = 0;
m_hasDriver = true;
}
if(!m_hasBuffer)
{
std::cout << "Warning : no buffer defined, setting to defaults
Inf(ohm), 0(ohm) 0(pf)\n";
m_bufferLoad = INF;
m_bufferRes = 0;
m_bufferCap = 0;
m_hasBuffer = true;
}
if(!m_hasLoad)
{
std::cout << "Warning : no load defined, setting to defult
Inf(ohm)\n";
m_load = INF;
m_hasLoad = true;
}

if(m_layers.size()==0)
{
std::cout << "Error : no layers selected for routing\n";
return false;
}

int numLayers = m_layers.size();
    for(int i=0;i<numLayers;i++)
{
if( i-2>=0 ) m_layers[i]->prev(true,m_layers[i-2]);
if( i-1>=0 ) m_layers[i]->prev(false,m_layers[i-1]);
if( i+2<numLayers ) m_layers[i]->next(true,m_layers[i+2]);
if( i+1<numLayers ) m_layers[i]->next(false,m_layers[i+1]);
}

for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
if(!pLayer->hasDir())
{
CCctLayer* pCctLayer = str.layer(pLayer->name());
if(pCctLayer==NULL && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pCctLayer->hasDirection() && pLayer->selected())
{
std::cout <<  "Error : selected layer routing
direction"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->dir( pCctLayer->dir());
std::cout << "setting " << pLayer->name() << " direction
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
to "
<< ::layerDirection(pLayer->dir()) << "\n";
}

if(!pLayer->hasPitch())
{
CCctLayer* pCctLayer = str.layer(pLayer->name());
if(pCctLayer==NULL  && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pCctLayer->hasPitch()  && pLayer->selected())
{
std::cout <<  "Error : selected layer pitch"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->pitch( pCctLayer->pitch());
std::cout << "setting " << pLayer->name()
<< " pitch to " << pLayer->pitch() << "u\n";
}

if(!pLayer->hasWidth())
{
CCctLayer* pCctLayer = str.layer(pLayer->name());
if(pCctLayer==NULL  && pLayer->selected())
{
std::cout << "Error : selected layer "
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
if(!pCctLayer->hasWidth()  && pLayer->selected())
{
std::cout <<  "Error : selected layer width"
<< pLayer->name() << " couldn't be found in library\n";
return false;
}
pLayer->width( pCctLayer->width());
std::cout << "setting " << pLayer->name()
<< " width to " << pLayer->width() << "u\n";
}


if(!pLayer->hasRes() && pLayer->selected())
{
std::cout << pLayer->name()
<< " doesn't have unit resistance\n";
return false;
}

if(!pLayer->hasCap() && pLayer->selected())
{
std::cout << pLayer->name()
<< " doesn't have unit capacitance\n";
return false;
}
}
```

```
std::cout << "calculating gcell capacities\n";
for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
pLayer->init(m_dieArea, m_gCell,m_gTile);
if(!pLayer->selected()) continue;
//read blockages that red from file
if(m_hasBlocks)
{
QValueListIterator< QPair<QString,QString> > blkIt;

for(blkIt=m_cellBlocks.begin();blkIt!=m_cellBlocks.end();blkIt++)
{
if((*blkIt).first!=pLayer->name()) continue;
QString blkS = (*blkIt).second;
QStringList params =
QStringList::split(" ",blkS.simplifyWhiteSpace());
int track = params[0].toInt();
int edge = params[1].toInt();
int blk = params[2].toInt();
if(blk>pLayer->cellCap()) blk =
pLayer->cellCap();
pLayer->cellBlock(track,edge,blk);
}

for(blkIt=m_tileBlocks.begin();blkIt!=m_tileBlocks.end();blkIt++)
{
if((*blkIt).first!=pLayer->name()) continue;
QString blkS = (*blkIt).second;
QStringList params =
QStringList::split(" ",blkS.simplifyWhiteSpace());
int track = params[0].toInt();
int edge = params[1].toInt();
int blk = params[2].toInt();
if(blk>pLayer->tileCap()) blk =
pLayer->tileCap();
pLayer->tileBlock(track,edge,blk);
}
continue;
}
QValueList<CRect> allrects;
allrects.clear();
//add keepouts
QPtrList<CCctKeepout> keepouts = str.keepouts();
QPtrListIterator<CCctKeepout> keepoutIt(keepouts);
for(keepoutIt.toFirst();keepoutIt.current();++keepoutIt)
{
CCctKeepout* pKeepout = *keepoutIt;
if(pKeepout->layer()!=pLayer->name())continue;
allrects += pKeepout->rect();
}
//add wiring
QValueList< QPair<QString,CRect> > wiring = dsn.wiring();
QValueListIterator< QPair<QString,CRect> > wiringIt;
for(wiringIt=wiring.begin();wiringIt!=wiring.end();wiringIt++)
{
if((*wiringIt).first!=pLayer->name() ) continue;
allrects += (*wiringIt).second;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
// //add comp pins and keepouts
QDict<CCctComp> comps = pla.comps();
QDictIterator<CCctComp> compIt(comps);
for(compIt.toFirst();compIt.current();++compIt)
{
CCctComp* pComp = *compIt;
CCctImage* pImage = lib.image(pComp->image());
if(pImage==NULL) continue;
//keepouts
QPtrList<CCctKeepout> keepouts = pImage->keepouts();
QPtrListIterator<CCctKeepout> keepoutIt(keepouts);
for(keepoutIt.toFirst();keepoutIt.current();++keepoutIt)
{
CCctKeepout* pKeepout = *keepoutIt;
if(pKeepout->layer()!=pLayer->name()) continue;
allrects += (pKeepout->rect() +
pComp->placement());
}
        //pins
QDict<CCctPin> pins = pImage->pins();
QDictIterator<CCctPin> pinIt(pins);
for(pinIt.toFirst();pinIt.current();++pinIt)
{
CCctPin* pPin = *pinIt;
QValueList< QPair<QString, CRect> > ports =
pPin->ports();
QValueListIterator< QPair<QString, CRect> >
portIt;

for(portIt=ports.begin();portIt!=ports.end();portIt++)
{

if((*portIt).first!=pLayer->name())continue;
allrects += ((*portIt).second +
pComp->placement());
}
}
}
pLayer->init(allrects);
}

//intialize nets
QDict<CCctNet> nets = netlist.nets();
QDictIterator<CCctNet> netIt(nets);
QStringList selectNets;
//select all nets
for(netIt.toFirst();netIt.current();++netIt)
{
CCctNet* pCctNet = *netIt;
selectNets.append(pCctNet->name());
}
//remove nets that matches unselected net patterns
QValueListIterator<QString> unsNetPIt;
for(unsNetPIt=m_unselectNets.begin();unsNetPIt!=m_unselectNets.end();unsNetPIt+
+)
{
QString pattern = *unsNetPIt;
QStringList unselectNets = selectNets.grep(
QRegExp(pattern,true,true) );
QValueListIterator<QString> unsNetIt;

for(unsNetIt=unselectNets.begin();unsNetIt!=unselectNets.end();unsNetIt++)
{
QString unsNet = *unsNetIt;
std::cout << "unselect net " << unsNet << "\n";
selectNets.remove(unsNet);
}
}

//initialize selected nets
QValueListIterator<QString> selNetIt;
for(selNetIt=selectNets.begin();selNetIt!=selectNets.end();selNetIt++)
{
QString nname = *selNetIt;
CCctNet* pCctNet = netlist.net(nname);
CNet* pNet = new CNet(nname);
QStringList pins = pCctNet->pins();
QValueListIterator<QString>  pinIt;
for(pinIt=pins.begin();pinIt!=pins.end();++pinIt)
{
QString pinstance = QStringList::split("-",*pinIt)[0];
QString pname = QStringList::split("-",*pinIt)[1];

pinDirectionType dir;
CRect r;
QString layer;
QValueList< QPair<QString, CPoint> > pinPoints;

CCctComp* pCctComp = pla.comp(pinstance);
if(pCctComp==NULL)
  { std::cout << "Warning : " << pinstance << "-" << pname
<< " of " << nname;
  std::cout << " reference to component couldn't
found, skipping pin\n";continue;}
  CCctImage* pCctImage = lib.image( pCctComp->image() );
  if( pCctImage==NULL)
  { std::cout << "Warning : " << pinstance << "-" << pname
<< " of " << nname;
  std::cout << " reference to macro couldn't
found, skipping pin\n";continue;}
  CCctPin* pCctPin = pCctImage->pin( pname );
      if( pCctPin==NULL)
  { std::cout << "Warning : " << pinstance << "-" << pname
<< " of " << nname;
  std::cout << " reference to pin couldn't found,
skipping pin\n";continue;}

dir = (hasTiming)?
timing.dir( nname, pinstance+"-"+pname ) : tim.dir( nname, pinstance );

QValueList< QPair<QString,CRect> > ports =
pCctPin->ports();
  QValueListIterator< QPair<QString,CRect> > portIt;
  for(portIt=ports.begin();portIt!=ports.end();portIt++)
  {
layer = (*portIt).first;
r = (*portIt).second;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
r = r + pCctComp->placement();
pinPoints <<
QPair<QString,CPoint>::QPair(layer,r.center());
  }
  if(pinPoints.isEmpty())
      { std::cout << "Warning : " << pinstance << "-" << pname
<< " of " << nname;
  std::cout << " has no ports, skipping
pin\n";continue;}
  if(dir==OUT)
  {
  if(pNet->root()!=NULL)
  { std::cout << "Warning : " << pinstance << "-"
<< pname << " of " << nname;
  std::cout
<< " the net already have a source, converting output pin to input pin\n";
  dir=IN;}
  else
  {
CSource* pSource = new
CSource(pinstance+"-"+pname,
pinPoints.front().first,
pinPoints.front().second,
m_driverRes,m_driverCap);
  pNet->root(pSource);
  }
  }
  if(dir==IN)
  {
double rat = (hasTiming)?
timing.rat(nname,pinstance+"-"+pname) : tim.rat(nname,pinstance);
//std::cout << nname << "\trat = " << rat <<
"\n";
CSink* pSink = new CSink(pinstance+"-"+pname,
pinPoints.front().first,
pinPoints.front().second,
m_load, rat);
  pNet->addSink( pSink );
  }
}
if(pNet->numNodes()==0)
{
std::cout << "Warning : " << (QString)*pNet << " has no
pins\n";
continue;
}
if(pNet->root()==NULL)
{
std::cout << "Warning : no source pin detected for " <<
(QString)*pNet;
std::cout << ", assuming one of the pins as source
pin\n";
pNet->makeRoot(m_driverRes,m_driverCap);

}
// extern QTextStream lg;
// if(pNet->numNodes()>5) {lg << "unselect net " << (QString)*pNet
<< "\n";continue;}//willbe removed, for faster debugging
//if(pNet->numNodes()!=4) continue;
m_nets.insert(nname, pNet);
//pNet->layer( CLayerPair(layer("METAL3"),layer("METAL4")) );
// if(m_nets.count()>=1000)  break;
}

return true;
}

void CGRoute::init(CNet* net)
{
net->clear();

QValueList<CEdge> edges = net->edges();
while(!edges.isEmpty())
{
CEdge edge = edges.front();edges.pop_front();
CNode* down = edge.down();
CNode* up = edge.up();
if(down->type()==ROOT || down->type()==SUBROOT ) continue;
gwire* wire = new gwire(up,down);
((QPtrList<gwire>*)net)->append(wire);
}
}

bool CGRoute::gen_rnd_tim()
{
extern CTiming timing;
randtim();
QDictIterator<CNet> netIt(m_nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
QString net = QString(*pNet);
timing.net(net);

CSource* root = (CSource*)pNet->root();
QString source = QString(*root);
timing.source(net,source);

QPtrList<CNode> nodes = pNet->nodes();
QPtrListIterator<CNode> nIt(nodes);
for(nIt.toFirst();nIt.current();++nIt)
{
CNode* node = *nIt;
if( node->type() != SINK ) continue;
CSink* sink = (CSink*)node;
QString sinks = QString(*sink);
double rat = sink->rat();
timing.sink(net,sinks,rat);
}
}
return true;
}
#include <qdatetime.h>

bool CGRoute::route()
{
// QTime t,tslabi,tgo,tnet;
// unsigned long int slabit,got,nett;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// t.start();
if(!m_initd) {if(!init()) return false;}
// tslabi.start();
slabi();
// slabit = tslabi.elapsed();
// tgo.start();
go();
// got = tgo.elapsed();
// tnet.start();
net();
// nett = tnet.elapsed();
// std::cout << "CPU SLABI " << slabit << "\n";
// std::cout << "CPU GO " << got << "\n";
// std::cout << "CPU NET " << nett << "\n";
// std::cout << "\tover :" << numCellOverflows() <<"\n";
// std::cout << "\tpads :" << numPadViolations() << "\n";
// std::cout << "Maze now\n";
// maze();
std::cout << "\tover :" << numCellOverflows() << "\n";
// std::cout << "\tpads :" << numPadViolations() << "\n";
// std::cout << "CPU : " << t.elapsed() << "\n";
/*
QMap<int,int> hist;
for(int i=0;i<1000;i++) hist[i] = 0;
for(unsigned int i=0;i<m_layers.count();i++)
{
CLayer* l = m_layers[i];
if(!l->selected()) continue;
for(int i=0;i<l->numcTracks();i++)
for(int j=0;j<l->numcEdges();j++)
{
int flow = l->cellFlow(i,j) + l->cellBlock(i,j);
int cap = l->cellCap();
int r = int(1000*(double(flow)/double(cap)));
hist[r] = hist[r] + 1;
}
}
QFile histF("testgo.hst");
histF.open(IO_WriteOnly);
QTextStream f(&histF);
for(int i=0;i<1000;i++)
f << i << " " << hist[i] << "\n";
histF.close();

QMap<int,int> histnet;
for(int i=0;i<1000;i++) histnet[i] = 0;
for(unsigned int i=0;i<m_layers.count();i++)
{
CLayer* l = m_layers[i];
if(!l->selected()) continue;
for(int i=0;i<l->numtTracks();i++)
for(int j=0;j<l->numtEdges();j++)
{
int flow = l->tileFlow(i,j) + l->tileBlock(i,j);
int cap = l->tileCap();
int r = int(1000*(double(flow)/double(cap)));
histnet[r] = histnet[r] + 1;
}
}
QFile histFNet("testnet.hst");
histFNet.open(IO_WriteOnly);
QTextStream fnet(&histFNet);
for(int i=0;i<1000;i++)
fnet << i << " " << histnet[i] << "\n";
histFNet.close();*/
return true;
}
#include "cgroute.h"
#include "cnet.h"
#include "cnode.h"
#include "csource.h"
#include "csubroot.h"
#include "csink.h"

double CGRoute::awe(CNet* net,bool log)
{
CNode *root;
CNode *node;
QPtrList<CNode> nodes;
double maxDV = -INF;
double DV;

//  QFile F("awe.log");
 //  F.open(IO_WriteOnly | IO_Append);
 //  QTextStream f(&F);
//   if(log) f << "\ncalculating the delay of " << *net << "\n";
//   F.close();

  // net->segment( m_lump );
root=net->root();
root->parent()->resizeMoment(2*m_order);

net->computeCapAndR();
root->delay( root->edge().res() * ((CSource*)root)->cap() * 0.693 );
nodes.append(root );
   while( !nodes.isEmpty() )
   {
    node = nodes.take(0);
QPtrListIterator<CNode> n(node->children());
     for(n.toFirst();n.current(); ++n) nodes.append( *n );
   if(node->type() == ROOT || node->type() == SUBROOT )
{
DV = awe(node,log);
if( DV > maxDV ) maxDV = DV;
}
}
   //net->desegment();
   net->maxDV( maxDV );
return maxDV;
}

double CGRoute::awe(CNode* node,bool log)
{
//QFile F("awe.log");
//F.open(IO_WriteOnly | IO_Append);
//QTextStream f(&F);
//if(log){f << "\nAWESUB\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//f<<"type \t\t node \t length \t cap \t res \t <--- \t parent\n";
//f <<
"---------------------------------------------------------------------------\n";}

QPtrList<CNode> nodes;
int ord=m_order;
bool reduce;
double td = 0;
double maxDV = -INF;
complex deno[MAX_ORDER+1], mat[MAX_ORDER+1][MAX_ORDER+1],
vec[2*MAX_ORDER+1];
   complex pole[MAX_ORDER+1], resid[MAX_ORDER+1];

if (node->type() == SUBROOT)
node->delay( node->parent()->delay() +
10 + node->edge().res() * ((CSubRoot*)node)->cap() * 0.693 );
double delay = node->delay();
   node->resetSubtree(m_order);
   for (int i=1; i<2*ord; i++)
{
     node->computeCurrent(i);
node->parent()->moment(0,i);
     node->computeMoment(i);
  }

nodes.append( node );
while(!nodes.isEmpty())
{
node = nodes.take(0);
QPtrListIterator<CNode> n(node->children());
for(n.toFirst();n.current();++n) if((*n)->type()!=SUBROOT )
nodes.append( *n );
//if(log){ f << ::nodeTypeStr(node->type()) << "\t" << node;
// f << "\t" << node->edge().manhDist() << "\t"
<< node->cap() << "\t" << node->edge().res();
// f << "\t<-- ";
//  f << ::nodeTypeStr(node->parent()->type()) << "\t" << node->parent() <<
"\n";}
if(node->type()==SINK || node->type()==BUFFER)
{
ord = m_order;
           reduce = true;
           while( (reduce==true) && (ord>1) )
{
for (int i=1; i<=ord;i++)
{
   for(int j=1; j<=ord; j++)  mat[i][j] =
node->moment(i+j-2);
   vec[i] = -node->moment(ord+i-1);
}
gauss(ord, mat, resid, vec);
deno[0] = 1/resid[1];
for (int i=1;i<ord;i++) deno[i] =
resid[ord-i+1]/resid[1];
         if(computePoles(deno, pole, ord))
{
for (int i=1;i<=ord;i++)
{
     for(int j=1;j<=ord; j++)  mat[i][j] =
1/pow(pole[j], i-1);
vec[i] = -node->moment(i-1);
        }
        gauss(ord, mat, resid, vec);
reduce = false;
       polish(resid, pole, ord, reduce);
       if(reduce) continue;
 td = solveDelay(node->moment(1), resid,
pole, ord);
} else ord--;
        }
      if(reduce) td = -.693*node->moment(1);
   //  td = td;*10.0;
     td += delay;
     node->delay(td);
//if(log){       f <<  "\tdelay = " << td << "\n";
 //     f << "\tmoments = ";
  //     for(int i=1;i<2*ord;i++)  f << ",  " << node->moment(i);
  //     f << "\n";}
if( node->type()==SINK && ((CSink*)node)->dv()>maxDV )
maxDV = ((CSink*)node)->dv();
}
}
// F.close();
return maxDV;
}

#include <qtextstream.h>

double CGRoute::solveDelay(double momone, complex resid[MAX_ORDER+1], complex
pole[MAX_ORDER+1], int odr)
{

  complex vec[MAX_ORDER+1], guess, tmp;
  double err, td, elmore;
  td = 0;
  int i, j, limit = 10;

  guess = elmore = -.693*momone;

  err = INF;
  i=0;
  while (err > SMALL && i <= limit) {
    vec[0] = 0.5;
    for (j = 1; j <= 4; j++) vec[j]=0;
    for (j = 1; j <= odr; j++) {
      tmp = resid[j]*exp(pole[j]*guess);
      vec[0] += tmp;
      vec[1] += tmp*pole[j];
      vec[2] += tmp*pow(pole[j],2)/2;
      vec[3] += tmp*pow(pole[j],3)/6;
      vec[4] += tmp*pow(pole[j],4)/24;
    }

    for (j = 0; j <= 4; j++)
      vec[j] = vec[j]/vec[4];
    if (!quartic(vec))
      return elmore;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    td = INF;
    for (j = 1; j <= 4; j++) {
      if (fabs(vec[j].get_real()) < fabs(td) && fabs(vec[j].get_imag()) < TINY)
td = vec[j].get_real();
    }
    if (td == INF || (td + guess.get_real() < 0)) {
      err = INF;
      guess = 0.8*guess;
    }
    else {
      err = fabs(td/guess.get_real());
      td += guess.get_real();
      guess = td;
    }
    i++;
  }

  if (i > limit && err > SMALL)
    td = elmore;

  return td;
}


#include "cgroute.h"
#include "cnet.h"
#include "cedge.h"
#include "cnode.h"
#include "csubroot.h"
#include "csink.h"


bool CGRoute::insertBuffer(QValueList<CNet*> bubbleList)
{
bool   inserted = false;
while(!bubbleList.isEmpty())
{
if(insertBuffer( bubbleList.back() )) inserted = true;
// while(insertBuffer( bubbleList.back() ) );
bubbleList.pop_back();
}
return inserted;
}

bool CGRoute::insertBuffer(CNet* net, bool moder)
{
double rate = 1.1;
     if(moder) rate = 1.0;
bool inserted = false;
     bool mode = true;
     awe(net);
net->saveDV();
if( net->maxDV() <=0.0 ) return false;
QValueList<CEdge> edges = net->edges();
  //  net->saveDV();
double bestDVimp = 0.1*net->maxDV();//net->DVImp();
double bestDV = 2*INF;
 //   if(bestDV<15) bestDV = 15;
     bestDVimp = 0;
```

```
CEdge bestEdge;
QValueListIterator<CEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
         //net->saveDV();
awe(net);
// net->dump("original bestDV="+QString().setNum(bestDV)+
" bestDVImp=" + QString().setNum(bestDVimp)+"
DV="+QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
if(net->insertBufferAtEdge(*eIt, m_bufferLoad,
m_bufferRes, m_bufferCap))
         {
                 awe(net);
// net->dump("buffer canidate at edge bestDV="+
QString().setNum(bestDV)+" bestDVImp=" + QString().setNum(bestDVimp)+
" DV="+QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
           if( net->DVImp() > bestDVimp && net->DVImp() > 15 )
if( (bestDV > 0) || (bestDV<=0 && net->maxDV()<bestDV) )
       //       if( net->DVImp() > bestDV && net->DVImp() > 0.1 )
           {
         //          std::cout << *net << "\t" << bestDV
<< "\t" <<  net->numBuffers() << "\t" << net->DVImp() << " Poss edge\n";
              bestDVimp = net->DVImp();
              bestEdge = *eIt;
              inserted = true;
                     mode = true; //it is an edge
// net->dump("DVImp
bestDV="+QString().setNum(bestDV)+
" bestDVImp=" + QString().setNum(bestDVimp)+" DV="
+QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
           }
if( net->maxDV() < bestDV && net->maxDV() <= 0 )
{
bestDV = net->maxDV();
bestEdge = *eIt;
inserted = true;
mode = true;
// net->dump("maxDV
bestDV="+QString().setNum(bestDV)+
" bestDVImp=" + QString().setNum(bestDVimp)+" DV="+
QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
}
                 net->removeBufferFromEdge(*eIt );
           }

if(net->insertBufferAtNode(*eIt,m_bufferLoad,m_bufferRes,m_bufferCap))
           {
                 awe(net);
// net->dump("buffer candidate at node bestDV="+
QString().setNum(bestDV)+" bestDVImp=" +
QString().setNum(bestDVimp)+" DV="+QString().setNum(net->maxDV())+"
DVImp="+QString().setNum(net->DVImp()));
                 if(net->DVImp() > bestDVimp  && net->DVImp() > 15 )
if( (bestDV > 0) || (bestDV<=0 && net->maxDV()<bestDV) )
               //if( net->DVImp() > bestDV && net->DVImp() > 0.1 )
                 {
         //          std::cout << *net << "\t" << bestDV <<
 net->numBuffers() << "\t" << net->DVImp() << " Poss node\n";
                   bestDVimp = net->DVImp();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    bestEdge = *eIt;
                    inserted = true;
                    mode = false; // it is a node
// net->dump("DVImp
bestDV="+QString().setNum(bestDV)+
" bestDVImp=" + QString().setNum(bestDVimp)+"
DV="+QString().setNum(net->maxDV())+
" DVImp="+QString().setNum(net->DVImp()));
                }
if( net->maxDV() < bestDV && net->maxDV() <= 0 )
{
bestDV = net->maxDV();
bestEdge = *eIt;
inserted = true;
mode = false;
// net->dump("maxDV
bestDV="+QString().setNum(bestDV)+
" bestDVImp=" + QString().setNum(bestDVimp)+
" DV="+QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
}
            net->removeBufferFromNode(*eIt );
            }
}
if(inserted)
    {
        if(mode)
 net->insertBufferAtEdge(bestEdge, m_bufferLoad, m_bufferRes, m_bufferCap);
        else
net->insertBufferAtNode(bestEdge, m_bufferLoad, m_bufferRes, m_bufferCap);
awe(net);
// net->dump("bufer inserted bestDV="+QString().setNum(bestDV)+"
 bestDVImp=" + QString().setNum(bestDVimp)+" DV="+
QString().setNum(net->maxDV())+" DVImp="+QString().setNum(net->DVImp()));
    }
awe( net );
return inserted;
}


void CGRoute::removeBuffers(QValueList<CNet*> bubbleList)
{

//QFile F("delay.log");
///F.open(IO_WriteOnly);
//QTextStream f(&F);

    //static int count = 0;
    //static int total = 0;
    //double rate = 1.1;
while(!bubbleList.isEmpty())
{
CNet* net = bubbleList.back();
        {
            QPtrList<CNode> nodes = net->nodes();
            while(!nodes.isEmpty())
            {
                CNode* n = nodes.take(0);
                if(n->type()!=SUBROOT) continue;
                ((CSubRoot*)n)->done(false);
                }
            }

//          f << "\n" << *net << "\n";

        again:
         QPtrList<CNode> nodes = net->nodes();
         while(!nodes.isEmpty())
         {
             CNode* n = nodes.take(0);
             if(n->type()!=SUBROOT) continue;
             if(((CSubRoot*)n)->done()) continue;
             double exDV = awe(net);
             net->removeBuffer( n, false);
             double DV = awe(net);
              if(DV<=0)
{/*f << "removed\n";*/ net->removeBuffer(n,true);goto again;}
              else if(exDV<=0 && DV>0)
 {net->reInsertBuffer(n,m_bufferLoad,m_bufferRes,m_bufferCap);}
              else if(DV-exDV>15)
{net->reInsertBuffer(n,m_bufferLoad,m_bufferRes,m_bufferCap);}
              else
{/* f << "removed\n";*/net->removeBuffer(n, true); goto again;}
    //        std::cout << "\n";
         }
awe(net);
// if((QString)(*net)=="net49") net->dump("buffers removed");
bubbleList.pop_back();
}
//    F.close();
}

void CGRoute::removeBuffers()
{
QDictIterator<CNet> netIt(m_nets);
for(netIt.toFirst();netIt.current();++netIt)
(*netIt)->removeBuffers();
}


#include "cgroute.h"
#include "clayer.h"
#include "cnet.h"
#include "cnode.h"
#include "cedge.h"
#include "gwire.h"
#include "gpath.h"
#include <qdatetime.h>

void CGRoute::go()
{
int c = 0;

QValueStack<int> overflows;
for(int i=0;i<10;i++) overflows.push(INF);
int numBnds = 0;
for(unsigned int i=0;i<m_layers.size();i++)
if(m_layers[i]->selected())
numBnds += m_layers[i]->numtTracks()
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* m_layers[i]->numtEdges();
double* fixes = new double[numBnds];
for(int i=0;i<numBnds;i++) fixes[i] = -1;
QValueList< QPair<int,double> > posFixes;

QPtrList<gwire> wires;
int pass = 1;
std::cout << "LP global optimization\n";
std::cout << "pass " << pass++ <<"\n";
double* pi = gomain(wires,fixes,&posFixes); //return;
int prevNumOver;
int numOver = numTileOverflows();
overflows.push(numOver);
std::cout << numOver <<  " overflows\n";
while(numOver>0)
{
prevNumOver = numOver;
//if(ripupwires()<=0) break;
if(ripupwires(pi,wires)<=0) break;
std::cout << "pass " << pass++ << "\n";
lawires();
pi = gomain(wires,fixes,&posFixes);
numOver = numTileOverflows();
std::cout << numOver << " overflows\n";
overflows.push(numOver);
if(overflows.count()>10) overflows.pop();
int tot=0;
for(int i=0;i<overflows.count();i++) tot+=overflows[i];
int avg = tot/overflows.count();
// if(numOver>avg) break;
// if(numOver<5000) break;
if(numOver>0 && numOver>=prevNumOver)
{
while(!posFixes.empty())
{
int seg = posFixes.front().first;
double val = posFixes.front().second;
posFixes.pop_front();
fixes[seg] = val;
}
}
}//while(numOver>0/* && numOver<prevNumOver*/);
delete[] pi;
}

double* CGRoute::gomain(QPtrList<gwire>& wires,
double* fixes, QValueList< QPair<int,double> >* posFixes)
{
int numBnds = 0;
for(int i=0;i<m_layers.size();i++)
if(m_layers[i]->selected())
numBnds +=
m_layers[i]->numtTracks() * m_layers[i]->numtEdges();
//get unrouted wires
wires = this->wires();
wires.first();
while(wires.current())
{
wires.current()->decTileFlows(m_gTile);
wires.current()->clear();
wires.current()->routed(false);
wires.next();
}
wires.first();
while(wires.current())
{
if(wires.current()->routed()) {wires.remove();continue;}
wires.current()->clear();
//within same tile, route it here

if(tpoint(*wires.current()->from())==tpoint(*wires.current()->to()))
{
CNode* down = wires.current()->to();
CNode* up = wires.current()->from();
CEdge edge = down->edge();
    CLayerPair layerp = edge.layer();
assert(layerp.vl()!=NULL && layerp.hl()!=NULL);
wires.current()->append(
new gpath(CLayerPair(layer(up->layer()),NULL),grect(*up),grect(*up),true));
// wires.current()->append(
new gpath(layerp,grect(gpoint(*up)),grect(gpoint(*down))));
wires.current()->append(
new gpath(layerp,grect(*up),grect(*down)));
wires.current()->append(
new
gpath(CLayerPair(layer(down->layer()),NULL),grect(*down),grect(*down),true));
wires.current()->routed(
true);
wires.remove();
continue;
}
wires.next();
}

int numWires = wires.count();
std::cout << "routing " << numWires << " wires\n";
//std::cout << "routing2 "
<< this->wires().count() << " wires\n";

CPXENVptr env;
CPXLPptr lp;
double obj;
CPXbuildgo(env,lp,wires,fixes);
// double* x = CPXoptgo(env,lp);
CPXmipopt(env,lp);
double* x = new double[CPXgetnumcols(env,lp)];
CPXgetmipx(env,lp,x,0,CPXgetnumcols(env,lp)-1);
CPXgetmipobjval(env,lp,&obj);
int base = CPXgetnumcols(env,lp)-numBnds-1;

CPXchgprobtype(env,lp,CPXPROB_LP);
double* pi = new double[CPXgetnumrows(env,lp)];
CPXlpopt(env,lp);
CPXgetpi(env,lp,pi,0,CPXgetnumrows(env,lp)-1);
posFixes->clear();
//CPXwriteprob(env,lp,"last.lp",NULL);
for(int i=0;i<numBnds;i++)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// std::cout << "pi["<<i<<"("<<numWires+numBnds+i<<")]="
<< pi[numWires+numBnds+i] << "\n";
if(pi[numWires+numBnds+i]!=0)
{
//assert(fixes[i]<0);
posFixes->append( QPair<int,double>::QPair(i,x[base+i])
);
// std::cout << "possible fixes (" << base+i << ") "
 << i << " " << x[base+i] << "\n";
}
}


////////////////////////////////////////////////////////////////////
//static int qwer = 0;
//{QString name = "logdual"+QString().setNum(qwer)+".txt";
//QFile F(name);F.open(IO_WriteOnly);QTextStream f(&F);
//for(int i=0;i<CPXgetnumrows(env,lp);i++) f << i << "\t" <<  pi[i] <<
"\n";
//F.close();}
//{QString name = "logsol"+QString().setNum(qwer)+".txt";
//QFile F(name);F.open(IO_WriteOnly);QTextStream f(&F);
//for(int i=0;i<CPXgetnumcols(env,lp);i++) f << i <<  "\t" <<  x[i] <<
"\n";
//F.close();}
//CPXwriteprob(env,lp,"mip"+QString().setNum(qwer)+".lp",NULL);
//qwer++;
////////////////////////////////////////////////////////////////////

CPXfreeprob(env,&lp);
CPXcloseCPLEX(&env);


//read in from solution and lay in wires
int col = 0;
QValueListIterator<int> it;
QPtrListIterator<gwire> wIt(wires);
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
assert(!wire->routed());
CNode* down = wire->to();
CNode* up = wire->from();
CEdge edge = down->edge();
QPoint from = tpoint(*up);
QPoint to = tpoint(*down);
CLayerPair layerp = edge.layer();
assert(layerp.vl()!=NULL && layerp.hl()!=NULL);

wire->append(new gpath(CLayerPair(layer(up->layer()),NULL)
,grect(*up),grect(*up),true));
//first set of paths
CValueList<int> path1(from.x(),to.x(),false,true);
for(it=path1.begin();it!=path1.end();it++)
{
if(x[col]>0.5)
{
if(from.y()!=to.y())
{
//wire->append(
new gpath(layerp,grect(gpoint(*up)),grect(*it,from.y())));
wire->append(
new gpath(layerp,grect(*up),grect(*it,from.y())));
if(*it!=to.x()) wire->append(
new gpath(layerp,grect(*it,from.y()),grect(*it,to.y())));
else wire->append(
new gpath(layerp,grect(*it,from.y()),grect(*down)));
//if(*it!=to.x()) wire->append(
new gpath(layerp,grect(*it,to.y()),grect(gpoint(*down))));
if(*it!=to.x()) wire->append(
new gpath(layerp,grect(*it,to.y()),grect(*down)));
}else wire->append(
new gpath(layerp,grect(*up),grect(*down)));
}
col++;
if(from.y()==to.y()) break;//don't let more than one
}

//second set of paths
CValueList<int> path2(from.y(),to.y(),false,true);
for(it=path2.begin();it!=path2.end();it++)
{
if(x[col]>0.5)
{
if(from.x()!=to.x())
{
//wire->append(
new gpath(layerp,grect(gpoint(*up)),grect(from.x(),*it)));
wire->append
(new gpath(layerp,grect(*up),grect(from.x(),*it)));
if(*it!=to.y()) wire->append(
new gpath(layerp,grect(from.x(),*it),grect(to.x(),*it)));
else wire->append(
new gpath(layerp,grect(from.x(),*it),grect(*down)));
//if(*it!=to.y()) wire->append(
new gpath(layerp,grect(to.x(),*it),grect(gpoint(*down))));
if(*it!=to.y()) wire->append(
new gpath(layerp,grect(to.x(),*it),grect(*down)));
}else wire->append(
new gpath(layerp,grect(*up),grect(*down)));
}
col++;
if(from.x()==to.x()) break;//don't let more than one
}
wire->append(new gpath(CLayerPair(layer(down->layer()),NULL),
grect(*down),grect(*down),true));
wire->routed(true);
wire->incTileFlows(m_gTile);
}
delete[] x;
return pi;
}

void CGRoute::CPXbuildgo(CPXENVptr& env,CPXLPptr& lp,
QPtrList<gwire> wires, double* fixes) const
{
int numLayers = m_layers.size();
int numWires = wires.count();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
int numBnds = 0;
for(int i=0;i<numLayers;i++)
if(m_layers[i]->selected())
numBnds += m_layers[i]->numtTracks() *
m_layers[i]->numtEdges();

//initialize cplex env
int status;
env  = CPXopenCPLEX(&status);
if(status!=0){ std::cout << "Error : failed to initialize CPLEX\n";
return; }
// CPXsetintparam(env,CPX_PARAM_SCRIND,CPX_ON);
CPXsetdblparam(env,CPX_PARAM_EPGAP,0.1);
CPXsetdblparam(env,CPX_PARAM_EPAGAP,0.01);
CPXsetdblparam(env,CPX_PARAM_TILIM,5*60);
    lp = CPXcreateprob(env,&status,"go.lp");
CPXchgprobtype(env,lp,CPXPROB_MILP);
std::cout << "num Wires : " << numWires << "\n";
//initalize right hand side
int numRows = numWires +  2 * numBnds;
{
double* rhs = new double[numRows];
char* sense = new char[numRows];
for(int i=0;i<numWires;i++)
{
rhs[i] = 1;
sense[i]='E';
}
for(int i=0;i<numBnds;i++)
{
rhs[numWires+i] = 0;
sense[numWires+i] = 'E';

// rhs[numWires+numBnds+i] = (fixes[i]<0)? 0 : fixes[i];
rhs[numWires+numBnds+i] = 0;
sense[numWires+numBnds+i] = 'L';
// if(fixes[i]<0);else std::cout << "fixing "
<< i << " " << fixes[i] << "\n";
}
CPXnewrows(env,lp,numRows,rhs,sense,NULL,NULL);
delete[] rhs;
delete[] sense;
}

//fill in wire matrix
int col = 0; int row = 0;
double zero = 0;
double one = 1;
char binary = 'B';
char integer = 'I';
char continuous = 'C';
QPtrListIterator<gwire> wIt(wires);
QValueListIterator<int> it;
QValueList<CTileEdge> edges;
QValueListIterator<CTileEdge> eIt;
layerDirectionType dir;
int track,edge,bnd;
bnd = -1;
```

```
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
CNode* up = wire->from();
CNode* down = wire->to();
QPoint from = tpoint(*up);
QPoint to = tpoint(*down);
CLayerPair layerp = down->edge().layer();

int vl = layerp.vl()->index();
int hl = layerp.hl()->index();
int numhEdges = m_layers[hl]->numtEdges();
int hBase = 0;
for(int i=0;i<hl;i++) if(m_layers[i]->selected() ) hBase
+= m_layers[i]->numtTracks() * m_layers[i]->numtEdges();
int numvEdges = m_layers[vl]->numtEdges();
int vBase = 0;
for(int i=0;i<vl;i++) if(m_layers[i]->selected() ) vBase
+= m_layers[i]->numtTracks() * m_layers[i]->numtEdges();

//temporarly lay the wire
wire->append(new gpath(CLayerPair(layer(up->layer()),NULL)
,grect(*up),grect(*up)));
//first set of paths
CValueList<int> path1(from.x(),to.x(),false,true);
for(it=path1.begin();it!=path1.end();it++)
{
CPXnewcols(env,lp,1,&zero,&zero,&one,&binary,NULL);
CPXchgcoef(env,lp,row,col,1);
if(from.y()!=to.y())
{
//wire->append(new
gpath(layerp,grect(gpoint(*up)),grect(*it,from.y())));
wire->append(new
gpath(layerp,grect(*up),grect(*it,from.y())));
if(*it!=to.x()) wire->append(new
gpath(layerp,grect(*it,from.y()),grect(*it,to.y())));
else wire->append(new
gpath(layerp,grect(*it,from.y()),grect(*down)));
//if(*it!=to.x() ) wire->append(new
gpath(layerp,grect(*it,to.y()),grect(gpoint(*down))));
if(*it!=to.x() ) wire->append(new
gpath(layerp,grect(*it,to.y()),grect(*down)));
}else wire->append(new
gpath(layerp,grect(*up),grect(*down)));
edges = wire->edges(m_gTile);
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge e = *eIt;
dir = e.layer()->dir();
track = e.track();
edge = e.edge();
if(dir==HORIZONTAL) bnd = bnd = hBase + track *
numhEdges + edge;
else if(dir==VERTICAL) bnd = vBase + track *
numvEdges + edge;
CPXchgcoef(env,lp,numWires+bnd,col,1);
}
wire->clear();
```

```
col++;
if(from.y()==to.y()) break;//don't let more than one
}

//second set of paths
CValueList<int> path2(from.y(),to.y(),false,true);
for(it=path2.begin();it!=path2.end();it++)
{
CPXnewcols(env,lp,1,&zero,&zero,&one,&binary,NULL);
CPXchgcoef(env,lp,row,col,1);
if(from.x()!=to.x())
{
//wire->append(new
gpath(layerp,grect(gpoint(*up)),grect(from.x(),*it)));
wire->append(new
gpath(layerp,grect(*up),grect(from.x(),*it)));
if(*it!=to.y()) wire->append(new
gpath(layerp,grect(from.x(),*it),grect(to.x(),*it)));
else  wire->append(new
gpath(layerp,grect(from.x(),*it),grect(*down)));
//if(*it!=to.y()) wire->append(new
gpath(layerp,grect(to.x(),*it),grect(gpoint(*down))));
if(*it!=to.y()) wire->append(new
gpath(layerp,grect(to.x(),*it),grect(*down)));
}else wire->append(new
gpath(layerp,grect(*up),grect(*down)));
edges = wire->edges(m_gTile);
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge e = * eIt;
dir = e.layer()->dir();
track = e.track();
edge = e.edge();
if(dir==HORIZONTAL) bnd =  bnd = hBase + track *
numhEdges + edge;
else if(dir==VERTICAL)  bnd = vBase + track *
numvEdges + edge;
CPXchgcoef(env,lp,numWires+bnd,col,1);
}
wire->clear();
col++;
if(from.x()==to.x()) break;//don't let more than one
same path
}
row++;
}

//fill in layer matrix
for(int i=0;i<numLayers;i++)
{
if(m_layers[i]->selected() )
for(int track=0;track<m_layers[i]->numtTracks();track++)
{
for(int edge=0;edge<m_layers[i]->numtEdges();edge++)
{
double block = -
m_layers[i]->tileBlock(track,edge) - m_layers[i]->tileFlow(track,edge);

CPXnewcols(env,lp,1,&zero,&zero,NULL,&integer,NULL);//cnt,obj,lb,ub
CPXchgcoef(env,lp,row,col,-1);
CPXchgcoef(env,lp,row+numBnds,col,1);
CPXchgrhs(env,lp,1,&row,&block);
row++;
col++;
}
}
}

//fill in weights
CPXnewcols(env,lp,1,&one,NULL,NULL,&continuous,NULL);
int seg=0;
for(int i=0;i<numLayers;i++)
{
if(m_layers[i]->selected() )
for(int track=0;track<m_layers[i]->numtTracks();track++)
{
for(int edge=0;edge<m_layers[i]->numtEdges();edge++)
{
double cap = - m_layers[i]->tileCap();// cap =
1;
if(fixes[seg]<0)
CPXchgcoef(env,lp,row,col,cap);
else
CPXchgcoef(env,lp,row,col,-fixes[seg]);
//if(fixes[seg]<0);else std::cout
<< "fixing " << seg << " " << fixes[seg] << "\n";
seg++;
row++;
}
}
}
col++;
}

double* CGRoute::CPXoptgo(CPXENVptr& env,CPXLPptr& lp) const
{
int numRows = CPXgetnumrows(env,lp);
int numLayers = m_layers.size();
int numBnds = 0;
for(int i=0;i<numLayers;i++)
if(m_layers[i]->selected())
numBnds +=
m_layers[i]->numtTracks() * m_layers[i]->numtEdges();
int numWires = numRows-2*numBnds;

double objval,tryobjval,newobjval;
double* slack = new double[numBnds];
bool* done = new bool[numBnds];
for(int i=0;i<numBnds;i++)  done[i] = false;
bool optimum = false;
QValueList< QPair<int, double> > zeroList,notZeroList;
QValueListIterator< QPair<int, double> > it;
int colIndex = CPXgetnumcols(env,lp)-1;
double zero = 0;

/* do
{  //break;
CPXmipopt(env,lp);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CPXgetmipslack(env,lp,slack,numWires+numBnds,numRows-1);
CPXgetmipobjval(env,lp,&objval);
std::cout << "global optimization objective\t:"
 << objval << "\n";
int q =0;for(int i=0;i<numBnds;i++)
if(done[i])q++;std::cout << "DONE " << q << " " << numBnds << "\n";
if( objval<=0.5 ) optimum = true;
zeroList.clear();
notZeroList.clear();
//get zero-slack rows
for(int i=0;i<numBnds;i++)
{
if( done[i] ) continue;
if( slack[i] > 0.5 ) continue;
double maxCap;
CPXgetcoef(env,lp,numWires+numBnds+i,colIndex,&maxCap);
maxCap = -maxCap;
zeroList <<
QPair<int ,double>::QPair(i,maxCap); //save the row and capacity
done[i] = true;
}
std::cout << "Zero-slack "
<< zeroList.count() << "\n";
//fix the zero slack flows
for(it=zeroList.begin();it!=zeroList.end();it++)
{
int right = numWires +numBnds+ (*it).first;

CPXchgcoef(env,lp,numWires+numBnds+(*it).first,colIndex,0);
double robjval = ceil(objval * (*it).second-0.5);
CPXchgrhs(env,lp,1,&right,&robjval);
}
//push each zero slack row down  a little
for(it=zeroList.begin();it!=zeroList.end();it++)
{
int right = numWires +numBnds+ (*it).first;
tryobjval = ceil(objval * (*it).second-0.5) - 1;
CPXchgrhs(env,lp,1,&right,&tryobjval);
CPXmipopt(env,lp);
          if(CPXgetstat(env,lp)==CPXMIP_OPTIMAL)
{//not done, still can go down, remove from list if
objectiv ev better
CPXgetmipobjval(env,lp,&newobjval);
if( ceil((*it).second*newobjval-0.5)
< ceil((*it).second*objval-0.5) ) notZeroList << *it;
}
//put back original
double robjval = ceil(objval * (*it).second-0.5);
CPXchgrhs(env,lp,1,&right,&robjval);
}
if(notZeroList.count()==zeroList.count())
{
//all zero slacks fake, accept all
notZeroList.clear();
}
for(it=notZeroList.begin();it!=notZeroList.end();it++)
{// put back original for fake zero slack
int right = numWires+numBnds+(*it).first;
done[(*it).first] = false;
CPXchgcoef(env,lp,numWires+
numBnds+(*it).first,colIndex,-(*it).second);
CPXchgrhs(env,lp,1,&right,&zero);
}
}while(!optimum);*/
/* QTime t;
t.start();
int c=0;
double nextobjval = 500;
double* qwer = new double[numBnds];
for(int i=0;i<numBnds;i++)
{
double maxCap;
CPXgetcoef(env,lp,numWires+numBnds+
i,colIndex,&maxCap);
qwer[i] = -maxCap;
}
 do
{  //break;
CPXwriteprob(env,lp,"",NULL);exit(0);
CPXmipopt(env,lp);
CPXgetmipslack(env,lp,slack,numWires+
numBnds,numRows-1);
CPXgetmipobjval(env,lp,&objval);

if(objval<nextobjval)
{
nextobjval = objval-0.1;
QString name = QString().setNum(c++)+"_"+
QString().setNum(objval)+".log";
//CPXwriteprob(env,lp,name+".lp",NULL);
QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);
double* x = new double[CPXgetnumcols(env,lp)];
CPXgetmipx(env,lp,x,0,CPXgetnumcols(env,lp)-1);
f << "objective = " << objval << "\n";
f << "time = " << t.elapsed() << "\n";
int base = CPXgetnumcols(env,lp)-numBnds-1;
int overflow = 0;
for(int i=0;i<numBnds;i++)
{
if(x[base+i]>qwer[i]) overflow += x[base+i]-qwer[i];
}
f << "overflow = " << overflow << "\n";
for(int i=0;i<numBnds;i++)
{
f << i << "\t" << x[base+i]/-qwer[i] << "\n";
}

delete[] x;
F.close();
}


std::cout << "global optimization objective\t:"
<< objval << "\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
int q =0;
for(int i=0;i<numBnds;i++) if(done[i])q++;
std::cout << q << " segment has been optimized out of "
<< numBnds << " \n";
if( objval<=0.1 ) {optimum = true;break;}
zeroList.clear();
notZeroList.clear();
//get zero-slack rows
//or minimum slacsk
double minSlack = 2*INF;
for(int i=0;i<numBnds;i++)
{
if( done[i] ) continue;
if( slack[i] < minSlack )
minSlack = slack[i];
}
if(minSlack>=INF) std::cout
<< " Error, can't find minimum slack\n";
assert(minSlack<INF);
for(int i=0;i<numBnds;i++)
{
if( done[i] ) continue;
if( slack[i] > minSlack + 0.5 ) continue;
double maxCap;
CPXgetcoef(env,lp,numWires+numBnds+i,colIndex,&maxCap);
maxCap = -maxCap;
zeroList << QPair<int ,double>::QPair(i,maxCap);
 //save the row and capacity
done[i] = true;
}
std::cout << zeroList.count()
<< " segments has zero slack (" << minSlack << ") : ";std::cout.flush();
if( zeroList.count()<=0) {std::cout
<< " Error there is no zero slack segments : ";std::cout.flush();}
assert( zeroList.count()>0 );
//fix the zero slack flows
for(it=zeroList.begin();it!=zeroList.end();it++)
{
int right = numWires +numBnds+ (*it).first;

CPXchgcoef(env,lp,numWires+numBnds+(*it).first,colIndex,0);
double robjval = ceil(objval * (*it).second-0.5);
CPXchgrhs(env,lp,1,&right,&robjval);
}
//push each zero slack row down  a little
for(it=zeroList.begin();it!=zeroList.end();it++)
{
int right = numWires +numBnds+ (*it).first;
tryobjval = ceil(objval * (*it).second-0.5) - 1;
CPXchgrhs(env,lp,1,&right,&tryobjval);
CPXmipopt(env,lp);
          if(CPXgetstat(env,lp)==CPXMIP_OPTIMAL)
{//not done, still can go down, remove from list if
objectiv ev better
CPXgetmipobjval(env,lp,&newobjval);
if( ceil((*it).second*newobjval-0.5)
< ceil((*it).second*objval-0.5) )
{
notZeroList << *it;
std::cout << "*";
}
else std::cout << ".";
} else std::cout << ".";std::cout.flush();
//put back original
double robjval = ceil(objval * (*it).second-0.5);
CPXchgrhs(env,lp,1,&right,&robjval);
}
if(notZeroList.count()==zeroList.count())
{
//all zero slacks fake, accept all
notZeroList.clear();
std::cout << " all accepted";
}
std::cout << "\n";
for(it=notZeroList.begin();it!=notZeroList.end();it++)
{// put back original for fake zero slack
int right = numWires+numBnds+(*it).first;
done[(*it).first] = false;

CPXchgcoef(env,lp,numWires+numBnds+(*it).first,colIndex,-(*it).second);
CPXchgrhs(env,lp,1,&right,&zero);
}
}while(!optimum);*/
CPXmipopt(env,lp);
double* x = new double[CPXgetnumcols(env,lp)];
CPXgetmipx(env,lp,x,0,CPXgetnumcols(env,lp)-1);

delete[] slack;
delete[] done;
return x;
}
```

```
#include <qpixmap.h>
#include <qpainter.h>
#include "cgroute.h"
#include "clayer.h"
#include "cnet.h"

bool CGRoute::saveBlk(const QString fileName)
{
QFile blkFile(fileName);
if(!blkFile.open(IO_WriteOnly))
```

```
{
std::cout << "Error : couldn't opened " << fileName << "\n";
return false;
}
QTextStream blk(&blkFile);

for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
blk << "set layer " << pLayer->name() << " usage ";
blk <<  pLayer->len() << " " << pLayer->bUsage() << "\n";
   }
for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* pLayer = m_layers[i];
for(int track=0;track<pLayer->numcTracks();track++)
for(int edge=0;edge<pLayer->numcEdges();edge++)
{
blk << "set layer " << pLayer->name()
 << " cellBlock ";
blk << track <<  " " << edge << " "
 << pLayer->cellBlock(track,edge) << "\n";
}
for(int track=0;track<pLayer->numtTracks();track++)
for(int edge=0;edge<pLayer->numtEdges();edge++)
{
blk << "set layer " << pLayer->name()
<< " tileBlock ";
blk << track <<  " " << edge << " "
 << pLayer->tileBlock(track,edge) << "\n";
}
}
blkFile.close();
return true;
}

bool CGRoute::saveFlow(const QString fileName)
{
QFile flwFile(fileName);
if(!flwFile.open(IO_WriteOnly))
{
std::cout << "Error : couldn't opened "
<< fileName << "\n";
return false;
}
QTextStream flw(&flwFile);

for(unsigned int i=0;i<m_layers.size();i++)
{
flw << "Cells\n";
CLayer* pLayer = m_layers[i];
for(int track=0;track<pLayer->numcTracks();track++)
for(int edge=0;edge<pLayer->numcEdges();edge++)
{
int b = pLayer->cellBlock(track,edge);
int f = pLayer->cellFlow(track,edge);
int c = pLayer->cellCap();
if(b+f>c) flw << "* ";
flw << pLayer->name();
flw << " " << track <<  " " << edge;
flw << " " << b << " + " << f << " = " << b+f <<
" <= "  << c << "\n";
}
flw << "Tiles\n";
for(int track=0;track<pLayer->numtTracks();track++)
for(int edge=0;edge<pLayer->numtEdges();edge++)
{
int b = pLayer->tileBlock(track,edge);
int f = pLayer->tileFlow(track,edge);
int c = pLayer->tileCap();
if(b+f>c) flw << "* ";
flw << pLayer->name();
flw << " " << track <<  " " << edge;
flw << " " << b << " + " << f << " = " << b+f <<
" <= " << c << "\n";
}
}
flwFile.close();
return true;
}

bool CGRoute::rteWrite(const QString fileName) const
{
QFile file(fileName);
if(!file.open(IO_WriteOnly | IO_Raw))
{
std::cout << "Error : couldn't opened " << fileName << "\n";
return false;
}
   QTextStream f(&file);
f << "(routes\n";
   f<<"(resolution um 1000)\n";
   f<<"(parser\n";
  f<<")\n";
   f<<"(structure_out\n";
for(unsigned int i=0;i<m_layers.size();i++)
{
    f<<"(layer " << m_layers[i]->name() << " (type signal)\n";
    f<<")\n";
   }
   f<< ")\n";
   f<< "(library_out\n";
for(unsigned int i=0;i<m_layers.size()-1;i++)
{
CLayer* pL1=m_layers[i];
CLayer* pL2=m_layers[i+1];
   f<<"(via_image ZC" << pL1->name() << pL2->name() << "\n";
    f<<"(shape (rect " << pL1->name();
f<< " " << -1000.0*pL1->width()/2.0 << " "  <<
-1000.0*pL1->width()/2.0;
f<< " " << 1000.0*pL1->width()/2.0 << " "  <<
1000.0*pL1->width()/2.0 << "))\n";
f<<"(shape (rect " << pL2->name();
f<< " " << -1000.0*pL2->width()/2.0 << " "  <<
-1000.0*pL2->width()/2.0;
f<< " " << 1000.0*pL2->width()/2.0 << " "  <<
1000.0*pL2->width()/2.0 << "))\n";
f<< ")\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
f<< ")\n";
    f << "(network_out\n";
QDictIterator<CNet> it(m_nets);
for(it.toFirst();it.current();++it) (*it)->rteWrite(f);
f << ")\n)\n";
file.close();
return true;
}


bool CGRoute::grteWrite(const QString fileName) const
{
QFile file(fileName);
if(!file.open(IO_WriteOnly | IO_Raw))
{
std::cout << "Error : couldn't opened " << fileName << "\n";
return false;
}
   QTextStream f(&file);
QDictIterator<CNet> it(m_nets);
for(it.toFirst();it.current();++it) (*it)->grteWrite(f);
file.close();
return true;
}

bool CGRoute::savePic(const QString fileName, const QString netName) const
{
QPixmap pic(int(m_dieArea.width()/10),int(m_dieArea.height()/10));
QPainter p;
p.begin(&pic);
p.setBackgroundColor("white");
p.eraseRect(0,0,int(m_dieArea.width()/10),int(m_dieArea.height()/10));
for(int i=0;i<pic.width();i++)
p.drawText(i*10,10,QString().setNum(i%10));
for(int i=0;i<pic.height();i++)
p.drawText(0,i*10+10,QString().setNum(i%10));
//layer now
for(unsigned int i=0;i<m_layers.size();i++) if(m_layers[i]->selected())
m_layers[i]->paint(&p);
/// net now
m_nets[netName]->paint(&p);
p.end();
pic.save(fileName,"BMP");
return true;
}


#include "cgroute.h"
#include "cnode.h"
#include "gwire.h"
#include "clayer.h"
#include "ctile.h"

int CGRoute::ripupwires(double* pi, QPtrList<gwire>& wires)
{
int numRipups = 0;
static int c = 0;
//QFile F(""+QString().setNum(c++)+".txt");
//F.open(IO_WriteOnly);
//QTextStream f(&F);

//f << "num Wires : " << wires.count() << "\n";
QPtrListIterator<gwire> wIt(wires);
int indx = 0;
for(wIt.toFirst();wIt.current();++wIt)
{
if( pi[indx++] == 0 ) continue;
//f << indx-1 << "\t" << pi[indx-1] << "\n";
gwire* wire = *wIt;
assert(wire->routed());

wire->decTileFlows(m_gTile);
wire->clear();
wire->routed(false);
numRipups++;
/*
QValueList<CTileEdge> edges = wire->edges(m_gTile);
QValueListIterator<CTileEdge> it;
for(it=edges.begin();it!=edges.end();it++)
{
//any overflowing
if(
(*it).layer()->tileOverflows((*it).track(),(*it).edge()) )
{
wire->decTileFlows(m_gTile);
wire->clear();
wire->routed(false);
numRipups++;
break;
}
}*/
}
//F.close();
std::cout << "Num Ripup : " << numRipups << "\n";
return numRipups;
}
/*
int CGRoute::ripupwires()
{
int numRipups = 0;
QPtrList<gwire> wires = this->wires();

QPtrListIterator<gwire> wIt(wires);
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
assert(wire->routed());
QValueList<CTileEdge> edges = wire->edges(m_gTile);
QValueListIterator<CTileEdge> it;
for(it=edges.begin();it!=edges.end();it++)
{
//any overflowing
if(
(*it).layer()->tileOverflows((*it).track(),(*it).edge()) )
{
```

```
wire->decTileFlows(m_gTile);
wire->clear();
wire->routed(false);
numRipups++;
break;
}
}
}
return numRipups;
}
*/
void CGRoute::lawires()
{
   //get unrouted wires
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
if(wires.current()->routed()) {wires.remove();continue;}
wires.current()->clear();
wires.next();
}
std::cout << "applying QP layer assignment algorithm for " <<
wires.count() << " wires\n";
// lawires(wires);
QPtrList<gwire> subwires;
wires.first();
while(wires.current())
{
subwires.append( wires.current());
if(subwires.count()>=500){lawires(subwires);subwires.clear();
}
wires.next();
}
if(subwires.count()>0) lawires(subwires);
}


void CGRoute::lawires(QPtrList<gwire> wires)
{

QValueVector< QPair<double,double> > ratio;
  QValueVector< QValueVector<double> > b = getBlockagesgo(ratio,wires);
QMap< QPair<int,int>, double> w = getInterferencego(wires);

int status;
CPXENVptr env  = CPXopenCPLEX(&status);
// CPXsetintparam(env,CPX_PARAM_SCRIND,CPX_ON);
CPXsetdblparam(env,CPX_PARAM_EPGAP,0.1);
    CPXLPptr lp = CPXcreateprob(env,&status,"lago.miqp");
CPXchgprobtype(env,lp,CPXPROB_MIQP);

int numLayers = b[0].size()-1;
int numEdges = b.size();
int numCols = numLayers * numEdges;
int numRows = numEdges;

double* obj = new double[numCols];
double* rhs = new double[numRows];
char* sense = new char[numRows];
int* matbeg = new int[numCols];
int* matcnt = new  int[numCols];
int* matind = new int[numCols];
double* matval = new double[numCols];
double* lb = new double[numCols];
double* ub = new double[numCols];
char* ctype = new char[numCols];
int* qmatbeg = new int[numCols];
int* qmatcnt = new int[numCols];
int* qmatind = new int[(2*w.count()+numRows)*
(numLayers+2*(numLayers-1))];
double* qmatval = new double[(2*w.count()+numRows)*
(numLayers+2*(numLayers-1))];
double* x = new double[numCols];

int colIndexBase,rowIndexBase,colIndex,rowIndex;

for(int i=0;i<numRows;i++)
{
rhs[i] = 1;
sense[i]='E';
colIndexBase = i * numLayers;
for(int j=0;j<numLayers;j++)
{
colIndex = colIndexBase + j;
if(m_layers[j]->dir() == VERTICAL )
obj[colIndex] = ratio[i].second * b[i][j] +
ratio[i].first * b[i][j+1];
else
obj[colIndex] = ratio[i].first * b[i][j] +
ratio[i].second * b[i][j+1];
matbeg[colIndex] = colIndex;
matcnt[colIndex] = 1;
matind[colIndex] = i;
matval[colIndex] = 1;
lb[colIndex] = 0;
ub[colIndex] = (m_layers[j]->selected())? 1:0;
ctype[colIndex] = 'B';
}
}
b.clear();
CPXcopylp(env,lp,numCols,numRows,1,obj,rhs,sense,matbeg,matcnt,matind,
matval,lb,ub,NULL);
CPXcopyctype(env,lp,ctype);
delete[] obj;
delete[] rhs;;
delete[] sense;
delete[] matbeg;
delete[] matcnt;
delete[] matind;
delete[] matval;
delete[] lb;
delete[] ub;
delete[] ctype;

   int index = 0;
int cnt;
double wc = 0;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    for(int i=0;i<numRows;i++)
{
colIndexBase = i * numLayers;
for(int j=0;j<numLayers;j++)
{
colIndex = colIndexBase+j;
qmatbeg[colIndex] = index;
cnt = 0;
for(int k=0;k<numRows;k++)
{
rowIndexBase = k * numLayers;
if( w.contains( QPair<int,int>::QPair(i,k) ) )
wc = w[ QPair<int,int>::QPair(i,k) ];
else if( w.contains( QPair<int,int>::QPair(k,i)
) ) wc = w[ QPair<int,int>::QPair(k,i) ];
else if( i!=k) continue;
for(int l=0;l<numLayers;l++)
{
rowIndex = rowIndexBase + l;
if( i==k )
{
if( j==l )
{
                        qmatind[ index ] =
rowIndex;
       qmatval[ index ] =
2.0/double(numEdges);
index++;
cnt++;
}
}else if( j==l )
{
qmatind[ index ] = rowIndex;
       qmatval[ index ] =
wc/double(numEdges);
index++;
cnt++;
}else if( abs(j-l)==1 )
{
       qmatind[ index ] = rowIndex;
       qmatval[ index ] =
wc/double(2*numEdges);
index++;
cnt++;
}
}
}
qmatcnt[colIndex] = cnt;
}
}
w.clear();
CPXcopyquad (env,lp,qmatbeg,qmatcnt,qmatind,qmatval);
delete[] qmatbeg;
delete[] qmatcnt;
delete[] qmatind;
delete[] qmatval;

//CPXwriteprob(env,lp,"lago.lp",NULL);
CPXmipopt(env,lp);
```

```
CPXgetmipx(env,lp,x,0,numCols-1);

for(int i=0;i<numRows;i++)
{
colIndexBase = i*numLayers;
for(int j=0;j<numLayers;j++)
{
colIndex = j + colIndexBase;
if( x[colIndex] > 0.5 )
wires.at(i)->to()->edge().layer(CLayerPair(
m_layers[j],m_layers[j]->next(false) ));
}
}
CPXfreeprob(env,&lp);
CPXcloseCPLEX(&env);
}

QValueVector< QValueVector<double> > CGRoute::getBlockagesgo( QValueVector<
QPair<double,double> >& ratio,


QPtrList<gwire> wires) const
{
int numWires = wires.count();
int numLayers = 0;
for(unsigned int i=0;i<m_layers.size();i++) numLayers++;
    QValueVector< QValueVector<double> > blockages;

blockages.resize(numWires);
for(int i=0;i<numWires;i++) blockages[i].resize(numLayers,0);
ratio.resize(numWires);

int index=0;
CRect bBox;
CLayer* layer;
QPtrListIterator<gwire> wIt(wires);
int totCap,totBlock,halfPerimeter;

for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
QPoint from = tpoint(*wire->from());
QPoint to = tpoint(*wire->to());
QRect r = QRect(from,to).normalize();
for(int i=0;i<numLayers;i++)
{
   layer = m_layers[i];
totCap = 0;
totBlock = 0;
switch(layer->dir())
{
case VERTICAL :
for(int i=r.top()+1;i<=r.bottom();i++)
for(int j=r.left();j<=r.right();j++)
{
totCap += layer->tileCap();
totBlock +=layer->tileBlock( i, j ) +
layer->tileFlow( i, j );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
break;
case HORIZONTAL :
                    for(int i=r.left()+1;i<=r.right();i++)
for(int j=r.top();j<=r.bottom();j++)
{
totCap += layer->tileCap();
totBlock += layer->tileBlock( i, j ) +
layer->tileFlow( i, j );
}
break;
default: std::cout << "CGRoute::getBlockagesgo() not
implemented case \n";
}
if( (totCap==0) || (totBlock==0) ) blockages[index][i] =
0.0;
   else blockages[index][i] = double(totBlock)/double(totCap);
}
halfPerimeter = r.right()-r.left()+r.bottom()-r.top();
if( halfPerimeter==0 )
{
ratio[index].first = 0.0;
ratio[index].second = 0.0;
}else
{
       ratio[index].first =
double(r.right()-r.left())/double(r.right()-r.left()+r.bottom()-r.top());
ratio[index].second =
double(r.bottom()-r.top())/double(r.right()-r.left()+r.bottom()-r.top());
}
index++;
}
return blockages;
}

QMap< QPair<int,int>, double> CGRoute::getInterferencego(QPtrList<gwire> wires)
const
{
QMap< QPair<int,int>, double> inter;
int index1=0,index2=0;
CRect b1,b2;
QPtrListIterator<gwire> wIt1(wires);
QPtrListIterator<gwire> wIt2(wires);
for(wIt1.toFirst();wIt1.current();++wIt1)
{
index2 = index1;
b1 = CRect(*(*wIt1)->from(),*(*wIt1)->to()).normalize();
for(wIt2.toFirst();wIt2.current();++wIt2)
{
b2 = CRect(*(*wIt2)->from(),*(*wIt2)->to()).normalize();
if(index1!=index2 && b1.intersects(b2) )
inter[ QPair<int,int>::QPair(index1,index2) ] =
(b1 & b2 ).area() / (b1 | b2).area();
index2++;
}
index1++;
}
return inter;
}
```

```
#include "cgroute.h"
#include "clayer.h"
#include "gwire.h"
#include "gpath.h"
#include "ctile.h"

int CGRoute::ripuppaths()
{
int numRipups = 0;
QPtrList<gwire> wires = this->wires();

QPtrListIterator<gwire> wIt(wires);
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
assert(wire->routed());
QPtrListIterator<gpath> pIt(*wire);
for(pIt.toFirst();pIt.current();++pIt)
{
gpath* path = *pIt;
assert(path->routed());
    QValueList<CTileEdge> edges = path->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
 if(
(*eIt).layer()->cellOverflows((*eIt).track(),(*eIt).edge()) )
{
path->decCellFlows();
path->routed(false);
// path->from( grect( path->from().center()
) );//blow out from and to to gTiles
// path->to( grect( path->to().center() )
);
path->layer( CLayerPair() );//no layer
assgined yet
numRipups++;
break;
}
}
}
wire->mergeExpandUnroutedPaths(m_gTile);
}
return numRipups;
}

void CGRoute::lapaths()
{
   //get unrouted wires
QPtrList<gwire> wires = this->wires();
QValueList< QPair<gwire*,gpath*> > paths;
wires.first();
while(wires.current())
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
wires.current()->first();
while(wires.current()->current())
{
if(!wires.current()->current()->routed())
paths <<
QPair<gwire*,gpath*>::QPair(wires.current(),wires.current()->current());
wires.current()->next();
}
wires.next();
}
std::cout << "applying QP layer assignment algorithm for " <<
paths.count() << " wires\n";
// lapaths(paths);
QValueList<QPair<gwire*,gpath*> > subpaths;
while(!paths.isEmpty())
{
subpaths.append( paths.front() );
paths.pop_front();
if(subpaths.count()>=1000){lapaths(subpaths);subpaths.clear();
}
}
if(subpaths.count()>0) lapaths(subpaths);
}


void CGRoute::lapaths(QValueList<QPair<gwire*,gpath*> > paths)
{

QValueVector< QPair<double,double> > ratio;
  QValueVector< QValueVector<double> > b = getBlockagesnet(ratio,paths);
QMap< QPair<int,int>, double> w = getInterferencenet(paths);

int status;
CPXENVptr env  = CPXopenCPLEX(&status);
// CPXsetintparam(env,CPX_PARAM_SCRIND,CPX_ON);
CPXsetdblparam(env,CPX_PARAM_EPGAP,0.1);
    CPXLPptr lp = CPXcreateprob(env,&status,"lago.miqp");
CPXchgprobtype(env,lp,CPXPROB_MIQP);

int numLayers = b[0].size()-1;
int numEdges = b.size();
int numCols = numLayers * numEdges;
int numRows = numEdges;

double* obj = new double[numCols];
double* rhs = new double[numRows];
char* sense = new char[numRows];
int* matbeg = new int[numCols];
int* matcnt = new  int[numCols];
int* matind = new int[numCols];
double* matval = new double[numCols];
double* lb = new double[numCols];
double* ub = new double[numCols];
char* ctype = new char[numCols];
int* qmatbeg = new int[numCols];
int* qmatcnt = new int[numCols];
int* qmatind = new int[(2*w.count()+numRows)*
(numLayers+2*(numLayers-1))];
double* qmatval = new double[(2*w.count()+numRows)*
(numLayers+2*(numLayers-1))];
double* x = new double[numCols];

int colIndexBase,rowIndexBase,colIndex,rowIndex;

for(int i=0;i<numRows;i++)
{
rhs[i] = 1;
sense[i]='E';
colIndexBase = i * numLayers;
for(int j=0;j<numLayers;j++)
{
colIndex = colIndexBase + j;
if(m_layers[j]->dir() == VERTICAL )
obj[colIndex] = ratio[i].second * b[i][j] +
ratio[i].first * b[i][j+1];
else
obj[colIndex] = ratio[i].first * b[i][j] +
ratio[i].second * b[i][j+1];
matbeg[colIndex] = colIndex;
matcnt[colIndex] = 1;
matind[colIndex] = i;
matval[colIndex] = 1;
lb[colIndex] = 0;
ub[colIndex] = (m_layers[j]->selected())? 1:0;
ctype[colIndex] = 'B';
}
}
b.clear();
CPXcopylp(env,lp,numCols,numRows,1,obj,rhs,sense,matbeg,matcnt,matind,
matval,lb,ub,NULL);
CPXcopyctype(env,lp,ctype);
delete[] obj;
delete[] rhs;;
delete[] sense;
delete[] matbeg;
delete[] matcnt;
delete[] matind;
delete[] matval;
delete[] lb;
delete[] ub;
delete[] ctype;

   int index = 0;
int cnt;
double wc = 0;
   for(int i=0;i<numRows;i++)
{
colIndexBase = i * numLayers;
for(int j=0;j<numLayers;j++)
{
colIndex = colIndexBase+j;
qmatbeg[colIndex] = index;
cnt = 0;
for(int k=0;k<numRows;k++)
{
rowIndexBase = k * numLayers;
if( w.contains( QPair<int,int>::QPair(i,k) ) )
wc = w[ QPair<int,int>::QPair(i,k) ];
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
else if( w.contains( QPair<int,int>::QPair(k,i)
) ) wc = w[ QPair<int,int>::QPair(k,i) ];
else if( i!=k) continue;
for(int l=0;l<numLayers;l++)
{
rowIndex = rowIndexBase + l;
if( i==k )
{
if( j==l )
{
                    qmatind[ index ] = rowIndex;
      qmatval[ index ] =
2.0/double(numEdges);
index++;
cnt++;
}
}else if( j==l )
{
qmatind[ index ] = rowIndex;
      qmatval[ index ] = wc/double(numEdges);
index++;
cnt++;
}else if( abs(j-l)==1 )
{
      qmatind[ index ] = rowIndex;
       qmatval[ index ] =
wc/double(2*numEdges);
index++;
cnt++;
}
}
}
qmatcnt[colIndex] = cnt;
}
}
w.clear();
CPXcopyquad (env,lp,qmatbeg,qmatcnt,qmatind,qmatval);
delete[] qmatbeg;
delete[] qmatcnt;
delete[] qmatind;
delete[] qmatval;

// CPXwriteprob(env,lp,"lanet.lp",NULL);
CPXmipopt(env,lp);
CPXgetmipx(env,lp,x,0,numCols-1);

for(int i=0;i<numRows;i++)
{
colIndexBase = i*numLayers;

for(int j=0;j<numLayers;j++)
{
colIndex = j + colIndexBase;
if( x[colIndex] > 0.5 );
paths[i].first->movePath( paths[i].second,
CLayerPair( m_layers[j],m_layers[j]->next(false) ) );
}
}
CPXfreeprob(env,&lp);
```

```
CPXcloseCPLEX(&env);
}

QValueVector< QValueVector<double> > CGRoute::getBlockagesnet( QValueVector<
QPair<double,double> >& ratio,


QValueList< QPair<gwire*,gpath*> > paths) const
{
int numWires = paths.count();
int numLayers = 0;
for(unsigned int i=0;i<m_layers.size();i++) numLayers++;
  QValueVector< QValueVector<double> > blockages;

blockages.resize(numWires);
for(int i=0;i<numWires;i++) blockages[i].resize(numLayers,0);
ratio.resize(numWires);

int index=0;
CRect bBox;
CLayer* layer;
QValueListIterator< QPair<gwire*,gpath*> > wIt;
int totCap,totBlock,halfPerimeter;

for(wIt=paths.begin();wIt!=paths.end();wIt++)
{
// gwire* wire = (*wIt).first;
gpath* path = (*wIt).second;
QRect r = *path;
for(int i=0;i<numLayers;i++)
{
   layer = m_layers[i];
totCap = 0;
totBlock = 0;
switch(layer->dir())
{
case VERTICAL :
for(int
i=r.top()+1;i<=r.bottom();i++)
for(int
j=r.left();j<=r.right();j++)
{
totCap +=
layer->cellCap();
totBlock
+=layer->cellBlock( i, j ) + layer->cellFlow( i, j );
}
break;
case HORIZONTAL :
                   for(int i=r.left()+1;i<=r.right();i++)
for(int
j=r.top();j<=r.bottom();j++)
{
totCap +=
layer->cellCap();
totBlock +=
layer->cellBlock( i, j ) + layer->cellFlow( i, j );
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
break;
default: std::cout << "CGRoute::getBlockagesnet() not
implemented case\n";
}
if( (totCap==0) || (totBlock==0) ) blockages[index][i] =
0.0;
   else blockages[index][i] = double(totBlock)/double(totCap);
}
halfPerimeter = r.right()-r.left()+r.bottom()-r.top();
if( halfPerimeter==0 )
{
ratio[index].first = 0.0;
ratio[index].second = 0.0;
}else
{
       ratio[index].first =
double(r.right()-r.left())/double(r.right()-r.left()+r.bottom()-r.top());
ratio[index].second =
double(r.bottom()-r.top())/double(r.right()-r.left()+r.bottom()-r.top());
}
index++;
}
return blockages;
}

QMap< QPair<int,int>, double>
CGRoute::getInterferencenet(QValueList<QPair<gwire*,gpath*> > paths) const
{
QMap< QPair<int,int>, double> inter;
int index1=0,index2=0;
CRect b1,b2;
QValueListIterator< QPair<gwire*,gpath*> > wIt1,wIt2;
for(wIt1=paths.begin();wIt1!=paths.end();wIt1++)
{
index2 = index1;
b1 = *(*wIt1).second;
for(wIt2=paths.begin();wIt2!=paths.end();wIt2++)
{
b2 = *(*wIt2).second;
if(index1!=index2 && b1.intersects(b2) )
inter[ QPair<int,int>::QPair(index1,index2) ] =
(b1 & b2 ).area() / (b1 | b2).area();
index2++;
}
index1++;
}
return inter;
}


#include "cgroute.h"
#include "gwire.h"
#include "gpath.h"
#include "clayer.h"
#include "cpathcut.h"

void CGRoute::net()
{
//QFile file("diag.log");
//file.open( IO_WriteOnly );
//QTextStream f(&file);

int pass = 1;
std::cout << "network optimization\n";
std::cout << "pass " << pass++ <<"\n";
netmain();//return;
//f << numCellOverflows() << "\n";
pp();
//f << numCellOverflows() << "\n";
int prevNumOver;
int numOver = numCellOverflows();
std::cout << numOver << " overflows\n";
do
{
prevNumOver = numOver;
if(ripuppaths()<=0) break;
std::cout << "pass " << pass++ << "\n";
  lapaths();
netmain();
//f << numCellOverflows() << "\n";
pp();
//f << numCellOverflows() << "\n";
numOver = numCellOverflows();
std::cout << numOver << " overflows\n";
}while(numOver>0 && numOver<prevNumOver);

//file.close();
}

void CGRoute::netmain()
{
//get unrouted paths
int numLayers = m_layers.size();
QPtrList<gwire> wires = this->wires();
QPtrListIterator<gwire> wIt(wires);
int numWires = 0;
for(wIt.toFirst();wIt.current();++wIt)
{
QPtrListIterator<gpath> pIt(**wIt);
for(pIt.toFirst();pIt.current();++pIt)
{
gpath* path = *pIt;
if(path->routed()) continue;
if(path->from().center()==path->to().center() &&
path->from().size()==QSize(1,1) &&
path->to().size()==QSize(1,1))
{
path->routed(true);
continue;
}
numWires++;
}
}
std::cout << "routing " << numWires << " wires\n";

QValueList<QPair<layerDirectionType,int> > cuts = this->cuts();
while(!cuts.isEmpty())
```

```
{
//std::cout << "CUTS " << cuts.count() << "\n";
QPair<layerDirectionType,int> cut =
cuts.front();cuts.pop_front();
layerDirectionType dir = cut.first;
int track = cut.second;
std::cout << layerDirection(dir) << " bisect at track " << track
<< ", cuts left "<< cuts.count() << "\n";
for(int l=0;l<numLayers;l++)
{
CLayer* layer = m_layers[l];
if(!layer->selected()) continue;
if(layer->dir()==dir) continue;
CPXENVptr env;
CPXNETptr net;
QValueList<CPathCut> cutpaths =
CPXbuildnet(env,net,layer,track);
int* x =
CPXoptnet(env,net,layer->numcEdges(),layer->cellCap());
QValueListIterator<CPathCut> xIt;
int i = 0;
for(xIt=cutpaths.begin();xIt!=cutpaths.end();xIt++)
{
gwire* wire = (*xIt).wire();
wire->rewire( (*xIt).paths(), layer, track, x[i]
);
i++;
}
cutpaths.clear();
delete[] x;
}
}
for(wIt.toFirst();wIt.current();++wIt) (*wIt)->mergeStraightPaths();
}

QValueList<QPair<layerDirectionType,int> > CGRoute::cuts() const
{
int minv = 0;int maxv = (m_layers[0]->dir()==HORIZONTAL)?
m_layers[0]->numcTracks() : m_layers[1]->numcTracks();
int minh = 0;int maxh = (m_layers[0]->dir()==VERTICAL)?
m_layers[0]->numcTracks() : m_layers[1]->numcTracks();

QValueList<int> verticalSequence,horizontalSequence;
QValueList< QPair< layerDirectionType, int> > cuts;

verticalSequence << minv << maxv-1;
horizontalSequence << minh << maxh-1;
bool iterated = false;
do
{
iterated = false;
QValueList<int> list;
QValueListIterator<int> x=verticalSequence.end();
QValueListIterator<int> y=verticalSequence.end();
y--;
while(y!=verticalSequence.begin())
{
x--;
y--;
if( *x!=*y+1 )
{
list << (int)floor( (*x+*y)/2 );
cuts << QPair<layerDirectionType, int>::QPair(
VERTICAL,(int)floor( (*x+*y)/2 ));
iterated = true;
}
}
verticalSequence += list;
qHeapSort( verticalSequence );

list.clear();
x=horizontalSequence.end();
y=horizontalSequence.end();
y--;
while(y!=horizontalSequence.begin())
{
x--;
y--;
if( *x!=*y+1 )
{
list << (int)floor( (*x+*y)/2 );
cuts << QPair<layerDirectionType,
int>::QPair(HORIZONTAL, (int)floor( (*x+*y)/2 ));
iterated = true;
}
}
horizontalSequence += list;
qHeapSort( horizontalSequence );
}while(iterated);
  return cuts;
}

QValueList<CPathCut> CGRoute::CPXbuildnet(CPXENVptr& env,CPXNETptr& net,CLayer*
layer,const int track) const
{
QValueList<CPathCut> list;
QValueListIterator<CPathCut> it;
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
QValueList<CPathCut> cutpaths;
if(wires.current()->cutPathsAndBounds(layer,track,cutpaths))
list += cutpaths;
wires.next();
}

int numNets = list.count();
int numSegs = layer->numcEdges();
int numVertices = 1 + numSegs + numNets;
double* supply = new double[ numVertices ];
int numArcs = numSegs;
for(it=list.begin();it!=list.end();it++) numArcs += (*it).ub() -
(*it).lb() + 1;
int* from = new int[numArcs];
int* to = new int[numArcs];
double* ub = new double[numArcs];
double* lb = new double[numArcs];
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
double* obj = new double[numArcs];

//supplys at segment layer, blockage + previous flow
supply[0] = 0;
for(int i=0;i<numSegs;i++)
{
supply[i+1] = - layer->cellFlow(track,i) -
layer->cellBlock(track,i);
supply[0] += - supply[i+1];
}

//set supply at net layer
QPair<QRect,QRect> layerRects = layer->divideAtTrack(track);
QRect left = layerRects.first;
QRect right = layerRects.second;

//std::cout << "left" << left.left() << " " << left.top() << ":" << left.right()
<< " " << left.bottom() << "\n";
//std::cout << "right" << right.left() << " " << right.top() << ":" <<
right.right() << " " << right.bottom() << "\n";

int i = numSegs+1;
for(it=list.begin();it!=list.end();it++)
{
CPathCut pathcut = *it;
gwire* wire = pathcut.wire();
gpath tmp;

QRect farfrom = wire->farfrom(pathcut.paths().first(),tmp);
QRect farto = wire->farto(pathcut.paths().last(),tmp);

/*{
QPtrListIterator<gpath> it(*wire);
std::cout << "\nwire\n";
for(it.toFirst();it.current();++it)
{
gpath* path = *it;
QRect from = path->from();
QRect to = path->to();
std::cout << "from\t" << from.left() << " " << from.top() << ":" <<
from.right() << " " << from.bottom() << "\n";
std::cout << "to\t" << to.left() << " " << to.top() << ":" << to.right()
<< " " << to.bottom() << "\n";
}
std::cout <<  layer->name() << " " << track << "\n";
std::cout << "left\t" << left.left() << " " << left.top() << ":" << left.right()
<< " " << left.bottom() << "\n";
std::cout << "right\t" << right.left() << " " << right.top() << ":" <<
right.right() << " " << right.bottom() << "\n";
std::cout << "farfrom\t" << farfrom.left() << " " << farfrom.top() << ":" <<
farfrom.right() << " " << farfrom.bottom() << "\n";
std::cout << "farto\t" << farto.left() << " " << farto.top() << ":" <<
farto.right() << " " << farto.bottom() << "\n";
}*/
QRect fromside,toside;
if(left.contains(farfrom)) fromside=left;
else if(right.contains(farfrom)) fromside=right;
else assert(false);
if(left.contains(farto)) toside=left;
else if(right.contains(farto)) toside=right;
else assert(false);
if(fromside!=toside)
{
supply[i] = -1;
supply[0] += 1;
}else supply[i] = 0; //if out of box, will not be cutting the
cut
i++;
}

//wire from source node to the segment node
for(int i=0;i<numSegs;i++)
{
from[i] = 0;
to[i] = i+1;
lb[i] = 0;
ub[i] = CPX_INFBOUND;
obj[i] = 0;
}

//wire form segment nodes to net nodes
int arcIndex = numSegs;
i = numSegs + 1;
for(it=list.begin();it!=list.end();it++)
{

CPathCut pathcut = *it;
gwire* wire = pathcut.wire();

gpath farfrompath,fartopath;

QRect farfrom =
wire->farfrom(pathcut.paths().first(),farfrompath);
QRect farto = wire->farto(pathcut.paths().last(),fartopath);
int min,max;
min = 0; max = 0;
layerDirectionType dirmin,dirmax;
dirmin = dirmax = UNKNOWNLAYERDIRECTION;


if(layer->dir()==VERTICAL)
{
min = ::min(farfrom.center().x(),farto.center().x());
max = :: max(farfrom.center().x(),farto.center().x());
if( min == farfrom.center().x() ) dirmin =
farfrompath.dir(); else dirmin = fartopath.dir();
if( max == farfrom.center().x() ) dirmax =
farfrompath.dir(); else dirmax = fartopath.dir();
} else if(layer->dir()==HORIZONTAL)
{
min = ::min(farfrom.center().y(),farto.center().y());
max = :: max(farfrom.center().y(),farto.center().y());
if( min == farfrom.center().y() ) dirmin =
farfrompath.dir(); else dirmin = fartopath.dir();
if( max == farfrom.center().y() ) dirmax =
farfrompath.dir(); else dirmax = fartopath.dir();
}
```

```
// {extern QTextStream lg;
// QPtrList<gpath> oh = pathcut.paths();
// QPtrListIterator<gpath> it2(oh);
// for(it2.toFirst();it2.current();++it2)
// {
// gpath* path =  *it2  ;
// lg << path->from().left() << " " <<
path->from().top()<< " " <<path->from().right()<< " " << path->from().bottom()
<< "\n";
// lg << path->to().left() << " "
<<path->to().top() << " " <<path->to().right()<< " " <<path->to().bottom() <<
"\n";
// lg << "\n";
// }
// lg << (*it).lb() <<  " " << (*it).ub() << "\n";
// lg << min << " " << max << "\n";
// lg << layer->name() << " " <<  track << "\n";
// lg << farfrom.left() << " " <<farfrom.top() << " "
<<farfrom.right()<< " " <<farfrom.bottom() << "\n";
// lg << farto.left() << " " <<farto.top() << " "
<<farto.right()<< " " <<farto.bottom() << "\n";
// lg << "\n\n";
// }

int str = (*it).lb();
int end = (*it).ub();
for(int j=str; j<=end; j++)
{
from[arcIndex] = j+1;
to[arcIndex] = i;
lb[arcIndex] = 0;
ub[arcIndex] = 1;
if( (j==min) )
{
if(dirmin==layer->dir() ||
dirmin==UNKNOWNLAYERDIRECTION)
obj[arcIndex] = 0; else obj[arcIndex] =
1;
}else if( j==max )
{
    if(dirmax==layer->dir() ||
dirmax==UNKNOWNLAYERDIRECTION)
obj[arcIndex] = 0; else obj[arcIndex] = 1;
}else if( (j>min) && (j<max) ) obj[arcIndex] = 1;
else if( j>max ) obj[arcIndex] = j-max+1;
else if( j<min ) obj[arcIndex] = min-j+1;
arcIndex++;
}
i++;
}

int status;
env = CPXopenCPLEX(&status);
// CPXsetintparam(env,CPX_PARAM_SCRIND,CPX_ON);
CPXsetintparam(env,CPX_PARAM_ROWGROWTH,numVertices);
CPXsetintparam(env,CPX_PARAM_COLGROWTH,numArcs);
net = CPXNETcreateprob(env,&status,"net.net");
CPXNETchgobjsen(env,net,CPX_MIN);
CPXNETaddnodes(env,net,numVertices,supply,NULL);
CPXNETaddarcs(env,net,numArcs,from,to,lb,ub,obj,NULL);
// CPXNETwriteprob(env,net,"net.net",NULL);
delete[] supply;
delete[] from;
delete[] to;
delete[] ub;
delete[] lb;
delete[] obj;
return list;
}

int* CGRoute::CPXoptnet(CPXENVptr& env, CPXNETptr& net, const int  numSegs,
const int cellCap) const
{
QValueList<int> zeroList;
QValueListIterator<int> it;
int res;
int obj = IINF;
int cost;
double dcost;
bool optimum = false;
bool* done = new bool[numSegs];
int* val = new int[numSegs];
for(int i=0;i<numSegs;i++) done[i]=false;

do
{ break;
res = CPXoptnet(env,net,numSegs,obj,val,done);
CPXNETgetobjval(env,net,&dcost);
cost = (int)dcost;
std::cout << "network objective :" << res << " : ";// << "\n";
zeroList.clear();
for(int i=0;i<numSegs;i++)
{
if(done[i]) continue;
if( res - val[i] > 0.5 ) continue;
zeroList << i;
done[i] = true;
}
for(it=zeroList.begin();it!=zeroList.end();it++) val[*it] = res;
std::cout << zeroList.count() << " has zero slack ; ";
for(it=zeroList.begin();it!=zeroList.end();it++)
{
val[*it] = res-1;
double resnext =
CPXoptnet(env,net,numSegs,res,val,done);
CPXNETgetobjval(env,net,&dcost);
// int costnext = (int)dcost;
if( resnext < res)
{
if( res > 0.8*cellCap)  {done[*it] =
false;std::cout << ".";}
else std::cout << "*";
//else if( costnext <= cost ) done[*it] = false;
//else done[*it] = false;
}else std::cout << "*";
val[*it] = res;
}
std::cout << "\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
bool all =
false;for(it=zeroList.begin();it!=zeroList.end();it++) if(done[*it]) all = true;
if( !all ) for(it=zeroList.begin();it!=zeroList.end();it++)
done[*it] = true;
obj = res;
optimum = true;
int q = 0;
for(int i=0;i<numSegs;i++) {optimum = optimum &&
done[i];if(done[i]) q++;}
// std::cout << "\t " << q << "/" << numSegs << "\n";
}while(!optimum);

CPXoptnet(env,net,numSegs,obj,val,done);
//CPXNETwriteprob(env,net,"out.net",NULL);

int numArcs = CPXNETgetnumarcs(env,net);
double* x = new double[numArcs];
CPXNETgetx(env,net,x,0,numArcs-1);
int numNets = CPXNETgetnumnodes(env,net)-numSegs-1;
int* assign = new int[numNets];
for(int i=0;i<numNets;i++)
{
int netIndex = numSegs + i + 1;
int numArcs,arcbeg,tmp,surplus,from,to;

CPXNETgetnodearcs (env,net, &numArcs,&arcbeg,&tmp,0,
&surplus,netIndex,netIndex);
int* arc = new int[-surplus];
CPXNETgetnodearcs (env,net,
&numArcs,&arcbeg,arc,-surplus,&surplus,netIndex,netIndex);
assert(surplus==0);
bool assigned = false;
for(int j=0;j<numArcs;j++)
{
if( x[ arc[j] ] < 0.5 ) continue;
CPXNETgetarcnodes (env,net, &from,&to, arc[j], arc[j] );
assert(to==netIndex);
assert(from<=numSegs);
assign[i] = from-1;
assigned = true;
}
if(!assigned) assign[i] = -1;
delete[] arc;
    }
CPXNETfreeprob(env,&net);
CPXcloseCPLEX(&env);
delete[] done;
delete[] val;
delete[] x;
return assign;
}


int CGRoute::CPXoptnet(CPXENVptr& env, CPXNETptr& net,const int numSegs,const
int start, int* val, bool* done)  const
{
int* index = new int[numSegs];
char*  lu = new char[numSegs];
char*  ll = new char[numSegs];
double*  ub = new double[numSegs];
double*  lb = new double[numSegs];

double up = start;
double down = 0;

for(int i=0;i<numSegs;i++)
{
index[i] = i;
lu[i] = 'U';
ll[i] = 'L';
if(done[i]){ub[i] = val[i];lb[i] = val[i];}else{ ub[i] =
start;lb[i] = 0;}
up = start;
down = 0;
}
CPXNETchgbds(env,net,numSegs,index,lu, ub );
CPXNETchgbds(env,net,numSegs,index,ll, lb );
// CPXNETwriteprob(env,net,"out.net",NULL);
CPXNETprimopt(env,net);
if( CPXNETgetstat(env,net)!=CPX_STAT_OPTIMAL) return (int)CPX_INFBOUND;
do
{
for(int i=0;i<numSegs;i++) if(done[i]){ ub[i] = val[i];lb[i] =
val[i];}else {ub[i] = floor((up+down)/2.0);lb[i] = 0;}
CPXNETchgbds(env,net,numSegs,index,lu, ub );
CPXNETchgbds(env,net,numSegs,index,ll, lb );
//CPXNETwriteprob(env,net,"out.net",NULL);
CPXNETprimopt(env,net);
if( CPXNETgetstat(env,net)!=CPX_STAT_OPTIMAL)
down = floor((up+down)/2.0);
else
up = floor((up+down)/2.0);
}while( up-down > 1 );
for(int i=0;i<numSegs;i++)
if(done[i])
{ub[i] = val[i];lb[i] = val[i];}
else{ub[i] = up;lb[i] = 0;}

CPXNETchgbds(env,net,numSegs,index,lu, ub );
CPXNETchgbds(env,net,numSegs,index,ll, lb );
//CPXNETwriteprob(env,net,"",NULL);
CPXNETprimopt(env,net);
double* x = new double[numSegs];
CPXNETgetx(env,net,x,0,numSegs-1);
assert( CPXNETgetstat(env,net)==CPX_STAT_OPTIMAL);
    double max = 0;
for(int i=0;i<numSegs;i++)
{
if( done[i] ) continue;
val[i] = (int)x[i];
if( x[i] >  max ) max = x[i];
}

delete[] x;
delete[] index;
delete[] lu;
delete[] ll;
delete[] ub;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
delete[] lb;

return (int)max;
}


#include "cgroute.h"
#include "gwire.h"
#include "gpath.h"
#include "clayer.h"

void CGRoute::pp()
{

extern QTextStream lg;
lg << "Start Post Processing\n";
int pass = 1;
std::cout << "\tpost processing\n";
std::cout << "\tpass " << pass++ <<"\n";
std::cout << "\tover :" << numCellOverflows(); pp1();std::cout << "\tpp1
fixed " << numCellOverflows() << "\n";
std::cout << "\tover :" << numCellOverflows(); pp2();std::cout << "\tpp2
fixed " << numCellOverflows() << "\n";
// std::cout << "\tpads :" << numPadViolations(); pp3();std::cout << "\tpp3
fixed " << numPadViolations() << "\n";
// std::cout << "\tpads :" << numPadViolations(); pp4();std::cout << "\tpp4
fixed " << numPadViolations() << "\n";
int prevNumOver;
int numOver = numCellOverflows();
int prevNumPad;
int numPad = numPadViolations();
std::cout << "\toverflows " << numOver << "\n";
std::cout << "\tpad violations " << numPad << "\n";
if(numOver<=0 && numPad<=0) return;

do
{
std::cout << "\tpass " << pass++ << "\n";
prevNumOver = numOver;
prevNumPad = numPad;
std::cout << "\tover :" << numCellOverflows(); pp1();std::cout
<< "\tpp1 fixed " << numCellOverflows() << "\n";
std::cout << "\tover :" << numCellOverflows(); pp2();std::cout
<< "\tpp2 fixed " << numCellOverflows() << "\n";
// std::cout << "\tpads :" << numPadViolations(); pp3();std::cout
<< "\tpp3 fixed " << numPadViolations() << "\n";
// std::cout << "\tpads :" << numPadViolations(); pp4();std::cout
<< "\tpp4 fixed " << numPadViolations() << "\n";
numOver = numCellOverflows();
numPad = numPadViolations();
std::cout << "\toverflows " << numOver << "\n";
std::cout << "\tpad violations " << numPad << "\n";
}while((numOver>0 && numOver<prevNumOver) || (numPad>0 &&
numPad<prevNumPad));
lg << "End Post Process\n\n";
}

void CGRoute::pp1()
{
bool overflows;
QValueListIterator<CTileEdge> eIt;
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
gwire* wire = wires.current();
wire->first();
while(wire->current())
{
gpath* path = wire->current();
QValueList<CTileEdge> edges = path->edges();
overflows = false;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge edge = *eIt;
if(
edge.layer()->cellOverflows(edge.track(),edge.edge())){overflows = true;
break; }
}
if(overflows)
{if(wire->pp1(path)){continue;}}//wire->first();continue;}}
wire->next();
}
wires.next();
}
}

void CGRoute::pp2()
{
bool overflows;
QValueListIterator<CTileEdge> eIt;
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
gwire* wire = wires.current();
wire->first();
while(wire->current())
{
gpath* path = wire->current();
QValueList<CTileEdge> edges = path->edges();
overflows = false;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge edge = *eIt;
if(
edge.layer()->cellOverflows(edge.track(),edge.edge())){overflows = true;
break; }
}
if(overflows)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
{if(wire->pp2(path)){continue;}}//wire->first();continue;}}
wire->next();
}
wires.next();
}
}

void CGRoute::pp3()
{
// bool overflows;
// QValueListIterator<CTileEdge> eIt;
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
gwire* wire = wires.current();
wire->pp3();
wires.next();
}

}

void CGRoute::pp4()
{
// bool overflows;
// QValueListIterator<CTileEdge> eIt;
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
gwire* wire = wires.current();
wire->pp4();
wires.next();
}

}


//void CGRoute::em()
//{
// QPtrList<gwire> wires = this->wires();
// wires.first();
// while(wires.current())
// {
// gwire* wire = wires.current();
// wire->first();
// while(wire->current())
// {
// gpath* path = wire->current();
// QValueList<CTileEdge> edges = path->edges();
// bool overflows = false;
// for(eIt=edges.begin();eIt!=edges.end() &&
!overflows;eIt++)
// {
// CTileEdge tileEdge = *eIt;
// if(
tileEdge->layer()->cellOverflows(tileEdge->track(),tileEdge->edge()) ) overflows
= true;
// }
// if(overflows) em(wire,path);
// wire->next();
// }
// wires.next();
// }
//}

//void CGRoute:em(gwire* wire)
//{
// gpath path = *wire->current();
// gpath prev = *wire->prev();
// wire->next(); gpath next = *wire->next();
//
// switch(path.dir())
// {
// case VERTICAL:
// CLayerPair nextLayer = layer;
// do
// {
// if(prev->dir()==HORIZONTAL)
// {
//
// }
// while(
// break;
// case HORIZONTAL:
// break;
// default:
// }
//}

//void CPrj::em()
//{
//// QFile file("shift.log");file.open(IO_WriteOnly);QTextStream f(&file);
// bool improved;
// do
// {
// improved = false;
// unsigned long int count;
// count = 0;
//   for(unsigned int l=0;l<m_layers.size();l++)
//  {
//  CLayer* pLayer = m_layers[l];
//  for(unsigned int i=0;i<pLayer->smallSizeX();i++)
//  {
//  for(unsigned int j=0;j<pLayer->smallSizeY();j++)
//  {
// QValueList< QPair<CTree*,CNode*> >
overflows =pLayer->smallAsgnedEdges(i,j);
//  int numOverflows =
pLayer->smallSegBlockage(i,j)+overflows.count()-pLayer->smallSegCap(i,j);
// if( numOverflows>0) count +=
numOverflows;
//  while(numOverflows>0)
//  {
// if(overflows.count()<=0)
break;
//      QPair<CTree*,CNode*> overflow =
overflows.front();
```

```
//
// overflows.pop_front();
// CTree* pTree =
overflow.first;
// CNode* pNode =
overflow.second;
// bool succ =
elim(pTree,pNode);
//        if( succ )
{improved=true;numOverflows--;}
//  }
//  }
//  }
//  }
// cout << "Overflows : \t" << count << "\n";
// }while(improved);
//// file.close();
//}
//
//bool  CPrj::elim(CTree* pTree, CNode* pNode)
//{
// CNode* pParent = pNode->parent();
// CNode* pChild = pNode->children().first();
// CNode* pDummy;
// CEdge* pEdge;
// CLayer* pLayer;
// bool elimd = false;
// int d;
//
// if(pParent->type()==DUMMY &&
m_layers[pNode->slayer()]->dir()!=m_layers[pParent->slayer()]->dir())
// {
//
// bool blocked = false;
// if(pNode->sasgnX()==pParent->sasgnY()) d = -1; else d = +1;
// int count = 0;
// do
// {
// pLayer = m_layers[pParent->slayer()];
// int i = pParent->sasgnX();
// int j = pParent->sasgnY()+d;
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// pEdge = pParent->upEdge();
// pParent = pParent->parent();
// count++;
// }while(pParent->type()==DUMMY &&
pLayer->dir()==m_layers[pParent->slayer()]->dir() && !blocked);
// if(!blocked &&
// pParent->type()==DUMMY &&
// pParent->sasgnX()!=pNode->sasgnX() )
// {
// int l = pTree->layer() + pEdge->relLayer();
//
if(m_layers[l]->dir()!=m_layers[pParent->slayer()]->dir()) l++;
// pLayer = m_layers[l];
// int i = pParent->sasgnX()+d;
// int j = pParent->sasgnY();
```

```
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// }else if(!blocked && pParent->type()!=DUMMY)
// {
// pParent = pNode->parent();
// if(pNode->sasgnY()==pParent->sasgnX()) d = -1;
else d = +1;
// int  i = pNode->sasgnX();
// int  j = pNode->sasgnY() + d * count;
// int l = pTree->layer() + pEdge->relLayer();
//
if(m_layers[l]->dir()!=m_layers[pNode->slayer()]->dir()) l++;
// pLayer = m_layers[l];
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// }
// if(!blocked)
// {
// pParent = pNode->parent();
// if(pNode->sasgnX()==pParent->sasgnY()) d = -1;
else d = +1;
// int count = 0;
// do
// {
// pLayer = m_layers[pParent->slayer()];
//
pLayer->ripUpFromSmall(pParent->sasgnX(),pParent->sasgnY(),
//

QPair<CTree*,CNode*>::QPair(pTree,pParent));
//
pLayer->smallSegAsgn(pParent->sasgnX(),pParent->sasgnY()+d,
//


QPair<CTree*,CNode*>::QPair(pTree,pParent));
// pParent->sasgnY( pParent->sasgnY()+d );
//              if(pLayer->dir()==VERTICAL)
// pParent->x(
pParent->x()+d*m_smallTileSize.width() );
// else
// pParent->y(
pParent->y()+d*m_smallTileSize.height() );
//
pParent->children().first()->upEdge()->update();
// pEdge = pParent->upEdge();
// pParent = pParent->parent();
// count++;
// }while(pParent->type()==DUMMY &&
pLayer->dir()==m_layers[pParent->slayer()]->dir());
// if( pParent->type()==DUMMY &&
pParent->sasgnX()==pNode->sasgnX() )
// {
//
m_layers[pParent->slayer()]->ripUpFromSmall(pParent->sasgnX(),pParent->sasgnY(),
//
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
QPair<CTree*,CNode*>::QPair(pTree,pParent));
// pTree->ripUp(pParent);
// }else
// {
// pParent = pNode->parent();
// if(pNode->sasgnY()==pParent->sasgnX()) d
= -1; else d = +1;
// int  i = pNode->sasgnX();
// int  j = pNode->sasgnY() + d * count;
// int l = pTree->layer() +
pEdge->relLayer();
//                  if(m_layers[l]->dir()!=m_layers[pNode->slayer()]->dir())
l++;
// pLayer = m_layers[l];
// double x,y;
//                  if(pLayer->dir()==VERTICAL)
// {
// x =
j*m_smallTileSize.width()+m_smallTileSize.width()/2.0+m_dieArea.left();
// y =
i*m_smallTileSize.height()+m_dieArea.bottom();
// }else
// {
// x =
i*m_smallTileSize.width()+m_dieArea.left();
// y =
j*m_smallTileSize.height()+m_smallTileSize.height()/2.0+m_dieArea.bottom();
// }
// pDummy =
pTree->insertDummyAtEdgeAtPoint(pEdge, CPoint(x,y));
//                  pDummy->smallSegAsgn( 1, i, j );
//
pLayer->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pDummy));
// }
//
m_layers[pNode->slayer()]->ripUpFromSmall(pNode->sasgnX(),pNode->sasgnY(),
//

QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pTree->ripUp(pNode);
//              elimd = true;
// }else elimd = false;
// }
// if(!elimd && pChild->type()==DUMMY &&
m_layers[pNode->slayer()]->dir()!=m_layers[pChild->slayer()]->dir() )
// {
// bool blocked = false;
// if(pNode->sasgnX()==pChild->sasgnY()) d = -1; else d = +1;
// int count = 0;
// do
// {
// pLayer = m_layers[pChild->slayer()];
// int i = pChild->sasgnX();
// int j = pChild->sasgnY()+d;
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// pChild = pChild->children().first();
// pEdge = pChild->upEdge();
```

```
// count++;
// }while(pChild->type()==DUMMY &&
pLayer->dir()==m_layers[pChild->slayer()]->dir() && !blocked);
// if(!blocked &&
// pChild->type()==DUMMY &&
// pChild->sasgnX()!=pNode->sasgnX() )
// {
// int l = pTree->layer() + pEdge->relLayer();
//
if(m_layers[l]->dir()!=m_layers[pChild->slayer()]->dir()) l++;
// pLayer = m_layers[l];
// int i = pChild->sasgnX()+d;
// int j = pChild->sasgnY();
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// }else if(!blocked && pChild->type()!=DUMMY)
// {
// pChild = pNode->children().first();
// if(pNode->sasgnY()==pChild->sasgnX()) d = -1;
else d = +1;
// int  i = pNode->sasgnX();
// int  j = pNode->sasgnY() + d * count;
// int l = pTree->layer() + pEdge->relLayer();
//
if(m_layers[l]->dir()!=m_layers[pNode->slayer()]->dir()) l++;
// pLayer = m_layers[l];
//
if(pLayer->smallSegBlockage(i,j)+pLayer->smallSegFlow(i,
j)>=pLayer->smallSegCap(i,j)) blocked = true;
// }
// if(!blocked)
// {
// pChild = pNode->children().first();
// if(pNode->sasgnX()==pChild->sasgnY()) d = -1;
else d = +1;
// int count = 0;
// do
// {
// pLayer = m_layers[pChild->slayer()];
//
pLayer->ripUpFromSmall(pChild->sasgnX(),pChild->sasgnY(),
//

QPair<CTree*,CNode*>::QPair(pTree,pChild));
//
pLayer->smallSegAsgn(pChild->sasgnX(),pChild->sasgnY()+d,
//

QPair<CTree*,CNode*>::QPair(pTree,pChild));
// pChild->sasgnY( pChild->sasgnY()+d );
//                  if(pLayer->dir()==VERTICAL)
// pChild->x(
pChild->x()+d*m_smallTileSize.width() );
// else
// pChild->y(
pChild->y()+d*m_smallTileSize.height() );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// pChild->upEdge()->update();
// pChild = pChild->children().first();
// pEdge =  pChild->upEdge();
// count++;
// }while(pChild->type()==DUMMY &&
pLayer->dir()==m_layers[pChild->slayer()]->dir());
// if( pChild->type()==DUMMY &&
pChild->sasgnX()==pNode->sasgnX() )
// {
//
m_layers[pChild->slayer()]->ripUpFromSmall(pChild->sasgnX(),pChild->sasgnY(),
//
QPair<CTree*,CNode*>::QPair(pTree,pChild));
// pTree->ripUp(pChild);
// }else
// {
// pChild = pNode->children().first();
// if(pNode->sasgnY()==pChild->sasgnX()) d
= -1; else d = +1;
// int  i = pNode->sasgnX();
// int  j = pNode->sasgnY() + d * count;
// int l = pTree->layer() +
pEdge->relLayer();
//                    if(m_layers[l]->dir()!=m_layers[pNode->slayer()]->dir())
l++;
// pLayer = m_layers[l];
// double x,y;
//                    if(pLayer->dir()==VERTICAL)
// {
// x =
j*m_smallTileSize.width()+m_smallTileSize.width()/2.0+m_dieArea.left();
// y =
i*m_smallTileSize.height()+m_dieArea.bottom();
// }else
// {
// x =
i*m_smallTileSize.width()+m_dieArea.left();
// y =
j*m_smallTileSize.height()+m_smallTileSize.height()/2.0+m_dieArea.bottom();
// }
// pDummy =
pTree->insertDummyAtEdgeAtPoint(pEdge, CPoint(x,y));
//                    pDummy->smallSegAsgn( l, i, j );
//
pLayer->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pDummy));
// }
//
m_layers[pNode->slayer()]->ripUpFromSmall(pNode->sasgnX(),pNode->sasgnY(),
// QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pTree->ripUp(pNode);
//                 elimd = true;
// }else elimd = false;
// }
// return elimd;
//}
//
//
//void CPrj::ef()
//{
// cout << "ef\n";
//    bool improved;
// do
// {
// improved = false;
// unsigned long int count;
// count = 0;
//    for(unsigned int l=0;l<m_layers.size();l++)
//  {
//  CLayer* pLayer = m_layers[l];
//  for(unsigned int i=0;i<pLayer->smallSizeX();i++)
//  {
//  for(unsigned int j=0;j<pLayer->smallSizeY();j++)
//  {
// QValueList< QPair<CTree*,CNode*> >
overflows =pLayer->smallAsgnedEdges(i,j);
//  int numOverflows =
pLayer->smallSegBlockage(i,j)+overflows.count()-pLayer->smallSegCap(i,j);
// if( numOverflows>0) count +=
numOverflows;
//  while(numOverflows>0)
// {
// if(overflows.count()<=0)
break;
//  QPair<CTree*,CNode*>
overflow = overflows.front();
// overflows.pop_front();
// CTree* pTree =
overflow.first;
// CNode* pNode =
overflow.second;
// bool succ =
flip(pTree,pNode);
//          if( succ )
{improved=true;numOverflows--;cu(pTree);}
//  }
//  }
//  }
// }
// cout << "Overflows : \t" << count << "\n";
// }while(improved);
//}
//
//bool CPrj::flip(CTree* pTree, CNode* n)
//{
//  CNode* u = n;
// CNode* d = n;
// CNode* uu,*dd;
//
// do
// {
// if(u->type()==DUMMY &&
m_layers[u->slayer()]->dir()==m_layers[n->slayer()]->dir() ) u = u->parent();
// if(d->type()==DUMMY &&
m_layers[d->slayer()]->dir()==m_layers[n->slayer()]->dir() ) d =
d->children().first();
//
// uu = u;dd = d;if( flip(pTree,uu,dd) ) return true;
// while( dd->type()==DUMMY &&
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
(m_layers[dd->slayer()]->dir()!=m_layers[n->slayer()]->dir() ||
dd->sasgnY()==n->sasgnY()) )
// {
// dd = dd->children().first();
// if( flip(pTree,uu,dd) ) return true;
// }
//
// dd = d;uu = u;if( flip(pTree,uu,dd) ) return true;
// while( uu->type()==DUMMY &&
(m_layers[uu->slayer()]->dir()!=m_layers[n->slayer()]->dir() ||
uu->sasgnY()==n->sasgnY()) )
// {
// uu = uu->parent();
// if( flip(pTree,uu,dd) ) return true;
// }
// }while(  (u->type()==DUMMY &&
m_layers[u->slayer()]->dir()==m_layers[n->slayer()]->dir() ) ||
// (d->type()==DUMMY &&
m_layers[d->slayer()]->dir()==m_layers[n->slayer()]->dir() ) );
// return false;
//}
//
//bool CPrj::flip(CTree* pTree, CNode* u, CNode* d)
//{
//   int dx,dy,ux,uy;
//
//
if(d->type()!=DUMMY){
dx=(int)floor((d->x()-m_dieArea.left())/m_smallTileSize.width());dy=(int)floor((
d->y()-m_dieArea.bottom())/m_smallTileSize.height());}
// else if( m_layers[d->slayer()]->dir()==VERTICAL ) {dx =
d->sasgnY();dy=d->sasgnX();}
// else{dx = d->sasgnX();dy=d->sasgnY();}
//
if(u->type()!=DUMMY){
ux=(int)floor((u->x()-m_dieArea.left())/m_smallTileSize.width());uy=(int)floor((
u->y()-m_dieArea.bottom())/m_smallTileSize.height());}
// else if( m_layers[u->slayer()]->dir()==VERTICAL ) {ux =
u->sasgnY();uy=u->sasgnX();}
// else{ux = u->sasgnX();uy=u->sasgnY();}
// if(d->type()==DUMMY && m_layers[d->slayer()]->dir()==HORIZONTAL && dx>ux
) dx--;
// if(u->type()==DUMMY && m_layers[u->slayer()]->dir()==HORIZONTAL && ux>dx
) ux--;
// if(d->type()==DUMMY && m_layers[d->slayer()]->dir()==VERTICAL && dy>uy )
dy--;
// if(u->type()==DUMMY && m_layers[u->slayer()]->dir()==VERTICAL && uy>dy )
uy--;
// if(ux==dx || uy==dy) return false;
//
// CNode* n = d->parent();
// do
// {
// int l = n->slayer();int x = n->sasgnX(); int y = n->sasgnY();
// if( m_layers[l]->dir()==VERTICAL)
// {
// if(y==dx) y = ux; else  if(y==ux) y = dx; else return
false;
// if(m_layers[l]->full(x,y)) return false;
// }else
// {
//           if(y==dy) y = uy; else if(y==uy) y = dy;else return false;
// if(m_layers[l]->full(x,y)) return false;
// }
// n = n->parent();
// }while(n!=u);
//
//
// bool mode = false;
// n = d->parent();
// CNode* node;
// layerDirectionType dir = m_layers[n->slayer()]->dir();
// do
// {
// int l = n->slayer();int x = n->sasgnX(); int y = n->sasgnY();
// if(!mode)
// {
// if(dir!=m_layers[n->slayer()]->dir())
// {
// mode = true;
// node = d;
// }
// dir = m_layers[n->slayer()]->dir();
// }
//
m_layers[l]->ripUpFromSmall(x,y,QPair<CTree*,CNode*>::QPair(pTree,n));
// if( m_layers[l]->dir()==VERTICAL)
// {
// if(y==dx) y = ux; else if(y==ux) y = dx;
// }else
// {
//         if(y==dy) y = uy; else if(y==uy) y = dy;
// }
// if(!mode)
// {
// if(m_layers[l]->dir()==VERTICAL)
// n->x( m_dieArea.left() +
y*m_smallTileSize.width()+m_smallTileSize.width()/2.0);
// else
// n->y( m_dieArea.bottom() +
y*m_smallTileSize.height()+m_smallTileSize.height()/2.0);
//
// n->sasgnY(y);
//
m_layers[l]->smallSegAsgn(x,y,QPair<CTree*,CNode*>::QPair(pTree,n));
// n = n->parent();
// }else
// {
// CPoint p;
// if(m_layers[l]->dir()==VERTICAL)
// {
// p.x(m_dieArea.left() +
y*m_smallTileSize.width()+m_smallTileSize.width()/2.0);
// p.y(m_dieArea.bottom() +
x*m_smallTileSize.height());
// }else
// {
// p.x(m_dieArea.left() +
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
x*m_smallTileSize.width());
// p.y(m_dieArea.bottom() +
y*m_smallTileSize.height()+m_smallTileSize.height()/2.0);
// }
// int elayer = node->upEdge()->relLayer();
// node =
pTree->insertDummyAtEdgeAtPoint(node->upEdge(),p);
// node->slayer(l);node->sasgnX(x);node->sasgnY(y);
// node->blayer( n->blayer() );
// node->upEdge()->relLayer(elayer);
//
m_layers[l]->smallSegAsgn(x,y,QPair<CTree*,CNode*>::QPair(pTree,node));
// CNode* tmp = n;
// n = n->parent();
// pTree->ripUp(tmp);
// }
// }while(n!=u);
// return true;
//}
//
//void  CPrj::lc()
//{
// cout << "lc1\n";
// QDictIterator<CTree> treeIt(m_trees);
// for(treeIt.toFirst();treeIt.current();++treeIt)
// {
// CTree* pTree = *treeIt;
// int treel = pTree->layer();
// QPtrList<CNode> nodes;
// nodes.append( pTree->root() );
// while(!nodes.isEmpty())
// {
// CNode* pNode = nodes.take(0);
// CNode* pParent = pNode->parent();
// QPtrListIterator<CNode> n(pNode->children());
//      for(n.toFirst();n.current(); ++n) nodes.append( *n
);
// if(pNode->type()==ROOT) continue;
// if(pNode->type()!=DUMMY)
// {
// if(pParent->type()==DUMMY)
pNode->upEdge()->relLayer( pParent->upEdge()->relLayer() );
// }else
// {
// int l = pNode->slayer();int i =
pNode->sasgnX();int j = pNode->sasgnY();
// unsigned int le = treel +
pNode->upEdge()->relLayer();
// if(m_layers[le]->dir()!=m_layers[l]->dir()) l =
le +1 ; else l = le;
// if(l!=pNode->slayer())
// {
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// unsigned int e = le;
//          for(unsigned int d=0;d<m_layers.size();d++)
// {
// if((unsigned int)pNode->slayer()<le)
// {
// if(le+d<(m_layers.size()-1) ) {e
= le+d;break;}
// else if(le>=d ){e = le-d;break;}
// }else
// {
// if(le>=d  ) {e = le-d;break;}
// else if(le+d<(m_layers.size()-1)
){e = le+d;break;}
// }
// }
// if(e!=le)
// {
//
if(m_layers[e]->dir()!=m_layers[pNode->slayer()]->dir()) l = e+1; else l = e;
// if(l!=pNode->slayer())
// {
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// pNode->upEdge()->relLayer( e - treel );
// pNode->upEdge()->update();
//// cout << "changed layer\n";
// }
// }
// }
// }
//}
//
//void  CPrj::lc1()
//{
// cout << "lc1\n";
// QDictIterator<CTree> treeIt(m_trees);
// for(treeIt.toFirst();treeIt.current();++treeIt)
// {
// CTree* pTree = *treeIt;
// int treel = pTree->layer();
// QPtrList<CNode> nodes;
// nodes.append( pTree->root() );
// while(!nodes.isEmpty())
// {
// CNode* pNode = nodes.take(0);
// CNode* pParent = pNode->parent();
// QPtrListIterator<CNode> n(pNode->children());
//      for(n.toFirst();n.current(); ++n) nodes.append( *n
);
// if(pNode->type()==ROOT) continue;
// if(pNode->type()!=DUMMY)
// {
// if(pParent->type()==DUMMY)
pNode->upEdge()->relLayer( pParent->upEdge()->relLayer() );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// }else
// {
// int l = pNode->slayer();int i =
pNode->sasgnX();int j = pNode->sasgnY();
// unsigned int le = treel +
pNode->upEdge()->relLayer();
// if(m_layers[le]->dir()!=m_layers[l]->dir()) l =
le +1 ; else l = le;
// if(l!=pNode->slayer())
// {
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// unsigned int e = le;
//          for(unsigned int d=0;d<m_layers.size();d++)
// {
// if((unsigned int)pNode->slayer()<le)
// {
// if(le+d<(m_layers.size()-1) &&
isEdgeOk(pTree, pNode, le+d, false) ) {e = le+d;break;}
// else if(le>=d && isEdgeOk(pTree,
pNode, le-d, false)){e = le-d;break;}
// }else
// {
// if(le>=d && isEdgeOk(pTree,
pNode, le-d, false) ) {e = le-d;break;}
// else if(le+d<(m_layers.size()-1)
&& isEdgeOk(pTree, pNode, le+d, false)){e = le+d;break;}
// }
// }
// if(e!=le)
// {
//
if(m_layers[e]->dir()!=m_layers[pNode->slayer()]->dir()) l = e+1; else l = e;
// if(l!=pNode->slayer())
// {
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// pNode->upEdge()->relLayer( e - treel );
// pNode->upEdge()->update();
//// cout << "changed layer\n";
// }else if(
m_layers[pNode->slayer()]->blocked(pNode->sasgnX(),pNode->sasgnY()) )
// {
// for(unsigned int
d=0;d<m_layers.size();d++)
// {
// if((unsigned
int)pNode->slayer()<le)
// {
//
if(le+d<(m_layers.size()-1))
// {
//
if(m_layers[le+d]->dir()==m_layers[pNode->slayer()]->dir() &&
!m_layers[le+d]->blocked(pNode->sasgnX(),pNode->sasgnY()) ) {e = le+d;break;}
// else
if(m_layers[le+d]->dir()!=m_layers[pNode->slayer()]->dir() &&
!m_layers[le+d+1]->blocked(pNode->sasgnX(),pNode->sasgnY()) ) {e = le+d;break;}
//          }
// else if(le>=d)
// {
//
if(m_layers[le-d]->dir()==m_layers[pNode->slayer()]->dir() &&
!m_layers[le-d]->blocked(pNode->sasgnX(),pNode->sasgnY()) ){e = le-d;break;}
// else
if(m_layers[le-d]->dir()!=m_layers[pNode->slayer()]->dir() &&
!m_layers[le-d+1]->blocked(pNode->sasgnX(),pNode->sasgnY()) ){e = le-d;break;}
// }
// }else
// {
// if(le>=d)
// {
//
if(m_layers[le-d]->dir()==m_layers[pNode->slayer()]->dir() &&
!m_layers[le-d]->blocked(pNode->sasgnX(),pNode->sasgnY()) ) {e = le-d;break;}
// else
if(m_layers[le-d]->dir()!=m_layers[pNode->slayer()]->dir() &&
!m_layers[le-d+1]->blocked(pNode->sasgnX(),pNode->sasgnY()) ) {e = le-d;break;}
// }
// else
if(le+d<(m_layers.size()-1))
// {
//
if(m_layers[le+d]->dir()==m_layers[pNode->slayer()]->dir() &&
!m_layers[le+d]->blocked(pNode->sasgnX(),pNode->sasgnY()) ){e = le+d;break;}
//
if(m_layers[le+d]->dir()!=m_layers[pNode->slayer()]->dir() &&
!m_layers[le+d+1]->blocked(pNode->sasgnX(),pNode->sasgnY()) ){e = le+d;break;}
// }
// }
// }
// if(e!=le)
// {
//
if(m_layers[e]->dir()!=m_layers[pNode->slayer()]->dir()) l = e+1; else l = e;
// if(l!=pNode->slayer())
// {
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// pNode->upEdge()->relLayer( e -
treel );
// pNode->upEdge()->update();
//                    }
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// }
// }
// }
// }
//}
//
//
//void  CPrj::lc2()
//{
// cout << "lc2\n";
// QDictIterator<CTree> treeIt(m_trees);
// for(treeIt.toFirst();treeIt.current();++treeIt)
// {
// CTree* pTree = *treeIt;
// int treel = pTree->layer();
// QPtrList<CNode> nodes;
// nodes.append( pTree->root() );
// while(!nodes.isEmpty())
// {
// CNode* pNode = nodes.take(0);
// QPtrListIterator<CNode> n(pNode->children());
// for(n.toFirst();n.current(); ++n) nodes.append( *n );
// if(pNode->type()==ROOT) continue;
// {
// unsigned int le = treel +
pNode->upEdge()->relLayer();
// unsigned int e = le;
// unsigned int l;
// for(unsigned int d=0;d<m_layers.size();d++)
// {
// if(pNode->type()==DUMMY && (unsigned
int)pNode->slayer()<le)
// {
// if(le+d<(m_layers.size()-1) &&
isEdgeOk(pTree, pNode, le+d, true) ) {e = le+d;break;}
// else if(le>=d && isEdgeOk(pTree,
pNode, le-d, true)){e = le-d;break;}
// }else
// {
// if(le>=d && isEdgeOk(pTree,
pNode, le-d, true) ) {e = le-d;break;}
// else if(le+d<(m_layers.size()-1)
&& isEdgeOk(pTree, pNode, le+d, true)){e = le+d;break;}
// }
// }
// if(e!=le)
// {
// if(pNode->type()==DUMMY)
// {
//
if(m_layers[e]->dir()!=m_layers[pNode->slayer()]->dir()) l = e+1; else l = e;
// if(l!=(unsigned
int)pNode->slayer())
// {
// int i = pNode->sasgnX();
int j = pNode->sasgnY();
//
m_layers[pNode->slayer()]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,
pNode));
//
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
// pNode->slayer(l);
// }
// }
// pNode->upEdge()->relLayer( e - treel );
// pNode->upEdge()->update();
//// cout << "changed layer\n";
// }
// }
// }
// }
//}
//
//
//bool CPrj::isEdgeOk(CTree* pTree, CNode* pNode,int le, bool mode)
//{
// CNode* pParent = pNode->parent();
// int treel = pTree->layer();
// int l,i,j;
// int ld,ldd,lu,luu;
//
// if(!mode)
// {
// l = pNode->slayer();i=pNode->sasgnX();j=pNode->sasgnY();
// if(m_layers[l]->dir()!=m_layers[le]->dir() ) l = le+1; else l =
le;
// if(m_layers[l]->full(i,j) ) return false;
// }
// if(!mode) return true;
//
// if(pNode->type()==DUMMY)
// {
// if(m_layers[le]->dir()==m_layers[pNode->slayer()]->dir() ) ld =
le; else ld = le + 1;
// CNode* pChild = pNode->children().first();
// l = treel + pChild->upEdge()->relLayer();
// if(m_layers[l]->dir()==m_layers[pNode->slayer()]->dir() ) ldd =
l; else ldd = l + 1;
// }else
// {
// if(pParent->type()==DUMMY)
// {
//
if(m_layers[le]->dir()==m_layers[pParent->slayer()]->dir() ) ld = le + 1; else
ld = le;
// }else ld = le;
// if(pNode->type()!=STEINER) ldd = layerInd(*pNode);
// else ldd = le;
// }
//
// if(pParent->type()==DUMMY)
// {
// if(m_layers[le]->dir()==m_layers[pParent->slayer()]->dir() ) lu
= le; else lu = le + 1;
// l = treel + pParent->upEdge()->relLayer();
// if(m_layers[l]->dir()==m_layers[pParent->slayer()]->dir() ) luu
= l; else luu = l + 1;
// }else
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// {
// if(pNode->type()==DUMMY)
// {
// if(m_layers[le]->dir()==m_layers[pNode->slayer()]->dir()
) lu = le + 1; else lu = le;
// }else lu = le;
// if(pParent->type()!=STEINER) luu = layerInd(*pParent);
// else luu = le;
// }
//
// int m,n;
//
// for(int k=min(ld,ldd);k<=max(ld,ldd);k++)
// {
// if(pNode->type()!=DUMMY)
// {
// m =
(int)floor((pNode->x()-m_dieArea.left())/m_smallTileSize.width());
// n =
(int)floor((pNode->y()-m_dieArea.bottom())/m_smallTileSize.height());
//
if(m_layers[k]->dir()==VERTICAL){if(m_layers[k]->blocked(n,m) &&
m_layers[k]->blocked(n+1,m) ) return false;}
// else {if(m_layers[k]->blocked(m,n) &&
m_layers[k]->blocked(m+1,n) ) return false;}
// }else
// {
// m = pNode->sasgnX();
// n = pNode->sasgnY();
// if(m_layers[pNode->slayer()]->dir()==m_layers[k]->dir())
// {
// if(m_layers[k]->blocked(m,n)) return false;
// }
// else
// {
// if( m_layers[k]->blocked(n,m) &&
m_layers[k]->blocked(n,m-1) &&
// m_layers[k]->blocked(n+1,m) &&
m_layers[k]->blocked(n+1,m-1) ) return false;
// }
// }
// }
//
// for(int k=min(lu,luu);k<=max(lu,luu);k++)
// {
// if(pParent->type()!=DUMMY)
// {
// m =
(int)floor((pParent->x()-m_dieArea.left())/m_smallTileSize.width());
// n =
(int)floor((pParent->y()-m_dieArea.bottom())/m_smallTileSize.height());
//
if(m_layers[k]->dir()==VERTICAL){if(m_layers[k]->blocked(n,m) &&
m_layers[k]->blocked(n+1,m) ) return false;}
// else {if(m_layers[k]->blocked(m,n) &&
m_layers[k]->blocked(m+1,n) ) return false;}
// }else
// {
// m = pParent->sasgnX();
// n = pParent->sasgnY();
//
if(m_layers[pParent->slayer()]->dir()==m_layers[k]->dir())
// {
// if(m_layers[k]->blocked(m,n)) return false;
// }
// else
// {
// if( m_layers[k]->blocked(n,m) &&
m_layers[k]->blocked(n,m-1) &&
// m_layers[k]->blocked(n+1,m) &&
m_layers[k]->blocked(n+1,m-1) ) return false;
// }
// }
// }
// return true;
//}
//
//void CPrj::dumpHiddenOverflows()
//{
// for(unsigned int l=0;l<m_layers.size();l++)
// {
// CLayer* pLayer = m_layers[l];
// for(unsigned int i=0;i<pLayer->smallSizeX();i++)
// {
// for(unsigned int j=0;j<pLayer->smallSizeY();j++)
// {
// QValueList< QPair<CTree*,CNode*> > overflows
=pLayer->smallAsgnedEdges(i,j);
// while(overflows.count()>0)
// {
//        QPair<CTree*,CNode*> overflow =
overflows.front();
// overflows.pop_front();
// CTree* pTree = overflow.first;
// CNode* pNode = overflow.second;
// CNode* pChild =
pNode->children().first();
// int treel = pTree->layer();
// int l = pNode->slayer();
// int e = treel +
pChild->upEdge()->relLayer();
// if(e==l || e+1==l ) continue;
//
if(m_layers[e]->dir()!=m_layers[l]->dir() ) e++;
// for(int k=min(e,l+1);k<max(e+1,l);k++)
(m_layers[k]->m_smallSegHFlow[i][j])++;
// }
// }
// }
// }
//}
////void  CPrj::pp2()
////{
//// QDictIterator<CTree> treeIt(m_trees);
//// for(treeIt.toFirst();treeIt.current();++treeIt)
//// {
//// CTree* pTree = *treeIt;
//// QPtrList<CNode> nodes;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//// nodes.append( pTree->root() );
//// while(!nodes.isEmpty())
//// {
//// CNode* pNode = nodes.take(0);
//// CNode* pParent = pNode->parent();
//// CNode* pChild = pNode->children().first();
//// QPtrListIterator<CNode> n(pNode->children());
////       for(n.toFirst();n.current(); ++n) nodes.append( *n
);
//// if(pNode->type()!=DUMMY) continue;
//// int lo = pNode->slayer();int i=pNode->sasgnX();int
j=pNode->sasgnY();
//// double min =
double(m_layers[lo]->smallSegBlockage(i,j)+m_layers[lo]->smallSegFlow(i,
j))/double(m_layers[lo]->smallSegCap(i,j));
//// int l = lo;
//// double flow;
////
if(m_layers[lo]->smallSegBlockage(i,j)+m_layers[lo]->smallSegFlow(i,j)
<=m_layers[lo]->smallSegCap(i,j) ) continue;
//// for(unsigned int k=0;k<m_layers.size();k++)
//// {
//// if(m_layers[lo]->dir()!=m_layers[k]->dir())
continue;
//// flow =
double(m_layers[k]->smallSegBlockage(i,j)+m_layers[k]->smallSegFlow(i,
j))/double(m_layers[k]->smallSegCap(i,j));
//// if(flow<min)
//// {
//// min = flow;
//// l = k;
//// }
//// }
////
m_layers[lo]->ripUpFromSmall(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
////
m_layers[l]->smallSegAsgn(i,j,QPair<CTree*,CNode*>::QPair(pTree,pNode));
//// pNode->slayer(l);
//// if(pParent->type()==DUMMY)
//// {
//// if(pParent->slayer()>=l)
pNode->upEdge()->relLayer( l - pTree->layer() );
//// else pNode->upEdge()->relLayer( l -1 -
pTree->layer() );
//// }else   pNode->upEdge()->relLayer( l - pTree->layer() );
//// if(pChild->type()==DUMMY)
//// {
//// if(pChild->slayer()>=l)
pChild->upEdge()->relLayer( l - pTree->layer() );
//// else pChild->upEdge()->relLayer( l -1 -
pTree->layer() );
//// }else   pChild->upEdge()->relLayer( l - pTree->layer()
);
//// }
//// }
////
//// QFile dumpFile("PostProc.dmp");
//// dumpFile.open(IO_WriteOnly);
//// QTextStream dump(&dumpFile);
//// dump << *this;
//// dumpFile.close();
////}
//
////void CPrj::pp2()
////{
////bool improved;
//// do
//// {
//// improved = false;
//// unsigned long int count;
//// count = 0;
////   for(unsigned int l=0;l<m_layers.size();l++)
////  {
////  CLayer* pLayer = m_layers[l];
////  for(unsigned int i=0;i<pLayer->smallSizeX();i++)
////  {
////  for(unsigned int j=0;j<pLayer->smallSizeY();j++)
////  {
//// QValueList< QPair<CTree*,CNode*> >
overflows =pLayer->smallAsgnedEdges(i,j);
////  int numOverflows =
pLayer->smallSegBlockage(i,j)+overflows.count()-pLayer->smallSegCap(i,j);
//// if( numOverflows>0) count +=
numOverflows;
////  while(numOverflows>=0)
////  {
//// if(overflows.count()<=0)
break;
////        QPair<CTree*,CNode*> overflow =
overflows.front();
//// overflows.pop_front();
//// CTree* pTree =
overflow.first;
//// CNode* pNode =
overflow.second;
//// bool succ =
elim2(pTree,pNode);
////       if( succ )
{improved=true;numOverflows--;}
////  }
////  }
////  }
////  }
//// cout << "Overflows : \t" << count << "\n";
//// }while(improved);
////// file.close();
////}
////
////bool  CPrj::elim2(CTree* pTree, CNode* pNode)
////{
//// CNode* pParent = pNode->parent();
//// CNode* pChild = pNode->children().first();
////
//// int i,j;
//// CLayer* pLayer;
//// i=pNode->sasgnX();
//// j=pNode->sasgnY();
//// l=pNode->slayer();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
////
//// if(pParent->type()!=DUMMY)
//// {
////
//// }
////}
//



#include "cgroute.h"
#include "clayer.h"
#include "cnet.h"
#include "clayerpair.h"
#include <qdatetime.h>
#include "cnode.h"
#include "csink.h"
#include <qpixmap.h>

void CGRoute::randtim()
{
std::cout << "Generating random timing from minimal trees\n";
int numNets = m_nets.count();
int numLayers = m_layers.size();
int numSLayers = 0;
int layer1 = -1;
int layer2 = -1;

//temp, all layer 1 and build tree
{
QDictIterator<CNet> netIt(m_nets);
QValueList<CNet*> bubbleList;
//assign alloc nets to each layer
CLayerPair layer(m_layers[2],m_layers[3]);
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
  bubbleList.append( pNet );
pNet->layer( layer );
QPtrList<CNode> nodes = pNet->nodes();
QPtrListIterator<CNode> it(nodes);
for(it.toFirst();it.current();++it)
{
CNode* node = *it;
if( node->type()==ROOT || node->type()==DRIVER )
continue;
if( node->type()!=SINK) continue;
CSink* sink = (CSink*)node;
sink->rat( 1000.0 );
}
}
QValueList<CNet*> tmpList = bubbleList;
while( !tmpList.isEmpty() )
{
QValueListIterator<CNet*> it = tmpList.begin();
while(it!=tmpList.end()) {if( !iterate(*it,false) )
it=tmpList.remove(it); else it++;}
//sort in delay violation and insert buffers those with
violations
//
do{if(level++<6)bubbleSort(bubbleList);}while(insertBuffer(bubbleList));
}
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
awe(pNet);
again:
QPtrList<CNode> nodes = pNet->nodes();
QPtrListIterator<CNode> it(nodes);
for(it.toFirst();it.current();++it)
{
CNode* node = *it;
if( node->type()==ROOT || node->type()==DRIVER )
continue;
node->edge().disconnect();
if( node->type()==SINK)
{
CSink* sink = (CSink*)node;
double delta =
sink->delay()*(double(rand()%50)/100.0);
if(rand()%100>30) sink->rat(
sink->delay() + delta ); else sink->rat( sink->delay() - delta );
// sink->rat( sink->delay()*2 );
}else if( node->type()==STEINER )
{
pNet->removeNode( node );
goto again;
}else continue;
}

}
}
}

void CGRoute::slabi()
{
//randtim();
std::cout << "applying simultaneous steiner tree and layer assignment
algorithm\n";
int numNets = m_nets.count();
int numLayers = m_layers.size();
int numSLayers = 0;
int layer1 = -1;
int layer2 = -1;

for(int i=0;i<numLayers;i++)
{
if(m_layers[i]->selected()) numSLayers++;
if(layer1==-1 && m_layers[i]->selected() ) layer1 = i;
if(layer1!=-1 && layer2==-1 &&  m_layers[i]->selected() &&
(m_layers[layer1]->dir() != m_layers[i]->dir()) ) layer2=i;
    }
CLayerPair layer(m_layers[layer1],m_layers[layer2]);
int count = 0;
int alloc = numNets/(numSLayers-1); //initial num nets per layer pair
QDictIterator<CNet> netIt(m_nets);
```

```
QValueList<CNet*> bubbleList;
//assign alloc nets to each layer
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
  bubbleList.append( pNet );
pNet->layer( layer );
count++;
if(count>alloc) {layer++;count=0;}
}

// QFile F("resn.log");
//      F.open(IO_WriteOnly);
//      QTextStream f(&F);

QValueList<CNet*> tmpList = bubbleList;
int level = 0;
QTime t;
t.start();
while( !tmpList.isEmpty() )
{
std::cout << "routing " << tmpList.count() << " nets\n";
//f << "routing " << tmpList.count() << " nets\n";
//iterate each net once, that is connect one sink to the tree
QValueListIterator<CNet*> it = tmpList.begin();
while(it!=tmpList.end())
{
if( !iterate(*it,true) )
{
it=tmpList.remove(it);
}
else
{
// if((QString)(**it)=="net80")
(*it)->dump("itreated");
it++;
}
}
//sort in delay violation and insert buffers those with violations
if(level++<10)
{do{bubbleSort(bubbleList);/*removeBuffers(bubbleList);*/}
while(insertBuffer(bubbleList));}
}
//
bubbleSort(bubbleList);
  while(insertBuffer(bubbleList));
// removeBuffers();
removeBuffers(bubbleList);

{double tot = 0;
double usg[numLayers];
for(int i=0;i<numLayers;i++)
{
usg[i]=0;
   }
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
QPtrList<CNode> nodes = pNet->nodes();
QPtrListIterator<CNode> it(nodes);
for(it.toFirst();it.current();++it)
{
CNode* n = *it;
CEdge e = n->edge();
CLayer* vl = e.layer().vl();
CLayer* hl = e.layer().hl();
double x=0;
double y=0;
if( vl!=NULL ){
//std::cout << vl->index() << "\n";
usg[vl->index()] += e.yDist();
assert(vl->index()<numLayers);
x = e.yDist();}
if( hl!=NULL ){
//std::cout << hl->index() << "\n";
usg[hl->index()] += e.xDist();
assert(hl->index()<numLayers);
y = e.xDist();}
tot += x+y;
}
}
for(int i=0;i<numLayers;i++)
{
std::cout << m_layers[i]->name() << " = " << usg[i] << " " <<
usg[i]/tot << "\n";
   }}

//std::cout << "Time = " << t.elapsed();
//exit(0);

{
double maxDV = -INF;
double totDV = 0;
int numDV=0;
int numNet=0;
double totLen=0;
int numBuffers = 0;
QString maxDVNet;
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
// std::cout << "net " << QString(*pNet);
awe(pNet);
// std::cout << "\t Delay Violation = " << pNet->maxDV() << "\n";
numNet++;
double DV = pNet->maxDV();
if(DV>0) { numDV++;totDV+=DV;}
if(DV>maxDV) {maxDV = DV;maxDVNet = (QString)*pNet;}
totLen += pNet->length();
numBuffers += pNet->numBuffers();
// pNet->dump("");
}
std::cout << "Wire Length = " <<  totLen << "\n";
std::cout << "Number of Nets = " << numNet << "\n";
std::cout << "Number of Delay Violations = " << numDV << "\n";
std::cout << "Total Delay Violation = " << totDV << "\n";
std::cout << "Maximum Delay Violation = " << maxDV << " (" << maxDVNet
<< ")\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
std::cout << "Number of Buffers = " << numBuffers << "\n";
}
//exit(0);


// QFile F("sorted.log");
// F.open(IO_WriteOnly);
// QTextStream f(&F);
// for(int i=0;i<bubbleList.count();i++)
// f << i << "\t" << *((QString*)bubbleList[i]) << "\n";
// F.close();
// bubbleSort(bubbleList);
//  while(insertBuffer(bubbleList));
/* {
    f << "Elapsed Time :" << t.elapsed() << "\n";
int numBuff = 0;
int numViol = 0;
double totViol = 0;
double worstViol = 0;
double totDel = 0;
double length = 0;
for(netIt.toFirst();netIt.current();++netIt)
{
         double q;
           q=awe(*netIt);
// / if( (QString)(**netIt)=="net49")/ (*netIt)->dump("final tree");
if( q > 0 )
{
numViol++;
totViol += (*netIt)->maxDV();
if((*netIt)->maxDV()>worstViol) worstViol =
(*netIt)->maxDV();
}
numBuff += (*netIt)->numBuffers();
length += (*netIt)->length();
QPtrList<CNode> nodes = (*netIt)->nodes();
QPtrListIterator<CNode> nIt(nodes);
for(nIt.toFirst();nIt.current();++nIt)
if( (*nIt)->type()==SINK) totDel += (*nIt)->delay();
}
f << "Buffers :" << numBuff << "\n";
f << "Violations :" << numViol << "\n";
f << "Tot Viol :" << totViol << "\n";
f << "Worst Viol :" << worstViol << "\n";
f << "Tot Delay :" << totDel << "\n";
f << "Tot Length :" << length << "\n";
f << "Delay-Power :" << numBuff * totDel << "\n";
f << "Viol-Power :" << numBuff * totViol << "\n";
f << "Num Of Nets :" << netIt.count() << "\n";
//f.flush();
 }*/
  std::cout << "removing buffers\n";
     removeBuffers(bubbleList);
/*{
   f << "Elapsed Time :" << t.elapsed() << "\n";
int numBuff = 0;
int numViol = 0;
double totViol = 0;
double worstViol = 0;
double totDel = 0;
double length = 0;
for(netIt.toFirst();netIt.current();++netIt)
{
double q;
        q=awe(*netIt);
// / if((QString)(**netIt)=="net49")/ (*netIt)->dump("final tree
after buffer removal");
if( q > 0 )
        {      //    std::cout << "\tviol";
numViol++;
totViol += (*netIt)->maxDV();
if((*netIt)->maxDV()>worstViol) worstViol =
(*netIt)->maxDV();
}
numBuff += (*netIt)->numBuffers();
length += (*netIt)->length();
QPtrList<CNode> nodes = (*netIt)->nodes();
QPtrListIterator<CNode> nIt(nodes);
for(nIt.toFirst();nIt.current();++nIt)
if( (*nIt)->type()==SINK) totDel += (*nIt)->delay();
}
f << "Buffers :" << numBuff << "\n";
f << "Violations :" << numViol << "\n";
f << "Tot Viol :" << totViol << "\n";
f << "Worst Viol :" << worstViol << "\n";
f << "Tot Delay :" << totDel << "\n";
f << "Tot Length :" << length << "\n";
f << "Delay-Power :" << numBuff * totDel << "\n";
f << "Viol-Power :" << numBuff * totViol << "\n";
f << "Num Of Nets :" << netIt.count() << "\n";
for(int q=0;q<numLayers;q++)
{
CLayer* l = m_layers[q];
if(!l->selected()) continue;
f << l->name() << " Usage : " << l->usage() << "\n";
}
//f.flush();
 }
     F.close();*/
// exit(0);
removeBuffers();
{
double maxDV = -INF;
double totDV = 0;
int numDV=0;
int numNet=0;
double totLen=0;
int numBuffers = 0;
QString maxDVNet;
for(netIt.toFirst();netIt.current();++netIt)
{
CNet* pNet = *netIt;
// std::cout << "net " << QString(*pNet);
awe(pNet);
// std::cout << "\t Delay Violation = " << pNet->maxDV() << "\n";
numNet++;
double DV = pNet->maxDV();
if(DV>0) { numDV++;totDV+=DV;}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if(DV>maxDV) {maxDV = DV;maxDVNet = (QString)*pNet;}
totLen += pNet->length();
numBuffers += pNet->numBuffers();
}
std::cout << "Wire Length = " <<  totLen << "\n";
std::cout << "Number of Nets = " << numNet << "\n";
std::cout << "Number of Delay Violations = " << numDV << "\n";
std::cout << "Total Delay Violation = " << totDV << "\n";
std::cout << "Maximum Delay Violation = " << maxDV << " (" << maxDVNet
<< ")\n";
std::cout << "Number of Buffers = " << numBuffers << "\n";
}
//exit(0);
//initiialize routing tree
for(netIt.toFirst();netIt.current();++netIt) init(*netIt);
bubbleList.clear();
//exit(0);
}

void CGRoute::bubbleSort(QValueList<CNet*>& nets)
{
    // Goto last element;
QValueListIterator<CNet*> b,e;
b = nets.begin();
e = nets.end();
     QValueListIterator<CNet*> last = e;
     --last;
    // only one element or no elements ?
     if ( last == b ) return;

double totDV,totDV2,xDV,yDV,xDV2,yDV2;
CLayerPair lx,ly;
bool swap,swapped;

    // So we have at least two elements in here
do
{
swapped = false;
QValueListIterator<CNet*> x = e;
QValueListIterator<CNet*> y = x;
y--;
do
{
--x;--y;
lx = (*x)->layer();
ly = (*y)->layer();
if( lx==ly )
{
//on same layer
if( (*x)->maxDV() < (*y)->maxDV() )
{
swapped = true;
qSwap( *x, *y );
}
}
else if((lx.vl()==ly.vl()) || (lx.hl()==ly.hl()))
//on  neighboring layers
{
CLayer* xl,*yl;
if(lx.vl()==lx.vl()) { xl = lx.hl(); yl =
ly.hl(); } else { xl = lx.vl(); yl = ly.vl(); };
if(  xl->usage() > 1.1 * yl->usage() ) //if
upper layer used more
{
(*x)->layer( ly );
awe( *x );
swapped = true;
if( (*x)->maxDV() < (*y)->maxDV() )
qSwap( *x, *y );
}
else if( xl->usage() < 0.9 * yl->usage()
) //if   lower layer used more
{
(*y)->layer( lx );
awe( *y );
swapped = true;
if( (*x)->maxDV() < (*y)->maxDV() )
qSwap( *x, *y );
}
else if( (*x)->maxDV() < (*y)->maxDV() ) //usage
ok look for dv
{
xDV = (*x)->maxDV();
yDV = (*y)->maxDV();
totDV = 0;
if( xDV>=0 ) totDV += xDV;
if( yDV>=0 ) totDV += yDV;
//temporarily swap
(*x)->layer( ly );
awe( *x );
(*y)->layer( lx );
awe( *y );
xDV2 = (*x)->maxDV();
yDV2 = (*y)->maxDV();

totDV2 = 0;
if( xDV2>=0 ) totDV2 += xDV2;
if( yDV2>=0 ) totDV2 += yDV2;
swap = false;
//if good swap
// if( ::max(xDV2,yDV2 < ::max(xDV,yDV) )
swap = true;
if( totDV2 < totDV ) swap = true;
if( swap )
{
swapped = true;
qSwap( *x, *y );
}
else
{   // rollback, not good swap
(*x)->layer( lx );
awe(*x);
(*y)->layer( ly );
awe(*y);
}
}
}
}while( y != b );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}while(swapped);
}

#include "cgroute.h"
#include "cnet.h"
#include "cnode.h"
#include "csink.h"
#include "cedge.h"

bool CGRoute::iterate(CNet* pNet, bool mode)
{
// std::cout << "\n" << *pNet << "\n";
    QPtrList<CNode> nodes = pNet->nodes();
QValueList<CEdge> edges = pNet->edges();
double bestDV = -2*INF;
double bestLen = 2*INF;
double DV,Len;
CNode *bestLNode,*bestDNode;
bestLNode = bestDNode = NULL;
CEdge bestLEdge,bestDEdge;
bool prio = false;
QPtrListIterator<CNode> nIt(nodes);
QValueListIterator<CEdge> eIt;
bool iterated = false;
for(nIt.toFirst();nIt.current();++nIt)
{
nodeType tp = (*nIt)->type();
if((*nIt)->parent()!=NULL || tp==DRIVER) continue;

double bestDVin = 2*INF;
double bestLenin = 2*INF;
CEdge bestLEdgein,bestDEdgein;
bool prioin =false;
bool iteratedin = false;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
pNet->joinNodeAtEdge( *nIt, *eIt );
DV = awe( pNet );
Len = pNet->length();
// /* if((QString)(*pNet)=="net49")*/ pNet->dump("candidate
iteration");
if( mode )
{
if( DV<=0 && false )
{
if(Len < bestLenin )
{
bestLenin = Len;
bestLEdgein = *eIt;
iteratedin = true;
prioin = true;
}
}else
{
if(DV < bestDVin )
{
bestDVin = DV;
bestDEdgein = *eIt;
iteratedin = true;
}

}
}else
{
if( Len < bestLenin )
{
bestLenin = Len;
bestLEdgein = *eIt;
iteratedin = true;
prioin = true;
}
}
pNet->disjoinNodeFromEdge(*nIt, *eIt );
}
if(iteratedin)
{
  if(prioin)
{
if(bestLenin < bestLen)
{
bestLen = bestLenin;
bestLNode = *nIt;
bestLEdge = bestLEdgein;
iterated = true;
}
}else
{
if(bestDVin > bestDV)
{
bestDV = bestDVin;
bestDEdge = bestDEdgein;
bestDNode = *nIt;
iterated = true;
prio = true;
}
}
}



}
if(iterated)
{
if( prio ) pNet->joinNodeAtEdge( bestDNode, bestDEdge );
else pNet->joinNodeAtEdge( bestLNode, bestLEdge );
awe(pNet);
// /* if((QString)(*pNet)=="net49")*/ pNet->dump("iterated");
}
awe(pNet);
return iterated;
}

#include "cgroute.h"
#include "clayer.h"
#include "cnet.h"
#include "gwire.h"
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CLayer* CGRoute::layer(const QString name) const
{
for(unsigned int i=0;i<m_layers.size();i++)
if(m_layers[i]->name()==name) return m_layers[i];
return NULL;
}

int CGRoute::layerInd(const QString name) const
{
for(unsigned int i=0;i<m_layers.size();i++)
if(m_layers[i]->name()==name) return i;
return -1;
}

int CGRoute::numTileOverflows() const
{
int o = 0;
for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* layer = m_layers[i];
for(int track=0;track<layer->numtTracks();track++)
for(int edge=0;edge<layer->numtEdges();edge++)
{
int b = layer->tileBlock(track,edge);
int f = layer->tileFlow(track,edge);
int c = layer->tileCap();
if(f+b>c) o += f+b-c;
}
}
return o;
}

int CGRoute::numCellOverflows() const
{
int o = 0;
for(unsigned int i=0;i<m_layers.size();i++)
{
CLayer* layer = m_layers[i];
for(int track=0;track<layer->numcTracks();track++)
for(int edge=0;edge<layer->numcEdges();edge++)
{
int b = layer->cellBlock(track,edge);
int f = layer->cellFlow(track,edge);
int c = layer->cellCap();
if(f+b>c) o += f+b-c;
}
}
return o;
}

int CGRoute::numPadViolations() const
{
int c = 0;
QDictIterator<CNet> nIt(m_nets);
for(nIt.toFirst();nIt.current();++nIt)
{
CNet* net = *nIt;
net->first();
while(net->current())
{
gwire* wire = net->current();
c += wire->numPadViolations();
net->next();
}
}
return c;
}

bool CGRoute::selectLayer(const QString layer)
{
unsigned int size = m_layers.size();
CLayer* pLayer = new CLayer(layer,size, true);
m_layers.resize( size+1);
m_layers.insert(size,pLayer);
return true;
}

bool CGRoute::unselectLayer(const QString layer)
{
unsigned int size = m_layers.size();
CLayer* pLayer = new CLayer(layer,size, false);
m_layers.resize( size+1);
m_layers.insert(size,pLayer);
return true;
}

//get unit size cell rectangle
QRect CGRoute::grect(const CPoint p) const
{
int x = (int)floor((p.x()-m_dieArea.left())/m_gCell.width());
int y = (int)floor((p.y()-m_dieArea.bottom())/m_gCell.height());
//if(x==58 && y==100) {std::cout << "1 " << p.x() << " " << p.y() <<
"\n";exit(0);}
return QRect(x,y,1,1);
}

//get cell rectangle that covers tile which cell is in
QRect CGRoute::grect(const QPoint p) const
{
//this is tile
int x = (int)floor(p.x()/m_gTile.width());
int y = (int)floor(p.y()/m_gTile.height());
//if(x==58 && y==100) {std::cout << "2 " << p.x() << " " << p.y() <<
"\n";exit(0);}
return grect(x,y);
}

//get cell rectangle that covers tile
QRect CGRoute::grect(const int x, const int y) const
{
return QRect(QPoint(x*m_gTile.width(),y*m_gTile.height()),m_gTile);
}

//get cell point
QPoint CGRoute::gpoint(const CPoint p) const
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
int x = (int)floor((p.x()-m_dieArea.left())/m_gCell.width());
int y = (int)floor((p.y()-m_dieArea.bottom())/m_gCell.height());
return QPoint(x,y);
}

//get tile point
QPoint CGRoute::tpoint(const CPoint p) const
{
int x = (int)floor((p.x()-m_dieArea.left())/(m_gCell.width() *
m_gTile.width()));
int y = (int)floor((p.y()-m_dieArea.bottom())/(m_gCell.height() *
m_gTile.height()));
return QPoint(x,y);
}

#include "cnode.h"
QPtrList<gwire> CGRoute::wires() const
{
   QPtrList<gwire> wires;
QDictIterator<CNet> nIt(m_nets);
for(nIt.toFirst();nIt.current();++nIt)
{
CNet* net = *nIt;
QPtrListIterator<gwire> wIt(*net);
for(wIt.toFirst();wIt.current();++wIt) wires.append(*wIt);
}
return wires;
}

#include <qwmatrix.h>
#include <qcanvas.h>
#include <qpixmap.h>
#include "clayer.h"

CLayer::CLayer(const QString name, const int index, const bool selected)
:m_name(name)
,m_selected(selected)
,m_index(index)
,m_hasDirection(false)
,m_hasPitch(false)
,m_hasWidth(false)
,m_hasRes(false)
,m_hasCap(false)
,m_next(NULL)
,m_sameNext(NULL)
,m_prev(NULL)
,m_samePrev(NULL)
{
}

CLayer::~CLayer()
{
m_cellBlock.clear();
m_cellFlow.clear();
m_tileBlock.clear();
m_tileFlow.clear();
}

QPair<QRect,QRect> CLayer::divideAtTrack(const  int  track) const
{
QPair<QRect,QRect> div;
if(m_dir==VERTICAL) {div.first = QRect(0,0,m_cwidth,track);  div.second
= QRect(0,track,m_cwidth,m_cheight-track);}
else if(m_dir==HORIZONTAL) {div.first = QRect(0,0,track,m_cheight);
div.second = QRect(track,0,m_cwidth-track,m_cheight);}
return div;
}

void CLayer::init(const CRect dieArea, const CSize gCell, const QSize gTile)
{
m_dieArea = dieArea;
m_gTile = gTile;
m_gCell = gCell;
m_twidth =
(int)ceil(m_dieArea.width()/(m_gTile.width()*m_gCell.width()));
m_theight =
(int)ceil(m_dieArea.height()/(m_gTile.height()*m_gCell.height()));
m_cwidth =  m_twidth * m_gTile.width();
m_cheight =   m_theight * m_gTile.height();
switch(m_dir)
{
case VERTICAL:
m_numtTracks = m_theight + 1;
m_numcTracks = m_cheight + 1;
m_numtEdges = m_twidth;
m_numcEdges = m_cwidth;
m_cellCap =  (int)floor(m_gCell.width()/m_pitch);
m_tileCap = m_cellCap * m_gTile.width();
break;
case HORIZONTAL:
m_numtTracks = m_twidth + 1;
m_numcTracks = m_cwidth + 1;
m_numtEdges = m_theight;
m_numcEdges = m_cheight;
m_cellCap = (int)floor(m_gCell.height()/m_pitch);
m_tileCap = m_cellCap * m_gTile.height();
break;
default: std::cout << "CLayer::init() not implemented case\n";
}
m_cellBlock.resize(m_numcTracks);
m_cellFlow.resize(m_numcTracks);
for(int i=0;i<m_numcTracks;i++)
{m_cellBlock[i].resize(m_numcEdges, 0);m_cellFlow[i].resize(m_numcEdges,
0);}
m_tileBlock.resize(m_numtTracks);
m_tileFlow.resize(m_numtTracks);
for(int i=0;i<m_numtTracks;i++)
{m_tileBlock[i].resize(m_numtEdges, 0);m_tileFlow[i].resize(m_numtEdges,
0);}
}
/*
```

```
void CLayer::init(const QValueList<CRect> rects)
{

//build a canvas so that all rects are not overlaping in the canvas
  QValueListConstIterator<CRect> rectIt;
for(rectIt=rects.begin();rectIt!=rects.end();rectIt++)
{
CRect rect = *rectIt;
if(!rect.intersects(m_dieArea)) continue;
    rect = rect & m_dieArea;
int colS =
(int)floor((rect.left()-m_dieArea.left())/m_gCell.width());
int colE =
(int)floor((rect.right()-m_dieArea.left())/m_gCell.width()());
int rowS =
(int)floor((rect.bottom()-m_dieArea.bottom())/m_gCell.height());
int rowE =
(int)floor((rect.top()-m_dieArea.bottom())/m_gCell.height());
int trackS = (m_dir==VERTICAL)? rowS+1 : colS+1;
int trackE = (m_dir==VERTICAL)? rowE : colE;
int edgeS = (m_dir==VERTICAL)? colS : rowS;
int edgeE = (m_dir==VERTICAL)? colE : rowE;
for(int track=trackS;track<=trackE;track++)
{
for(int edge=edgeS;edge<=edgeE;edge++)
{

}
}

}
}
*/
void CLayer::init(const QValueList<CRect> rects)
{
//build a canvas so that all rects are not overlaping in the canvas
QWMatrix wm;
wm.translate( (int)floor(-RES*m_dieArea.left()),
(int)floor(-RES*m_dieArea.bottom()) );
QCanvas* canvas;
QPixmap pixmap(1,1,1);
  canvas = new QCanvas(
QPixmap(1,1,1),m_cwidth,m_cheight,int(m_gCell.width()*RES),
int(m_gCell.height()*RES));
  QValueListConstIterator<CRect> rectIt;
unsigned long int c = 0;
for(rectIt=rects.begin();rectIt!=rects.end();rectIt++)
{
std::cout << "all rects " << c++/* << " / " <<
canvas->allItems().count()*/ << "\n";
CRect rect = *rectIt;
if(!rect.intersects(m_dieArea)) continue;
    rect = rect & m_dieArea;
QRect qRect = (rect * RES).toQRect();
qRect = wm.mapRect( qRect );
if(qRect.isNull() || qRect.isEmpty() || !qRect.isValid() )
continue;
QCanvasRectangle* cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );
QCanvasItemList collidedRects = cRect->collisions( true );
while( !collidedRects.isEmpty() )
{
QCanvasRectangle* collidedcRect =
(QCanvasRectangle*)collidedRects.front();
collidedRects.pop_front();
if( !cRect->rect().intersects( collidedcRect->rect()))
continue;
if( collidedcRect->rect().contains( cRect->rect()
)){delete cRect;break; }
if( cRect->rect().contains( collidedcRect->rect()
)){delete collidedcRect;continue;}
QValueList<QRect> clipRects = ::clipRect(
collidedcRect->rect(), cRect->rect() );
delete collidedcRect;
while(!clipRects.isEmpty())
{
if(!clipRects.front().isNull() &&
!clipRects.front().isEmpty() && clipRects.front().isValid())
{
collidedcRect = new QCanvasRectangle(
clipRects.front(), canvas );
collidedcRect->setVisible( true );
}
clipRects.pop_front();
}
}
}
//std::cout << "Layer done clipping\n";
//fill outside of the die area
{
double left = m_dieArea.right();
double width = m_cwidth * m_gCell.width() -m_dieArea.width();
double bottom = m_dieArea.top();
double height = m_cheight * m_gCell.height() -
m_dieArea.height();
{CRect rect(left,m_dieArea.bottom(),width,m_dieArea.height());
QRect qRect = (rect * RES).toQRect();
qRect = wm.mapRect( qRect );
QCanvasRectangle* cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );}
{CRect rect(m_dieArea.left(),bottom,m_dieArea.width(),height);
QRect qRect = (rect * RES).toQRect();
qRect = wm.mapRect( qRect );
QCanvasRectangle* cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );}
{CRect rect(left,bottom,width,height);
QRect qRect = (rect * RES).toQRect();
qRect = wm.mapRect( qRect );
QCanvasRectangle* cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );}
}

//if(m_name=="METAL6")
//{
// QFile file(name()+".keepout");
// file.open(IO_WriteOnly);
// QTextStream f(&file);
// QValueListIterator<QCanvasItem*> it;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//  QCanvasItemList items = canvas->allItems();
// for(it=items.begin();it!=items.end();it++)
// {
// QCanvasRectangle* cr = (QCanvasRectangle*)*it;
// QRect r = cr->rect();
// f << "(keepout (rect metal8";
// f << " " << r.left()/100.0 << " " << r.top()/100.0 +
m_dieArea.bottom();
// f << " " << r.right()/100.0 << " " << r.bottom()/100.0 +
m_dieArea.bottom();
// f << "))\n";
// }
// file.close();
//}
//calculate usage by blockages
QCanvasItemList items = canvas->allItems();
QValueListIterator<QCanvasItem*> it;
QRect die(0,0,int(m_dieArea.width()*RES),int(m_dieArea.height()*RES));
m_len = m_dieArea.height() * m_dieArea.width() / m_pitch;
m_bUsage = 0;
m_nUsage = 0;
for(it=items.begin();it!=items.end();it++)
{
QRect rect = ((QCanvasRectangle*)*it)->rect();
rect &= die;
m_bUsage += (double)rect.width() * (double)rect.height() /
(RES*RES) / m_pitch;
}

//calculate gcell blockages
for(int track=0;track<m_numcTracks;track++)
{
for(int edge=0;edge<m_numcEdges;edge++)
{
//set all blocked for die Area  boundaries
if(track==0 || track==m_numcTracks-1)
{
m_cellBlock[track][edge] = m_cellCap;
m_cellFlow[track][edge] = 0;
continue;
}

//calculate it here from blocked area
double left,bottom,width,height;
left = bottom = width = height = -1;
QRect qRect,qRectLo,qRectHi;
switch(m_dir)
{
case VERTICAL:
left = edge * m_gCell.width();
width = m_gCell.width();
bottom = (track-1) * m_gCell.height();
height = 2 * m_gCell.height();
qRect =  (CRect(left,bottom,width,height) * RES
).toQRect();
qRectLo = (CRect(left,bottom,width,height/2) *
RES ).toQRect();
qRectHi =
(CRect(left,bottom+height/2,width,height/2) * RES ).toQRect();
break;
case HORIZONTAL:
left = (track-1) * m_gCell.width();
width = 2 * m_gCell.width();
bottom = edge * m_gCell.height();
height = m_gCell.height();
qRect =(CRect(left,bottom,width,height) * RES
).toQRect();
qRectLo =  (CRect(left,bottom,width/2,height) *
RES ).toQRect();
qRectHi =
(CRect(left+width/2,bottom,width/2,height) * RES ).toQRect();
break;
default: std::cout << "CLayer::init() not implemented
case\n";
}//two cell rectangle
double totArea = (double)qRect.width() *
(double)qRect.height() / 2.0;
double blkAreaLo = 0;
double blkAreaHi = 0;
QCanvasRectangle* cRect = new QCanvasRectangle( qRect,
canvas );
cRect->setVisible( true );
QCanvasItemList collidedRects = cRect->collisions( true
);
while( !collidedRects.isEmpty() )
{
QCanvasRectangle* collidedcRect =
(QCanvasRectangle*)collidedRects.front();
collidedRects.pop_front();
if( !qRect.intersects( collidedcRect->rect()))
continue;
if( qRectLo.intersects( collidedcRect->rect()))
{
QRect bRect = qRectLo &
collidedcRect->rect();
blkAreaLo += (double)bRect.width() *
(double)bRect.height();
}
if( qRectHi.intersects( collidedcRect->rect()))
{
QRect bRect = qRectHi &
collidedcRect->rect();
blkAreaHi += (double)bRect.width() *
(double)bRect.height();
}
}
delete cRect;
int blkLo = ::min(int(m_cellCap * (blkAreaLo /
totArea)), m_cellCap);
int blkHi = ::min(int(m_cellCap * (blkAreaHi /
totArea)), m_cellCap);
int blk1 = ::max(blkLo,blkHi);
//calculate here from blocked line
switch(m_dir)
{
case VERTICAL:
left = edge * m_gCell.width();
width = m_gCell.width();
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
bottom = track * m_gCell.height() - 1;
height = 2;
break;
case HORIZONTAL:
left = track * m_gCell.width() - 1;
width = 2;
bottom = edge * m_gCell.height();
height = m_gCell.height();
break;
default: std::cout << "CLayer::init() not implemented
case\n";
}//two cell rectangle
qRect = (CRect(left,bottom,width,height) *
RES).toQRect();
double totLine = (m_dir==VERTICAL)? qRect.width() :
qRect.height();
double blkLine = 0;
cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );
collidedRects = cRect->collisions( true );
while( !collidedRects.isEmpty() )
{
QCanvasRectangle* collidedcRect =
(QCanvasRectangle*)collidedRects.front();
collidedRects.pop_front();
if( !qRect.intersects( collidedcRect->rect()))
continue;
QRect bRect = qRect & collidedcRect->rect();
blkLine = (m_dir==VERTICAL)? blkLine +
bRect.width() : blkLine + bRect.height();
}
delete cRect;
int blk2 = ::min(int(m_cellCap * (blkLine / totLine)),
m_cellCap);
m_cellBlock[track][edge] = ::max(blk1,blk2);
if(m_cellBlock[track][edge]>=m_cellCap*0.99)
m_cellBlock[track][edge]=m_cellCap;
m_cellFlow[track][edge] = 0;
//std::cout << "gCELL ";
//std::cout << track*m_numcEdges + edge << " / ";
//std::cout << m_numcEdges*m_numcTracks << "\n";
}
}
// std::cout << "done gcell \n";
//calculate gtile blockages
for(int track=0;track<m_numtTracks;track++)
{
for(int edge=0;edge<m_numtEdges;edge++)
{
//set all blocked for die Area  boundaries
if(track==0 || track==m_numtTracks-1)
{
m_tileBlock[track][edge] = m_tileCap;
m_tileFlow[track][edge] = 0;
continue;
}
//calculate it here from blocked area
double left,bottom,width,height;
left = bottom = width = height = -1;
QRect qRect,qRectLo,qRectHi;
switch(m_dir)
{
case VERTICAL:
left = edge * m_gTile.width() * m_gCell.width();
width = m_gTile.width() * m_gCell.width();
bottom = (track-1) * m_gTile.height() *
m_gCell.height();
height = 2 * m_gTile.height() *
m_gCell.height();
qRect =  (CRect(left,bottom,width,height) * RES
).toQRect();
qRectLo = (CRect(left,bottom,width,height/2) *
RES ).toQRect();
qRectHi =
(CRect(left,bottom+height/2,width,height/2) * RES ).toQRect();
break;
case HORIZONTAL:
left = (track-1) * m_gTile.width() *
m_gCell.width();
width = 2 * m_gTile.width() * m_gCell.height();
bottom = edge * m_gTile.height() *
m_gCell.height();
height = m_gTile.height() * m_gCell.height();
qRect = (CRect(left,bottom,width,height) * RES
).toQRect();
qRectLo =  (CRect(left,bottom,width/2,height) *
RES ).toQRect();
qRectHi =
(CRect(left+width/2,bottom,width/2,height) * RES ).toQRect();
break;
default: std::cout << "CLayer::init() not implemented
case\n";
}//two cell rectangle
double totArea = (double)qRect.width() *
(double)qRect.height() / 2.0;
double blkAreaLo = 0;
double blkAreaHi = 0;
QCanvasRectangle* cRect = new QCanvasRectangle( qRect,
canvas );
cRect->setVisible( true );
QCanvasItemList collidedRects = cRect->collisions( true
);
while( !collidedRects.isEmpty() )
{
QCanvasRectangle* collidedcRect =
(QCanvasRectangle*)collidedRects.front();
collidedRects.pop_front();
if( !qRect.intersects( collidedcRect->rect()))
continue;
if( qRectLo.intersects( collidedcRect->rect()))
{
QRect bRect = qRectLo &
collidedcRect->rect();
blkAreaLo += (double)bRect.width() *
(double)bRect.height();
}
if( qRectHi.intersects( collidedcRect->rect()))
{
```

```
QRect bRect = qRectHi &
collidedcRect->rect();
blkAreaHi += (double)bRect.width() *
(double)bRect.height();
}
}
delete cRect;
int blkLo = ::min(int(m_tileCap * (blkAreaLo /
totArea)), m_tileCap);
int blkHi = ::min(int(m_tileCap * (blkAreaHi /
totArea)), m_tileCap);
int blk1 = ::max(blkLo,blkHi);
//calculate here from blocked line
switch(m_dir)
{
case VERTICAL:
left = edge * m_gTile.width() * m_gCell.width();
width = m_gTile.width() * m_gCell.width();
bottom = track * m_gTile.height() *
m_gCell.height() - 1;
height = 2;
break;
case HORIZONTAL:
left = track * m_gTile.width() * m_gCell.width()
- 1;
width = 2;
bottom = edge * m_gTile.height() *
m_gCell.height();
height = m_gTile.height() * m_gCell.height();
break;
default: std::cout << "CLayer::init() not implemented
case\n";
}//two cell rectangle
qRect = (CRect(left,bottom,width,height) *
RES).toQRect();
double totLine = (m_dir==VERTICAL)? qRect.width() :
qRect.height();
double blkLine = 0;
cRect = new QCanvasRectangle( qRect, canvas );
cRect->setVisible( true );
collidedRects = cRect->collisions( true );
while( !collidedRects.isEmpty() )
{
QCanvasRectangle* collidedcRect =
(QCanvasRectangle*)collidedRects.front();
collidedRects.pop_front();
if( !qRect.intersects( collidedcRect->rect()))
continue;
QRect bRect = qRect & collidedcRect->rect();
blkLine = (m_dir==VERTICAL)? blkLine +
bRect.width() : blkLine + bRect.height();
}
delete cRect;
int blk2 = ::min(int(m_tileCap * (blkLine / totLine)),
m_tileCap);
m_tileBlock[track][edge] = max(blk1,blk2);
if(m_tileBlock[track][edge]>=m_tileCap*0.99)
m_tileBlock[track][edge]=m_tileCap;
m_tileFlow[track][edge] = 0;
//std::cout << "gTILE ";
//std::cout << track*m_numtEdges + edge << " / ";
//std::cout << m_numtEdges*m_numtTracks << "\n";
}
}
// std::cout << "done gtile \n";
delete canvas;
}

void CLayer::paint(QPainter* p)
{
int off = 0;
if(m_name=="METAL3")
{
p->setPen("black");
off = -1;
}else if(m_name=="METAL4")
{
p->setPen("yellow");
off = -1;
}else if(m_name=="METAL5")
{
p->setPen("red");
off = 0;
}else if(m_name=="METAL6")
{
p->setPen("green");
off = +1;
} else if(m_name=="METAL7")
{
p->setPen("blue");
off = +1;
}

switch(m_dir)
{
case VERTICAL:
{
for(int i=0;i<m_numcTracks;i++) for(int
j=0;j<m_numcEdges;j++)
{

p->drawPoint(int(j*m_gCell.width()/10),int(i*m_gCell.height()/10));

p->drawPoint(int((j+1)*m_gCell.width()/10),int(i*m_gCell.height()/10));
if(cellFull(i,j))
{

p->drawLine(int(j*m_gCell.width()/10),int(i*m_gCell.height()/10)+off,

int((j+1)*m_gCell.width()/10),int(i*m_gCell.height()/10)+off);
if(cellOverflows(i,j))
{

p->drawEllipse(int(j*m_gCell.width()/10)+3,int(i*m_gCell.height()/10)+off-2,4,
4);
}
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
break;
}
case HORIZONTAL:
{
for(int i=0;i<m_numcTracks;i++) for(int
j=0;j<m_numcEdges;j++)
{

p->drawPoint(int(i*m_gCell.width()/10),int(j*m_gCell.height()/10));

p->drawPoint(int(i*m_gCell.width()/10),int((j+1)*m_gCell.height()/10));
if(cellFull(i,j))
{

p->drawLine(int(i*m_gCell.width()/10)+off,int(j*m_gCell.height()/10),

int(i*m_gCell.width()/10)+off,int((j+1)*m_gCell.height()/10));
if(cellOverflows(i,j))
{

p->drawEllipse(int(i*m_gCell.width()/10)+off-2,int(j*m_gCell.height()/10)+3,4,
4);
}
}
}
break;
default: std::cout << "CLayer::paint() not implemented case\n";
}
}
}


#include "clayerpair.h"
#include "clayer.h"

CLayerPair::CLayerPair(CLayer* l1, CLayer* l2)
:QPair<CLayer*,CLayer*>::QPair(l1,l2)
{

}


CLayerPair::CLayerPair(const CLayerPair& layer)
:QPair<CLayer*,CLayer*>::QPair(layer.first,layer.second)
{
}

CLayerPair::~CLayerPair()
{
}

CLayerPair& CLayerPair::operator++(int)
{
CLayer* ll = this->ll();
if(ll!=NULL)
{
CLayer* next = ll->next(true);
if(next!=NULL)
{
if(first==ll) first = next;
else if(second==ll) second = next;
}
}
return *this;
}

CLayerPair& CLayerPair::operator--(int)
{
CLayer* ul = this->ul();
if(ul!=NULL)
{
CLayer* prev = ul->prev(true);
if(prev!=NULL && prev->selected())
{
if(first==ul) first = prev;
else if(second==ul) second = prev;
}
}
return *this;
}

bool operator==(const CLayerPair& lp1, const CLayerPair& lp2)
{
return lp1.vl()==lp2.vl() && lp1.hl()==lp2.hl();
}

CLayer* CLayerPair::ll() const
{
if(first==NULL && second==NULL ) return NULL;
else if(first!=NULL && second==NULL ) return first;
else if(first==NULL && second!=NULL ) return second;
else if(*first > *second ) return second;
else if(*first < *second ) return first;
else if(*first==*second ) return first;
else return NULL;
}

CLayer* CLayerPair::ul() const
{
if(first==NULL && second==NULL ) return NULL;
else if(first!=NULL && second==NULL ) return first;
else if(first==NULL && second!=NULL ) return second;
else if(*first > *second ) return first;
else if(*first < *second ) return second;
else if(*first==*second ) return second;
else return NULL;
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
CLayer* CLayerPair::vl() const
{
if(first !=NULL && first->dir()==VERTICAL ) return first;
else if(second!=NULL && second->dir()==VERTICAL ) return second;
else return NULL;
}

CLayer* CLayerPair::hl() const
{
if(first!=NULL && first->dir()==HORIZONTAL ) return first;
else if(second!=NULL && second->dir()==HORIZONTAL ) return second;
else return NULL;
}



#include "clef.h"
#include "cleflayer.h"
#include "clefmacro.h"
#include "clefvia.h"

CLef::CLef()
:m_hasVersion(false)
,m_hasCaseSensitive(false)
,m_hasBusBitChars(false)
,m_hasDividerChar(false)
,m_layerIndex(0)
{
m_layers.setAutoDelete(true);
m_macros.setAutoDelete(true);
m_vias.setAutoDelete(true);
}

CLef::~CLef()
{
m_layers.clear();
m_macros.clear();
m_vias.clear();
}

bool CLef::load(const QString fileName)
{
FILE* lefFile;
lefrInit();
lefrSetAntennaInoutCbk(  lefrAntennaInoutCbk );
lefrSetAntennaInputCbk( lefrAntennaInputCbk );
lefrSetAntennaOutputCbk( lefrAntennaOutputCbk );
lefrSetBusBitCharsCbk( lefrBusBitCharsCbk );
lefrSetClearanceMeasureCbk( lefrClearanceMeasureCbk );
lefrSetDividerCharCbk( lefrDividerCharCbk );
lefrSetLibraryEndCbk( lefrLibraryEndCbk );
lefrSetLayerCbk( lefrLayerCbk );
lefrSetMacroBeginCbk( lefrMacroBeginCbk );
lefrSetMacroCbk( lefrMacroCbk );
lefrSetMacroClassTypeCbk( lefrMacroClassTypeCbk );
lefrSetObstructionCbk( lefrObstructionCbk );
lefrSetPinCbk( lefrPinCbk );
lefrSetManufacturingCbk( lefrManufacturingCbk );
lefrSetCaseSensitiveCbk( lefrCaseSensitiveCbk );
lefrSetNoWireExtensionCbk( lefrNoWireExtensionCbk );
lefrSetNonDefaultCbk( lefrNonDefaultCbk );
lefrSetPropBeginCbk( lefrPropBeginCbk );
lefrSetPropCbk( lefrPropCbk );
lefrSetPropEndCbk( lefrPropEndCbk );
lefrSetSpacingBeginCbk( lefrSpacingBeginCbk );
lefrSetSpacingCbk( lefrSpacingCbk );
lefrSetSpacingEndCbk( lefrSpacingEndCbk );
lefrSetSiteCbk( lefrSiteCbk );
lefrSetUnitsCbk( lefrUnitsCbk );
lefrSetUseMinSpacingCbk( lefrUseMinSpacingCbk );
lefrSetVersionCbk( lefrVersionCbk );
lefrSetVersionStrCbk( lefrVersionStrCbk );
lefrSetViaCbk( lefrViaCbk );
lefrSetViaRuleCbk( lefrViaRuleCbk );
  lefFile = fopen(fileName,"r");
if( lefFile==NULL)
{
std::cout << "ERROR : couldn't open " <<  fileName << "\n";
return false;
}
 lefrRead(lefFile,fileName, (void*)this);
  fclose(lefFile);
return true;
}


bool CLef::save(const QString fileName)
{
FILE* lefFile;
lefFile = fopen(fileName,"w");
if( lefFile==NULL)
{
std::cout << "ERROR : couldn't open " <<  fileName << "\n";
return false;
}
lefwInitCbk(lefFile);
lefwSetAntennaCbk(  lefwAntennaCbk );
lefwSetBusBitCharsCbk( lefwBusBitCharsCbk );
lefwSetClearanceMeasureCbk( lefwClearanceMeasureCbk );
lefwSetDividerCharCbk( lefwDividerCharCbk );
lefwSetExtCbk( lefwExtCbk );
lefwSetEndLibCbk( lefwEndLibCbk );
lefwSetLayerCbk( lefwLayerCbk );
lefwSetMacroCbk( lefwMacroCbk );
lefwSetManufacturingGridCbk( lefwManufacturingGridCbk );
lefwSetCaseSensitiveCbk( lefwCaseSensitiveCbk );
lefwSetNonDefaultCbk( lefwNonDefaultCbk );
lefwSetNoWireExtensionCbk( lefwNoWireExtensionCbk );
lefwSetPropDefCbk( lefwPropDefCbk );
lefwSetSiteCbk( lefwSiteCbk );
lefwSetSpacingCbk( lefwSpacingCbk );
lefwSetUnitsCbk( lefwUnitsCbk );
lefwSetUseMinSpacingCbk( lefwUseMinSpacingCbk );
lefwSetVersionCbk( lefwVersionCbk );
lefwSetViaCbk( lefwViaCbk );
lefwSetViaRuleCbk( lefwViaRuleCbk );
```

```
lefwWrite(lefFile,fileName, (void*)this);
fclose(lefFile);

return true;
}

CLefLayer* CLef::layer(const QString name) const
{
return m_layers[name];
}

QPtrVector<CLefLayer> CLef::routingLayers() const
{
int numRoutingLayers = 0;
QDictIterator<CLefLayer> it(m_layers);
QPtrVector<CLefLayer> dummy(m_layers.count());
for(it.toFirst();it.current();++it)
{
dummy.insert((*it)->index(), *it );
if((*it)->type()==ROUTING) numRoutingLayers++;
}
QPtrVector<CLefLayer> layers(numRoutingLayers);
int ind = 0;
for(unsigned int i=0;i<dummy.count();i++)
{

if(dummy[i]->type()!=ROUTING) continue;
layers.insert(ind++, dummy[i]);
}
return layers;
}


#include <lefiMacro.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "clefgeom.h"

CLefGeom::CLefGeom()
{
}

CLefGeom::CLefGeom(lefiGeometries* lefiGeo)
{
lefr( lefiGeo );
}

CLefGeom::~CLefGeom()
{
while(!m_items.isEmpty() )
{
switch(m_items.front().first)
{
case GEOMRECT : delete (CRect*)m_items.front().second;break;
case GEOMLAYER : delete (QString*)m_items.front().second;break;
default : std::cout << "CLefGeom::~CLefGeom() not implemented
case\n";
}
m_items.pop_front();
}
m_items.clear();
}

void CLefGeom::addLayer(const QString layer)
{
QValueListIterator< QPair<geomType, void*> > itemIt;
bool exist = false;
for(itemIt=m_items.begin();itemIt!=m_items.end() && !exist;itemIt++)
    if( (*itemIt).first==GEOMLAYER && *(QString*)(*itemIt).second ==
layer ) exist=true;
if( !exist ) m_items.push_back( QPair<geomType, void*>::QPair(
GEOMLAYER, new QString(layer) ) );
}

void CLefGeom::addRect(const QString layer, const CRect rect)
{
QValueListIterator< QPair<geomType, void*> > itemIt;
bool exist = false;
for(itemIt=m_items.begin();itemIt!=m_items.end() && !exist;itemIt++)
    if( (*itemIt).first==GEOMLAYER && *(QString*)(*itemIt).second ==
layer ) exist=true;
if( !exist )
{
m_items.push_back( QPair<geomType, void*>::QPair( GEOMLAYER, new
QString(layer) ) );
m_items.push_back( QPair<geomType, void*>::QPair( GEOMRECT, new
CRect(rect) ) );
}else  m_items.insert( itemIt, QPair<geomType, void*>::QPair( GEOMRECT,
new CRect(rect) ) );
}

void CLefGeom::lefr(lefiGeometries* lefiGeo)
{
geomType type;
lefiGeomRect* rect;
QPair<geomType, void*> item;

for(int i=0;i<lefiGeo->numItems();i++)
{
type = ::geomType( lefiGeo->itemType(i) );
switch(type)
{
case GEOMRECT :
         rect = lefiGeo->getRect(i);
item.first = type;
item.second = (void*)new CRect(rect->xl,
rect->yl, rect->xh-rect->xl, rect->yh-rect->yl);
m_items.push_back( item );
break;
case GEOMLAYER :
item.first = type;
item.second = (void*)new
QString(lefiGeo->getLayer(i));
m_items.push_back( item );
break;
default :
std::cout << "CLefGeom::lefr() not implemented
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
case\n";
}
}
}

#include <lefiLayer.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "cleflayer.h"

CLefLayer::CLefLayer(const QString name, const int index)
:m_index(index)
,m_name(name)
,m_hasType(false)
,m_hasDirection(false)
,m_hasPitch(false)
,m_hasOffset(false)
,m_hasWidth(false)
,m_hasResistance(false)
,m_hasCapacitance(false)
,m_hasEdgeCap(false)
{
}

CLefLayer::CLefLayer(lefiLayer* layer, const int index)
:m_index(index)
,m_hasType(false)
,m_hasDirection(false)
,m_hasPitch(false)
,m_hasOffset(false)
,m_hasWidth(false)
,m_hasResistance(false)
,m_hasCapacitance(false)
,m_hasEdgeCap(false)
{
lefr(layer);
}

CLefLayer::~CLefLayer()
{
}

void CLefLayer::lefr(lefiLayer* layer)
{
setName( layer->name() );
if (layer->hasType()) setType( ::layer( layer->type()) );
if (layer->hasPitch()) setPitch( layer->pitch() );
if (layer->hasOffset()) setOffset( layer->offset() );
if (layer->hasWidth()) setWidth( layer->width() );
if (layer->hasDirection()) setDirection(
::layerDirection(layer->direction()) );
if (layer->hasResistance()) setResistance( layer->resistance() );
if (layer->hasCapacitance()) setCapacitance(layer->capacitance() );
if (layer->hasEdgeCap()) setEdgeCap(layer->edgeCap() );
}

void CLefLayer::lefw() const
{
if( type()==CUT ) lefwCut();
else if( type()==MASTERSLICE || type()==OVERLAP ) lefwMasOvr();
else if( type()==ROUTING ) lefwRout();
}

void CLefLayer::lefwCut() const
{
lefwStartLayer(name(), ::layer(type()));
lefwEndLayer(name());
}

void CLefLayer::lefwMasOvr() const
{
lefwStartLayer( name(), ::layer(type()) );
lefwEndLayer( name() );
}

void CLefLayer::lefwRout() const
{
QString s;

lefwStartLayerRouting(name());
if (hasWidth() && hasDirection()) lefwLayerRouting(
::layerDirection(direction()), width());
if (hasPitch()) lefwLayerRoutingPitch( pitch() );
if (hasOffset()) lefwLayerRoutingOffset( offset() );
if (hasResistance()) lefwLayerRoutingResistance( s.setNum( resistance())
);
if (hasCapacitance()) lefwLayerRoutingCapacitance( s.setNum(
capacitance()) );
if (hasEdgeCap()) lefwLayerRoutingEdgeCap( edgeCap() );
lefwEndLayerRouting(name());
}

double CLefLayer::unitRes() const
{
if(hasResistance() && hasWidth()) return resistance()/width();
else return 0;
}

double CLefLayer::unitCap() const
{
double c = (hasCapacitance())? capacitance() : 0;
double w = (hasWidth())? width() : 0;
double e = (hasEdgeCap())? edgeCap() : 0;
return c * w + 2 * e;
}
#include <lefiMacro.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "clefmacro.h"
#include "clefobs.h"
#include "clefpin.h"

CLefMacro::CLefMacro(const QString name)
:m_name(name)
,m_hasSize(false)
,m_hasOrigin(false)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
m_pins.setAutoDelete( true );
m_obs.setAutoDelete( true );
}

CLefMacro::~CLefMacro()
{
m_pins.clear();
m_obs.clear();
}

void CLefMacro::lefr(lefiMacro* macro)
{
setName( macro->name() );
if(macro->hasSize() ) setSize( macro->sizeX(), macro->sizeY() );
if(macro->hasOrigin() ) setOrigin( macro->originX(), macro->originY() );
}

void CLefMacro::lefw() const
{
QDictIterator<CLefPin> pinIt(m_pins);
QPtrListIterator<CLefObs> obsIt(m_obs);

lefwStartMacro(name());
if( hasSize() ) lefwMacroSize(sizeX(),sizeY());
if( hasOrigin() ) lefwMacroOrigin(originX(),originY());
for(pinIt.toFirst();pinIt.current();++pinIt) (*pinIt)->lefw();
for(obsIt.toFirst();obsIt.current();++obsIt) (*obsIt)->lefw();
lefwEndMacro(name());
}

void CLefMacro::addPin(CLefPin* pin)
{
m_pins.insert( pin->name(), pin );
}

void CLefMacro::addObs(CLefObs* obs)
{
m_obs.append( obs );
}


#include <lefiMacro.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "clefobs.h"

CLefObs::CLefObs()
:CLefGeom()
{
}

CLefObs::CLefObs(lefiGeometries* lefiGeo)
:CLefGeom( lefiGeo )
{
}

CLefObs::~CLefObs()
{
}

void CLefObs::lefw() const
{
CRect rect;
QString layer;
QValueListConstIterator< QPair<geomType, void*> > geomIt;

lefwStartMacroObs();
for(geomIt=m_items.begin();geomIt!=m_items.end();geomIt++)
{
switch((*geomIt).first)
{
case GEOMRECT:
rect = *(CRect*)(*geomIt).second;
lefwMacroObsLayerRect(
rect.left(),rect.bottom(),rect.right(),rect.top(),0,0,0,0);
break;
case GEOMLAYER:
layer = *(QString*)(*geomIt).second;
lefwMacroObsLayer(layer,0);
break;
default: std::cout << "CLefObs::lefw() not implemented case\n";
}
}
lefwEndMacroObs();
}

#include <lefiMacro.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "clefpin.h"
#include "clefport.h"

CLefPin::CLefPin(const QString name)
:m_name(name)
,m_hasDirection(false)
,m_hasUse(false)
{
m_ports.setAutoDelete(true);
}

CLefPin::CLefPin(lefiPin* pin)
:m_hasDirection(false)
,m_hasUse(false)
{
m_ports.setAutoDelete(true);
lefr(pin);
}

CLefPin::~CLefPin()
{
m_ports.clear();
}

void CLefPin::lefr(lefiPin* pin)
{
setName( pin->name() );
if( pin->hasDirection() ) setDirection( ::pinDirection(pin->direction())
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
);
if( pin->hasUse() ) setUse( ::use( pin->use() ) );
for(int  i=0;i<pin->numPorts();i++) addPort( new CLefPort( pin->port(i)
) );
}

void CLefPin::lefw()
{
QPtrListIterator<CLefPort> portIt(m_ports);

lefwStartMacroPin(name());
if( hasDirection() ) lefwMacroPinDirection(::pinDirection(direction()));
if( hasUse() ) lefwMacroPinUse( ::use(use()) );
for( portIt.toFirst();portIt.current();++portIt) (*portIt)->lefw();
lefwEndMacroPin(name());
}
#include <lefiMacro.hpp>
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include "clefport.h"

CLefPort::CLefPort()
:CLefGeom()
{
}

CLefPort::CLefPort(lefiGeometries* lefiGeo)
:CLefGeom(lefiGeo)
{
}

CLefPort::~CLefPort()
{

}

void CLefPort::lefw()  const
{
QValueListConstIterator< QPair<geomType, void*> > geomIt;
CRect rect;
QString layer;

lefwStartMacroPinPort(0);
for(geomIt=m_items.begin();geomIt!=m_items.end();geomIt++)
{
switch((*geomIt).first)
{
case GEOMRECT:
rect = *(CRect*)(*geomIt).second;

lefwMacroPinPortLayerRect(rect.left(),rect.bottom(),rect.right(),rect.top(),0,0,
0,0);
break;
case GEOMLAYER:
layer = *(QString*)(*geomIt).second;
lefwMacroPinPortLayer(layer,0);
break;
default :
std::cout << "CLefPort::lefw() no implemented case\n";
```

```
}
}
lefwEndMacroPinPort();
}
#include "clef.h"
#include "cleflayer.h"
#include "clefmacro.h"
#include "clefobs.h"
#include "clefpin.h"
#include "clefvia.h"

int lefrAntennaInoutCbk(lefrCallbackType_e type, double inout, lefiUserData
data)
{
assert(type==lefrAntennaInoutCbkType);
std::cout << "lefrAntennaInoutCbk " << inout << data << "\n";
return 0;
}

int lefrAntennaInputCbk(lefrCallbackType_e type, double input, lefiUserData
data)
{
assert(type==lefrAntennaInputCbkType);
std::cout << "lefrAntennaInputCbk " << input << data << "\n";
return 0;
}

int lefrAntennaOutputCbk(lefrCallbackType_e type, double output, lefiUserData
data)
{
assert(type==lefrAntennaOutputCbkType);
std::cout << "lefrAntennaOutputCbk " <<  output << data << "\n";
return 0;
}

int lefrBusBitCharsCbk(lefrCallbackType_e type, const char* busBits,
lefiUserData data)
{
assert(type==lefrBusBitCharsCbkType);
CLef* lef = (CLef*)data;

lef->busBitChars( busBits );
return 0;
}

int lefrClearanceMeasureCbk(lefrCallbackType_e type, const char* clearance,
lefiUserData data)
{
assert(type==lefrClearanceMeasureCbkType);
std::cout << "lefrClearanceMeasureCbk " <<  clearance << data << "\n";
return 0;
}

int lefrDividerCharCbk(lefrCallbackType_e type, const char* divider,
lefiUserData data)
{
assert(type==lefrDividerCharCbkType);
CLef* lef = (CLef*)data;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
lef->dividerChar( divider );
return 0;
}

int lefrLibraryEndCbk(lefrCallbackType_e type, void* ptr, lefiUserData data)
{
assert(type==lefrLibraryEndCbkType);
std::cout << "lefrLibraryEndCbk " << ptr << data << "\n";
return 0;
}

int lefrLayerCbk(lefrCallbackType_e type, lefiLayer* layer, lefiUserData data)
{
assert(type==lefrLayerCbkType);
CLef* lef = (CLef*)data;

lef->m_layers.insert( layer->name(), new CLefLayer( layer,
lef->m_layerIndex ) );
(lef->m_layerIndex)++;
   return 0;
}

int lefrMacroBeginCbk(lefrCallbackType_e type, const char* macro, lefiUserData
data)
{
assert(type==lefrMacroBeginCbkType);
CLef* lef = (CLef*)data;

lef->m_macros.insert( macro, new CLefMacro( macro ) );
lef->m_currMacro = macro;
return 0;
}

int lefrMacroCbk(lefrCallbackType_e type, lefiMacro* macro, lefiUserData data)
{
assert(type==lefrMacroCbkType);
CLef* lef = (CLef*)data;

lef->m_macros[lef->m_currMacro]->lefr( macro );
return 0;
}

int lefrMacroClassTypeCbk(lefrCallbackType_e type, const char* macroClass,
lefiUserData data)
{
assert(type==lefrMacroClassTypeCbkType);
std::cout << "lefrMacroClassTypeCbk " << macroClass << data << "\n";
return 0;
}

int lefrObstructionCbk(lefrCallbackType_e type, lefiObstruction* pObs,
lefiUserData data)
{
assert(type==lefrObstructionCbkType);
CLef* lef = (CLef*)data;

lef->m_macros[lef->m_currMacro]->addObs( new CLefObs(pObs->geometries())
);
return 0;
}

int lefrPinCbk(lefrCallbackType_e type, lefiPin* pin, lefiUserData data)
{
assert(type==lefrPinCbkType);
CLef* lef = (CLef*)data;

lef->m_macros[lef->m_currMacro]->addPin( new CLefPin( pin ) );
return 0;
}

int lefrManufacturingCbk(lefrCallbackType_e type,double grid,lefiUserData data)
{
assert(type==lefrManufacturingCbkType);
std::cout << "lefrManufacturingCbk " << grid << data << "\n";
return 0;
}

int lefrCaseSensitiveCbk(lefrCallbackType_e type,int caseSensitive,lefiUserData
data)
{
assert(type==lefrCaseSensitiveCbkType);
std::cout << "lefrCaseSensitiveCbk " << caseSensitive << data << "\n";
CLef* lef = (CLef*)data;

lef->caseSensitive( caseSensitive==1 );
return 0;
}

int lefrNoWireExtensionCbk(lefrCallbackType_e type,const char*
wireExtension,lefiUserData data)
{
assert(type==lefrNoWireExtensionCbkType);
std::cout << "lefrNoWireExtensionCbk " << wireExtension << data << "\n";
return 0;
}

int lefrNonDefaultCbk(lefrCallbackType_e type, lefiNonDefault* def, lefiUserData
data)
{
assert(type==lefrNonDefaultCbkType);
std::cout << "lefrNonDefaultCbk " << def << data << "\n";
return 0;
}

int lefrPropBeginCbk(lefrCallbackType_e type,void* ptr,lefiUserData data)
{
assert(type==lefrPropBeginCbkType);
std::cout << "lefrPropBeginCbk " << ptr << data << "\n";
return 0;
}

int lefrPropCbk(lefrCallbackType_e type,lefiProp* prop,lefiUserData data)
{
assert(type==lefrPropCbkType);
std::cout << "lefrPropCbk " << prop << data << "\n";
return 0;
```

```
}

int lefrPropEndCbk(lefrCallbackType_e type,void* ptr,lefiUserData data)
{
assert(type==lefrPropEndCbkType);
std::cout << "lefrPropEndCbk " << ptr << data << "\n";
return 0;
}

int lefrSpacingBeginCbk(lefrCallbackType_e type,void* ptr,lefiUserData data)
{
assert(type==lefrSpacingBeginCbkType);
std::cout << "lefrSpacingBeginCbk " << ptr << data << "\n";
return 0;
}

int lefrSpacingCbk(lefrCallbackType_e type,lefiSpacing* spacing, lefiUserData
data)
{
assert(type==lefrSpacingCbkType);
std::cout << "lefrSpacingCbk " << spacing << data << "\n";
return 0;
}

int lefrSpacingEndCbk(lefrCallbackType_e type,void* ptr,lefiUserData data)
{
assert(type==lefrSpacingEndCbkType);
std::cout << "lefrSpacingEndCbk " << ptr << data << "\n";
return 0;
}

int lefrSiteCbk(lefrCallbackType_e type, lefiSite* site, lefiUserData data)
{
assert(type==lefrSiteCbkType);
std::cout << "lefrSiteCbk " << site << data << "\n";
return 0;
}

int lefrUnitsCbk(lefrCallbackType_e type, lefiUnits* units, lefiUserData data)
{
assert(type==lefrUnitsCbkType);
std::cout << "lefrUnitsCbk " << units << data << "\n";
return 0;
}

int lefrUseMinSpacingCbk(lefrCallbackType_e type, lefiUseMinSpacing* minspacing,
lefiUserData data)
{
assert(type==lefrUseMinSpacingCbkType);
std::cout << "lefrUseMinSpacingCbk " << minspacing << data << "\n";
return 0;
}

int lefrVersionCbk(lefrCallbackType_e type, double version, lefiUserData data)
{
assert(type==lefrVersionCbkType);
CLef* lef = (CLef*)data;
lef->version( version );
return 0;
}

int lefrVersionStrCbk(lefrCallbackType_e type,const char* version,lefiUserData
data)
{
assert(type==lefrVersionStrCbkType);
CLef* lef = (CLef*)data;

lef->versionStr( version );
return 0;
}

int lefrViaCbk(lefrCallbackType_e type, lefiVia* via, lefiUserData data)
{
assert(type==lefrViaCbkType);
CLef* lef = (CLef*)data;

lef->m_vias.insert(via->name(),new CLefVia(via));
return 0;
}

int lefrViaRuleCbk(lefrCallbackType_e type, lefiViaRule* viarule, lefiUserData
data)
{
assert(type==lefrViaRuleCbkType);
std::cout << "lefrViaRuleCbk " << viarule << data << "\n";
return 0;
}
#include "clefvia.h"
#include <lefwWriter.hpp>
#include <lefwWriterCalls.hpp>
#include <lefiVia.hpp>

CLefVia::CLefVia(lefiVia* lefiVia)
{
lefr(lefiVia);
}
CLefVia::~CLefVia()
{
m_data.clear();
}

void CLefVia::lefr(lefiVia* lefiVia)
{
m_name = lefiVia->name();
for(int i=0;i<lefiVia->numLayers();i++)
{
QString layerName = lefiVia->layerName(i);
QValueList<CRect> rects;
for(int j=0;j<lefiVia->numRects(i);j++)
{
CRect rect;
rect.left( lefiVia->xl(i,j) );
rect.bottom( lefiVia->yl(i,j) );
rect.right( lefiVia->xh(i,j) );
rect.top( lefiVia->yh(i,j) );
rects.append( rect );
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
}
m_data.append( QPair<QString, QValueList<CRect>
>::QPair(layerName,rects) );
}
}

void CLefVia::lefw()
{
QValueListIterator< QPair<QString, QValueList<CRect> > > layerIt;
QValueListIterator<CRect> rectIt;
lefwStartVia(m_name,"");
for(layerIt=m_data.begin();layerIt!=m_data.end();layerIt++)
{
lefwViaLayer( (*layerIt).first );

for(rectIt=(*layerIt).second.begin();rectIt!=(*layerIt).second.end();rectIt++)
{

lefwViaLayerRect((*rectIt).left(),(*rectIt).bottom(),(*rectIt).right(),
(*rectIt).top());
}
}
lefwEndVia(m_name);
}
#include "clef.h"
#include "cleflayer.h"
#include "clefmacro.h"
#include "clefvia.h"

int lefwAntennaCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwAntennaCbkType);
std::cout << "lefwAntennaCbk " << data << "\n";
return 0;
}

int lefwBusBitCharsCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwBusBitCharsCbkType);
CLef* lef = (CLef*)data;

if( lef->hasBusBitChars() ) lefwBusBitChars( lef->m_busBitChars );
return 0;
}

int lefwClearanceMeasureCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwClearanceMeasureCbkType);
std::cout << "lefwClearanceMeasureCbk " << data << "\n";
return 0;
}

int lefwDividerCharCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwDividerCharCbkType);
CLef* lef = (CLef*)data;

if( lef->hasDividerChar() ) lefwDividerChar( lef->m_dividerChar );
return 0;
}
int lefwExtCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwExtCbkType);
std::cout << "lefwExtCbk " << data << "\n";
return 0;
}

int lefwEndLibCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwEndLibCbkType);
std::cout << "lefwEndLibCbk " << data << "\n";
lefwEnd();
return 0;
}

int lefwLayerCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwLayerCbkType);
CLef* lef = (CLef*)data;

QDictIterator<CLefLayer> layerIt(lef->m_layers);
for(unsigned int i=0;i<layerIt.count();i++)
for(layerIt.toFirst();layerIt.current();++layerIt)
if( (*layerIt)->index()==i ) {(*layerIt)->lefw();break;}
return 0;
}
int lefwMacroCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwMacroCbkType);
CLef* lef = (CLef*)data;
QDictIterator<CLefMacro> macroIt(lef->m_macros);

for(macroIt.toFirst();macroIt.current();++macroIt) (*macroIt)->lefw();
return 0;
}
int lefwManufacturingGridCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwManufacturingGridCbkType);
std::cout << "lefwManufacturingGridCbk " << data << "\n";
return 0;
}
int lefwCaseSensitiveCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwCaseSensitiveCbkType);
CLef* lef = (CLef*)data;

if( lef->hasCaseSensitive() )
{
if( lef->m_caseSensitive ) lefwCaseSensitive( "ON" );
else lefwCaseSensitive( "OFF" );
}
return 0;
}
int lefwNonDefaultCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwNonDefaultCbkType);
std::cout << "lefwNonDefaultCbk " << data << "\n";
return 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
}
int lefwNoWireExtensionCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwNoWireExtensionCbkType);
std::cout << "lefwNoWireExtensionCbk " << data << "\n";
return 0;
}
int lefwPropDefCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwPropDefCbkType);
std::cout << "lefwPropDefCbk " << data << "\n";
return 0;
}
int lefwSiteCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwSiteCbkType);
std::cout << "lefwSiteCbk " << data << "\n";
return 0;
}
int lefwSpacingCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwSpacingCbkType);
std::cout << "lefwSpacingCbk " << data << "\n";
return 0;
}
int lefwUnitsCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwUnitsCbkType);
std::cout << "lefwUnitsCbk " << data << "\n";
return 0;
}
int lefwUseMinSpacingCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwUseMinSpacingCbkType);
std::cout << "lefwUseMinSpacingCbk " << data << "\n";
return 0;
}
int lefwVersionCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwVersionCbkType);
CLef* lef = (CLef*)data;

if( lef->hasVersion() )
{
int ver1 = (int)floor(lef->m_versionStr.toDouble());
int ver2 =
(int)(10.0*(lef->m_versionStr.toDouble()-(double)ver1));
lefwVersion( ver1, ver2 );
}
return 0;
}
int lefwViaCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwViaCbkType);
CLef* lef = (CLef*)data;

QDictIterator<CLefVia> viaIt(lef->m_vias);
for(viaIt.toFirst();viaIt.current();++viaIt) (*viaIt)->lefw();
return 0;
}
int lefwViaRuleCbk(lefwCallbackType_e type, lefiUserData data)
{
assert(type==lefwViaRuleCbkType);
std::cout << "lefwViaRuleCbk " << data << "\n";
return 0;
}
#include "clib.h"
#include "ccctimage.h"
#include "ccctpin.h"
#include "ccctkeepout.h"

CLib::CLib()
{
m_images.setAutoDelete(true);
}
CLib::~CLib()
{
m_images.clear();
}

bool CLib::load(const QString fileName)
{

QFile libFile(fileName);
if( !libFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open " << fileName << "\n";
return false;
}
QTextStream lib(&libFile);

QString var,val,val1,val2;
QString imageName;
QString pinName,layer;
double x1,x2,y1,y2;
CPoly poly;
CRect macroRect;
CCctImage* pImage = NULL;
CCctPin* pPin;

while( !lib.atEnd() )
{
lib >> var;
var = var.replace( QRegExp("\\(|\\)"),"" );
if( var=="image" )
{
lib >> imageName;
pImage = new CCctImage(imageName);
m_images.insert( pImage->name(), pImage );
}
else if( var=="outline" )
{
lib >> val;
val = val.replace( QRegExp("\\(|\\)"),"");
if ( val=="polygon" )
{
poly.clear();
lib >> val >> val;
```

```
lib >> val1 >> val2;
do
{
poly.append( val1.toDouble(),
val2.toDouble() );
lib >> val1 >> val2;
}while( val1!=")" && val2!=")" );
pImage->setPoly( poly );
}
else if( val=="rect" )
{
lib >> val >> x1 >> y1 >> x2 >> y2;
pImage->setRect( CRect(x1,y1,x2-x1,y2-y1) );
}
}
else if( var=="spin" )
{
lib >> pinName >> val >> val  >> layer >> x1 >> y1 >> x2
>> y2;
layer = layer.upper();
if( pImage->pin( pinName )!=NULL )
{
pPin = pImage->pin( pinName );
}
else
{
pPin = new CCctPin( pinName );
pImage->addPin( pPin );
}
pPin->addPort(layer,CRect(x1,y1,x2-x1,y2-y1));
}
else if(var=="keepout" )
{
lib >> val >> layer >> x1 >> y1 >> x2 >> y2;
layer = layer.upper();
pImage->addKeepout( new CCctKeepout(layer,
CRect(x1,y1,x2-x1,y2-y1)));
}
}
libFile.close();

return true;
}

bool CLib::save(const QString fileName)
{
std::cout << "CLib::save() not implemented yet " << fileName << "\n";
return true;
}

#include "cmaze.h"
#include "cmnode.h"
#include "clayer.h"
#include "openset.h"
#include "gwire.h"
#include "gpath.h"

CMaze::CMaze(CTile from, CTile to, QPtrVector<CLayer> layers, gwire* wire)
:m_wire(wire)
,m_from(from)
,m_to(to)
,m_layers(layers)
,m_layerChangeCost(1)
{
// std::cout << "maze " << from.layer()->name() << " " << from.x() << " "<<
from.y();
// std::cout << " -> " << to.layer()->name() << " " <<  to.x() << " " <<
to.y() << "\n";
}

CMaze::~CMaze()
{
}

bool CMaze::route()
{
//static int s=0;
//static int f=0;
CMTile *node = new CMTile(m_from);
node->costFromStart(0);
node->costToGoal(costEstimate(node));

QPtrList<CMTile> back;
CSortedPtrList front;

front.inSort(node);
while(!front.isEmpty())
{
node = front.take(0);
//std::cout << c << " node " << node->layer()->name() << " " << node->x() << " "
<< node->y();
//std::cout << " \t: " << node->costFromStart()+node->costToGoal() << "\n";
if(isLast(node))
{

// std::cout << s++ << "\tSuccess.\n";
m_wire->decCellFlows();
m_wire->clear();
m_wire->append( new
gpath(CLayerPair(node->layer(),NULL),node->rect(),node->rect(),true) );
assert(node->parent()!=NULL);
CMTile* to = node;
CMTile* from = node;
do
{
if(node->layer()==to->layer()) from =
node;
else
{
if( from->x()!=to->x() ||
from->y()!=to->y() )
m_wire->prepend( new
gpath(CLayerPair(from->layer(),NULL),from->rect(),to->rect(),true) );
to = node;
from = node;
}
node = node->parent();
}while(node!=NULL);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if(from->x()!=to->x() || from->y()!=to->y())
m_wire->prepend( new
gpath(CLayerPair(from->layer(),NULL),from->rect(),to->rect(),true) );
m_wire->prepend( new
gpath(CLayerPair(from->layer(),NULL),from->rect(),from->rect(),true) );
m_wire->incCellFlows();

front.setAutoDelete(true);
back.setAutoDelete(true);
front.clear();
back.clear();
return true;
}
QPtrList<CMTile> succ = this->succ(node);
succ.first();
while(succ.current())
{
CMTile* newNode = succ.current();
int cfs = node->costFromStart();
  int tcost = travelCost(node, newNode);
int newCost = cfs + tcost;
bool insert = true;
//std::cout << "\tsucc " << newNode->layer()->name() << " " << newNode->x() << "
" << newNode->y();
//std::cout << " \t: " << newCost;
CMTile* oldNode = find(back,newNode);
if(oldNode!=NULL)
{
if(newCost < oldNode->costFromStart() )
{
back.remove(oldNode);
delete oldNode;
insert = true;
}else insert = false;
}
oldNode = find(front,newNode);
if(oldNode!=NULL)
{
if(newCost < oldNode->costFromStart() )
{
front.remove(oldNode);
// delete oldNode;
insert = true;
}else insert = false;
}
if(insert)
{
//std::cout << " inserted \t";
newNode->parent(node);
newNode->costFromStart( newCost );
newNode->costToGoal(costEstimate(newNode));
//std::cout << newNode->costFromStart()+newNode->costToGoal() << "\n";
front.inSort(newNode);
}else { delete newNode;/*std::cout << "\n";*/}
succ.next();
}
back.append(node); //done with this node
}
```

```
//std::cout << f++ << "\tNo Path!!\n";
//no path found
return false;
}

int CMaze::costEstimate(CMTile* from)
{
int dist = abs(from->x()-m_to.x());
dist += abs(from->y()-m_to.y());
dist += abs( (from->layer()->index()-m_to.layer()->index()) *
m_layerChangeCost );
return dist;
}

int CMaze::travelCost(CMTile *fromNode, CMTile *toNode)
{
int retVal = 1;

// add some if we are changing layer
if(fromNode->layer() != toNode->layer())
      retVal += m_layerChangeCost;
return retVal;
}

bool CMaze::isLast(CMTile* tile)
{
return  (tile->x()==m_to.x()) &&
(tile->y()==m_to.y()) &&
(tile->layer()==m_to.layer());
}

CMTile* CMaze::find(QPtrList<CMTile> set,CMTile* node)
{
set.first();
while(set.current())
{
CMTile* curr = set.current();
if( curr->x()==node->x() &&
curr->y()==node->y() &&
curr->layer()==node->layer() ) return curr;
set.next();
}
return NULL;
}

QPtrList<CMTile> CMaze::succ(CMTile* node)
{
QPtrList<CMTile> set;
CLayer* layer = node->layer();
int x = node->x();
int y = node->y();
int minX = ::min(m_from.x(),m_to.x())-abs(m_from.x()-m_to.x());
int maxX = ::max(m_from.x(),m_to.x())+abs(m_from.x()-m_to.x());
int minY = ::min(m_from.y(),m_to.y())-abs(m_from.y()-m_to.y());
int maxY = ::max(m_from.y(),m_to.y())+abs(m_from.y()-m_to.y());
if(maxX-minX<=4) {minX -= 2;maxX += 2;}
if(maxY-minY<=4) {minY -= 2;maxY += 2;}
// Adjacent tiles on the X axis
if(node->layer()->dir()==HORIZONTAL)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
if(!layer->cellFull(x,y) && x-1>=minX) set.append( new
CMTile(layer,x-1,y) );
if(!layer->cellFull(x+1,y) && x+1<=maxX) set.append( new
CMTile(layer,x+1,y) );
}else if(node->layer()->dir()==VERTICAL)
{
if(!layer->cellFull(y,x) && y-1>=minY) set.append( new
CMTile(layer,x,y-1) );
if(!layer->cellFull(y+1,x) && y+1<=maxY) set.append( new
CMTile(layer,x,y+1) );
}else assert(false);

  CLayer* nl = layer->next();
if( nl!=NULL && nl->selected() )
{
if(nl->dir()==HORIZONTAL)
{
if(!nl->cellFull(x,y) || !nl->cellFull(x+1,y) )
set.append( new CMTile(nl,x,y) );
}else if(nl->dir()==VERTICAL )
{
if(!nl->cellFull(y,x) || !nl->cellFull(y+1,x) )
set.append( new CMTile(nl,x,y) );
}else assert(false);
}

CLayer* pl = layer->prev();
if( pl!=NULL && pl->selected() )
{
if(pl->dir()==HORIZONTAL)
{
if(!pl->cellFull(x,y) || !pl->cellFull(x+1,y) )
set.append( new CMTile(pl,x,y) );
}else if(pl->dir()==VERTICAL )
{
if(!pl->cellFull(y,x) || !pl->cellFull(y+1,x) )
set.append( new CMTile(pl,x,y) );
}else assert(false);
}
   return set;
}

/***************************************************************************
                          cmnode.cpp  -  description
                             -------------------
    begin                : Tue Jan 20 2004
    copyright            : (C) 2004 by
    email                :
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include "cmnode.h"

CMNode::CMNode(){
}
CMNode::~CMNode(){
}
/***************************************************************************
                          cmtile.cpp  -  description
                             -------------------
    begin                : Tue Jan 20 2004
    copyright            : (C) 2004 by
    email                :
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include "cmtile.h"
#include "global.h"

CMTile::CMTile()
{
}

CMTile::CMTile(const CMTile& tile)
:CTile(tile)
{
}

CMTile::CMTile(CTile& tile,CMTile* parent)
:CTile(tile)
,m_parent(parent)
{

}

CMTile::CMTile(CLayer* layer, int x, int y)
:CTile(layer,x,y,QSize(1,1))
,m_parent(NULL)
{

}

CMTile::~CMTile()
{
}

CSortedPtrList::CSortedPtrList()
:QPtrList<CMTile>::QPtrList()
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}

CSortedPtrList::~CSortedPtrList()
{
}

int CSortedPtrList::compareItems(QPtrCollection::Item item1,
QPtrCollection::Item item2)
{
int c1 = ((CMTile*)item1)->costFromStart() +
((CMTile*)item1)->costToGoal();
int c2 = ((CMTile*)item2)->costFromStart() +
((CMTile*)item2)->costToGoal();
if(c1==c2) return 0;
if(c1 > c2 ) return +1;
if(c1 < c2 ) return -1;
assert(false);
}
```



```
#include "cnet.h"
#include "clayer.h"
#include "cnode.h"
#include "csource.h"
#include "cbuffer.h"
#include "csubroot.h"
#include "csink.h"
#include "gwire.h"
#include "ccctpath.h"
#include "cgroute.h"
#include "crte.h"

CNet::CNet(const QString name)
:QString(name)
,m_root(NULL)
{
    m_nodes.setAutoDelete(true);
setAutoDelete(true);
}
CNet::~CNet()
{
m_nodes.clear();
clear();
}

void CNet::root(CNode* root)
{
m_root = root;
CNode* driver = new CNode(DRIVER,*root);
connectNodes(driver,root);
m_nodes.append(root);
m_nodes.append(driver);
}

void CNet::makeRoot(const double res, const double cap)
{
CSink* sink = (CSink*)m_nodes.first();
CSource* src = new
CSource((QString)*sink,sink->layer(),(CPoint)*sink,res,cap);
m_nodes.remove(sink);
root(src);
}

void CNet::layer(const CLayerPair layer)
{
    QValueList<CEdge> edges = this->edges();
QValueListIterator<CEdge> eIt;

for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
if(m_layer.vl()!=NULL) m_layer.vl()->decnUsage( (*eIt).yDist()
);
layer.vl()->incnUsage( (*eIt).yDist() );
if(m_layer.hl()!=NULL) m_layer.hl()->decnUsage( (*eIt).xDist()
);
layer.hl()->incnUsage( (*eIt).xDist() );
(*eIt).down()->edge().layer( layer );
}
m_layer = layer;
}

CNode* CNet::pin(const QString pin) const
{
CSink* sink;
QPtrListIterator<CNode> it(m_nodes);
for(it.toFirst();it.current();++it)
{
CNode* n = *it;
if(n->type()==SINK)
{
sink = (CSink*)n;
if(sink->name()==pin) return n;
}
}
return NULL;
}

QValueList<CEdge> CNet::edges() const
{
QValueList<CEdge> edges;
QPtrListIterator<CNode> nIt(m_nodes);
for(nIt.toFirst();nIt.current();++nIt)
if( (*nIt)->parent()!=NULL ) edges.append( CEdge((*nIt)->edge())
);
return edges;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
double CNet::length() const
{
double l = 0;
QPtrListIterator<CNode> nIt(m_nodes);
for(nIt.toFirst();nIt.current();++nIt)
if( (*nIt)->parent()!=NULL ) l += (*nIt)->edge().manhDist();
return l;
}

void CNet::joinNodeAtEdge(CNode* node, const CEdge edge)
{
CNode* up = edge.up();
CNode* down = edge.down();


CPoint mid( ::median(up->x(),node->x(),down->x()),

::median(up->y(),node->y(),down->y()));

  nodeType tpd = down->type();
  //nodeType tbu = up->type();

if (tpd == ROOT || tpd == SUBROOT)
connectNodes(down, node);
//if edge is buffer or driver, connect directly to the output of
the gate
else
    joinNodeAtPoint(node, edge, mid);
//steiner point at hannan grid, median of the points

}

void CNet::joinNodeAtPoint(CNode* node, const CEdge edge, const CPoint p)
{
CNode* down = edge.down();
CNode* up = edge.up();

    if (*up==p)
{  //close to  upper node, connect directly to upper node
    connectNodes(up, node);
    }
else if(*down==p )
{ //close to donw node, connect directly down node
    connectNodes(down, node);
    }
    else
{
    edge.disconnect();
     if(*node==p)
{//steiner close to node, connect upper to node and node to down
connectNodes(up, node);
connectNodes(node, down);
    }
else
{//create the steiner
CNode* steiner = new CNode(STEINER,p);
m_nodes.append(steiner);
connectNodes(up, steiner);
connectNodes(steiner, down);
connectNodes(steiner, node);
    }
    }
}

void CNet::disjoinNodeFromEdge(CNode* node, const CEdge edge)
{
CNode* steiner = NULL;
if( node->parent()!=edge.up() && node->parent()!=edge.down())
{  //there is steiner
steiner = node->parent();
steiner->edge().disconnect();
}
node->edge().disconnect();
edge.down()->edge().disconnect();
edge.connect();
if(steiner!=NULL) m_nodes.remove(steiner);
}

void CNet::connectNodes(CNode* up, CNode* down)
{
// std::cout << "connectNodes1\n";
up->addChild( down );
// std::cout << "connectNodes2\n";
down->parent( up );
// std::cout << "connectNodes3\n";
down->edge( CEdge(up,down, m_layer) );
// std::cout << "connectNodes4\n";
if(down->type()!=ROOT &&  down->type()!=SUBROOT)
{
// std::cout << "connectNodes5\n";
// std::cout << m_layer.hl()->name() << "\n";
m_layer.vl()->incnUsage( down->edge().yDist() );
// std::cout << "connectNodes6\n";
m_layer.hl()->incnUsage( down->edge().xDist() );
}
// std::cout << "connectNodes7\n";
}


bool CNet::insertBufferAtEdge(const CEdge edge,  const double load, const double
res, const double cap)
{
//inser buffer at edge
// if down node is steiner, insert buffer at steiner pioint
// otherwise inser buffer in the middle
//inp : pEdge edge to insert buffer
//inp : buffer paramaters
CNode* down = edge.down();
CNode* up = edge.up();
CBuffer *buffer = NULL;
CSubRoot *subroot = NULL;
CPoint mid;

if( down->type() ==ROOT || down->type() == SUBROOT ) return false;
// if( down->type() ==SINK || down->type()==BUFFER)
mid = CPoint( (up->x()+down->x())/2.0, (up->y()+down->y())/2.0
);
// else mid=*down;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
buffer =  new CBuffer(mid,load);
subroot = new CSubRoot(mid,res,cap);
m_nodes.append( buffer );
m_nodes.append( subroot );
edge.disconnect();
connectNodes(up,buffer);
connectNodes(buffer,subroot);
connectNodes(subroot,down);
   return true;
}

bool CNet::insertBufferAtNode(const CEdge edge,  const double load, const double
res, const double cap)
{
//inser buffer at edge
// if down node is steiner, insert buffer at steiner pioint
// otherwise inser buffer in the middle
//inp : pEdge edge to insert buffer
//inp : buffer paramaters
CNode* down = edge.down();
CNode* up = edge.up();
CBuffer *buffer = NULL;
CSubRoot *subroot = NULL;
CPoint mid;

if( down->type() ==ROOT || down->type() == SUBROOT ) return false;
if( down->type() ==SINK || down->type()==BUFFER) return false;
    if( down->type()!=STEINER ) return false;
    mid=*down;

buffer =  new CBuffer(mid,load);
subroot = new CSubRoot(mid,res,cap);
m_nodes.append( buffer );
m_nodes.append( subroot );
edge.disconnect();
connectNodes(up,buffer);
connectNodes(buffer,subroot);
connectNodes(subroot,down);
   return true;
}

//removes buffer from edge
//inp : edge has the buffer to remove,
void CNet::removeBufferFromEdge(CEdge edge)
{
CNode* subroot = edge.down()->parent();
CNode* buffer = subroot->parent();

edge.down()->edge().disconnect();
subroot->edge().disconnect();
buffer->edge().disconnect();
m_nodes.remove(subroot);
m_nodes.remove(buffer);

edge.connect();
}

void CNet::removeBufferFromNode(CEdge edge)
{
CNode* subroot = edge.down()->parent();
CNode* buffer = subroot->parent();

edge.down()->edge().disconnect();
subroot->edge().disconnect();
buffer->edge().disconnect();
m_nodes.remove(subroot);
m_nodes.remove(buffer);

edge.connect();
}

void CNet::removeBuffer(CNode* n, bool perm)
{
  if(perm)
  {
      CNode* d = n;
      CNode* u = n->parent();
      d->edge().disconnect();
      QPtrList<CNode> children = d->children();
      while(!children.isEmpty())
      {
          CNode* child = children.take(0);
child->edge().disconnect();
connectNodes(u,child);
       }
       m_nodes.remove( d );

       if(u->children().count()==1)
       {
            CNode* uu = u->parent();
            CNode* dd = u->children().first();
            u->edge().disconnect();
            dd->edge().disconnect();
            connectNodes(uu,dd);
            m_nodes.remove( u );
       }

    }
    else
    {
      CSubRoot* s = (CSubRoot*)n;
      CBuffer* b = (CBuffer*)n->parent();
      s->edge().disconnect();
      b->type( STEINER );
      s->type( STEINER );
      b->load(0);
      connectNodes( b, s );
    }
}

void CNet::reInsertBuffer(CNode* n,  const double load, const double res, const
double cap)
{
    CSubRoot* s = (CSubRoot*)n;
    CBuffer* b = (CBuffer*)n->parent();
    s->edge().disconnect();
    s->type( SUBROOT );
```

```
    s->outRes( res );
    s->outCap(cap );
    s->done( true );
    b->type( BUFFER );
    b->load( load );
    connectNodes( b, s );
}


int CNet::numBuffers() const
{
QPtrListIterator<CNode> n(m_nodes);
int c = 0;
for(n.toFirst();n.current();++n)
if((*n)->type()==BUFFER) c++;
return c;
}

void CNet::removeBuffers()
{
QPtrList<CNode> nodes = m_nodes;
nodes.setAutoDelete(false);
while(!nodes.isEmpty())
{
CNode* node = nodes.take(0);
if(node->type()!=SUBROOT && node->type()!=BUFFER) continue;
if(node->children().count()>1)
{
CNode* st = new CNode(STEINER,*node);
CNode* up = node->parent();
node->edge().disconnect();
connectNodes(up,st);
QPtrList<CNode> children =node->children();
while(!children.isEmpty())
{
CNode* child = children.take(0);
child->edge().disconnect();
connectNodes(st,child);
}
m_nodes.remove(node);
m_nodes.append(st);
}else
{
CNode* down = node->children().first();
CNode* up = node->parent();
down->edge().disconnect();
node->edge().disconnect();
connectNodes(up,down);
m_nodes.remove(node);
}
}

}


void CNet::segment(const double lump)
{
  double len;
double deltaX,deltaY,newX,newY;
int numSegments;
CNode* up, *down;
CNode* seg;
CNode* node;

QPtrList<CNode> nodes = m_nodes;
while( !nodes.isEmpty() )
{
down = nodes.take(0);
if(down->parent()==NULL) continue;
len = down->edge().manhDist();
if( len<lump ) continue;
up = down->parent();
down->edge().disconnect();
numSegments = int(ceil(len/lump));
deltaX = (down->x()-up->x())/numSegments;
deltaY = (down->y()-up->y())/numSegments;
newX = up->x();
newY = up->y();
node = up;
for (int i =1; i < numSegments; i++)
{
  newX += deltaX;
  newY += deltaY;
  seg = new CNode(AWE,CPoint(newX,newY));
  m_nodes.append(seg);
  connectNodes(node, seg);
  node = seg;
}
connectNodes(node, down);
  }
}

//clean awe nodes used for delay calculation
void CNet::desegment()
{
QPtrList<CNode> nodes = m_nodes;
CNode *up,*down,*node;
while(!nodes.isEmpty())
{
node = nodes.take(0);
if( node->type()!=AWE ) continue;
up = node->parent();
down = node->children().first();
node->edge().disconnect();
down->edge().disconnect();
m_nodes.remove(node);
connectNodes(up,down);
}
}

void CNet::computeCapAndR()
{
QPtrListIterator<CNode> nIt(m_nodes);
for(nIt.toFirst();nIt.current();++nIt) (*nIt)->computeCapAndR();
}


void CNet::saveDV()
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
    QPtrListIterator<CNode> nIt(m_nodes);
    for(nIt.toFirst();nIt.current();++nIt)
    {
        CNode* n = *nIt;
        if(n->type()!=SINK) continue;
        if(n->parent()==NULL) continue;
        CSink* s = (CSink*)n;
        s->saveDV();
    }
}

double CNet::DVImp() const
{
    //std::cout << "\n calc improvement\n";
    double maxImp = -INF;
    QPtrListIterator<CNode> nIt(m_nodes);
    for(nIt.toFirst();nIt.current();++nIt)
    {
        CNode* n = *nIt;
        if(n->type()!=SINK) continue;
        if(n->parent()==NULL) continue;
        CSink* s = (CSink*)n;
        if(s->DVImp()>maxImp) maxImp = s->DVImp();
    }
    return maxImp;
}

void CNet::buildTree(CNode* rootN, QString root, int x, int y, const
QPtrList<CCctPath> T, QPtrList<CCctPath>* P, QTextStream& f)
{
extern CGRoute route;
extern CRte rte;

double res = (double)rte.res();
QString newroot;
CNode* newrootN;
int x1,y1,x2,y2;
x1=x;y1=y;
QPtrListIterator<CCctPath> itT(T);
f << "\nPARAMS " << root << " " << x << " " << y << " : " << T.count()
<< " " << P->count() << "\n";
QPtrListIterator<CCctPath> tIt(T);
f << "Candidate paths\n";
for(tIt.toFirst();tIt.current();++tIt)
{
CCctPath* t = *tIt;
f << t->pin1() << "\t( " << t->x1() << " " << t->y1() << ") ";
f << t->pin2() << " -- ( " << t->x2() << " " << t->y2() <<
")\n";
}
QPtrListIterator<CCctPath> pIt(*P);
f << "Remaining paths paths\n";
for(pIt.toFirst();pIt.current();++pIt)
{
CCctPath* p = *pIt;
f << p->pin1() << "\t( " << p->x1() << " " << p->y1() << ") ";
f <<p->pin2() << " -- ( " << p->x2() << " " << p->y2() << ")\n";
}

// if (T.isEmpty()) std::cout << "EMPTY\n"; else std::cout << "Not
EMPTY\n";
for(itT.toFirst();itT.current();++itT)
{
CCctPath* path = *itT;
f << "path " << path->pin1() << " (" <<path->x1() << " " <<
path->y1() << ") ";
f << path->pin2() << "-- (" <<path->x2() << " " << path->y2() <<
")\n";
if(root!="" && path->pin1()!="" && root==path->pin1())
{
assert(x1==path->x1());
assert(y1==path->y1());
x2=path->x2();
y2=path->y2();
newroot = path->pin2();
if(newroot!="")
{
f << "1\n";
newrootN = pin(newroot);
assert(newrootN!=NULL);
}else newrootN = new CNode(STEINER,
CPoint(double(x2)/res,double(y2)/res));
f << "1connect " << path->pin1() << " (" << x << " " <<
y;
f << ") - " << path->pin2() << " (" <<path->x2() << " "
<< path->y2() << ")\n";
}else if(root!="" && path->pin2()!="" && root==path->pin2())
{
assert(x1==path->x2());
assert(y1==path->y2());
x2=path->x1();
y2=path->y1();
newroot = path->pin1();
if(newroot!="")
{
f << "2\n";
newrootN = pin(newroot);
assert(newrootN!=NULL);
}else newrootN = new CNode(STEINER,
CPoint(double(x2)/res,double(y2)/res));
f << "2connect " << path->pin2() << " (" << x << " " <<
y;
f << ") - " << path->pin1() << " (" <<path->x1() << " "
<< path->y1() << ")\n";
}else if(path->x1()==x1 && path->y1()==y1 && path->pin1()=="")
{
//assert(path->pin1()==root);
x2=path->x2();
y2=path->y2();
newroot = path->pin2();
if(newroot!="")
{
f<< "3\n";
f << newroot << "\n";
newrootN = pin(newroot);
assert(newrootN!=NULL);
}else newrootN = new CNode(STEINER,
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CPoint(double(x2)/res,double(y2)/res));
f << "3connect " << path->pin1() << " (" << x << " " <<
y;
f << ") - " << path->pin2() << " (" <<path->x2() << " "
<< path->y2() << ")\n";
}else if(path->x2()==x && path->y2()==y && path->pin2()=="")
{
//assert(path->pin2()==root);
x2=path->x1();
y2=path->y1();
newroot = path->pin1();
if(newroot!="")
{
f << "4\n";
newrootN = pin(newroot);
assert(newrootN!=NULL);
}else newrootN = new CNode(STEINER,
CPoint(double(x2)/res,double(y2)/res));
f << "4connect " << path->pin2() << " (" << x << " " <<
y;
f << ") - " << path->pin1() << " (" <<path->x1() << " "
<< path->y1() << ")\n";
}else {f << "Error buildTree\n";assert(false);}
QPtrList<CCctPath> S;
QPtrListIterator<CCctPath> itP(*P);
for(itP.toFirst();itP.current();++itP)
{
CCctPath* path = *itP;
if( (x2==path->x1() && y2==path->y1()) ||
(x2==path->x2() && y2==path->y2()))
S.append(path);
}
QPtrListIterator<CCctPath> itS(S);
for(itS.toFirst();itS.current();++itS)
{
CCctPath* path = *itS;
P->remove(path);
}
if(S.isEmpty() && newroot=="")
{
f << "noConnect\n";
delete newrootN;
newrootN=NULL;

}else if( newroot=="")
{
m_nodes.append( newrootN );
f << "connect1 " << root << " -> " << newroot << "\n";
f << x1 << " " << y1 << " -> " << x2 << " " << y2 <<
"\n";
connectNodes(rootN,newrootN);
f << "connected11\n";
CLayer* l1 = route.layer(path->layer());
CLayer* l2;
f << "connected12\n";
if(l1->dir()==VERTICAL) {if(l1->next()!=NULL)
l2=l1->next(); else l2=l1->prev();}
else if(l1->dir()==HORIZONTAL) {if(l1->next()!=NULL)
l2=l1->next(); else l2=l1->prev();}
f << "connected13\n";
newrootN->edge().layer( CLayerPair(l1,l2) );
f << "connected14\n";
f << newrootN->edge().layer().vl()->name() << "\n";
f << newrootN->edge().layer().hl()->name() << "\n\n";
}else
{
f << "connect2 " << root << " -> " << newroot << "\n";
f << x1 << " " << y1 << " -> " << x2 << " " << y2 <<
"\n";
connectNodes(rootN,newrootN);
CLayer* l1 = route.layer(path->layer());
CLayer* l2;
if(l1->dir()==VERTICAL) {if(l1->next()!=NULL)
l2=l1->next(); else l2=l1->prev();}
else if(l1->dir()==HORIZONTAL) {if(l1->next()!=NULL)
l2=l1->next(); else l2=l1->prev();}
newrootN->edge().layer( CLayerPair(l1,l2) );
std::cout << newrootN->edge().layer().vl()->name() <<
"\n";
std::cout << newrootN->edge().layer().hl()->name() <<
"\n\n";
}
//if (S.isEmpty()) std::cout << "SEMPTY\n"; else std::cout << "S
Not EMPTY\n";
buildTree(newrootN, newroot,x2,y2,S,P,f);
}
}
```

```
#include "cnet.h"
#include "cnode.h"
#include "gwire.h"
#include "clayer.h"
#include "clayerpair.h"
#include "csink.h"

void CNet::rteWrite(QTextStream& f)
{
f << "(Net " << *this << "\n";
QPtrListIterator<CNode> nIt(m_nodes);
QPtrListIterator<gwire> wIt(*this);
for(nIt.toFirst();nIt.current();++nIt)
{
CNode* down = *nIt;
if(down->type()!=SINK) continue;
gwire wire;
do
{
first();while(current()->to()!=down) next();
assert(current()->from()==down->parent());
wire += *current();
down =  down->parent();
}while(down->type()!=SINK&& down->type()!=ROOT);
nodeType tp = wire.from()->type();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
assert(tp==ROOT || tp==SINK);
tp = wire.to()->type();
assert(tp==SINK);
wire.rteWrite(f);
}
f << ")\n";
}

void CNet::grteWrite(QTextStream& f)
{
f << "(Net " << *this << "\n";
QPtrListIterator<CNode> nIt(m_nodes);
QPtrListIterator<gwire> wIt(*this);
for(nIt.toFirst();nIt.current();++nIt)
{
CNode* down = *nIt;
if(down->type()!=SINK) continue;
gwire wire;
do
{
first();while(current()->to()!=down) next();
assert(current()->from()==down->parent());
wire += *current();
down =  down->parent();
}while(down->type()!=SINK&& down->type()!=ROOT);
nodeType tp = wire.from()->type();
assert(tp==ROOT || tp==SINK);
       tp = wire.to()->type();
assert(tp==SINK);
wire.grteWrite(f);
}
f << ")\n";
}

void CNet::paint(QPainter* p)
{
QPtrListIterator<CNode> nIt(m_nodes);
QPtrListIterator<gwire> wIt(*this);
for(nIt.toFirst();nIt.current();++nIt)
{
CNode* down = *nIt;
if(down->type()!=SINK) continue;
gwire wire;
do
{
   first();while(current()->to()!=down) next();
assert(current()->from()==down->parent());
wire += *current();
down =  down->parent();
}while(down->type()!=SINK&& down->type()!=ROOT);
nodeType tp = wire.from()->type();
assert(tp==ROOT || tp==SINK);
       tp = wire.to()->type();
assert(tp==SINK);
wire.paint(p);
}
}

void CNet::dump(QString  msg)
{
     QFile F("net.log");
     F.open(IO_WriteOnly | IO_Append);
     QTextStream f(&F);
     f << "\n";
     f << "########################################\n";
     if( maxDV() > 0 ) f << "*"; else f << " ";
     f << "net\t: " << *this << "\n";
     f << "message\t: " << msg << "\n";
     f << "-----------------------------------\n";
      computeCapAndR();
      CNode* n = root();
      f << "layer pair\t: " << layer().vl()->name() << ":" <<
layer().hl()->name()<< "\n";
      f << "maximum DV\t: " << maxDV() << " ps\n";
      f << "#Buffers \t: " << numBuffers() << "\n";
      f << "Length \t: " << length() << "\n";
      f << "------------------------------------\n";
      QPtrList<CNode> que;
      que.append(n);
      while(!que.isEmpty())
      {
             CNode* n = que.take(0);
             QPtrListIterator<CNode> nIt(n->children());
             for(nIt.toFirst();nIt.current();++nIt) que.append( *nIt );
             f << ::nodeTypeStr(n->type()) << "(" << n << ")\n";
             f << "\tLength\t= " << n->edge().manhDist() << " um\n";
             f << "\tRes\t= " << n->edge().res() << " ohm\n";
             f << "\tCap\t= " << n->cap() << " pF\n";
     f << "\tLayer\t= ";
     if(n->edge().layer().vl()!=NULL) f << " " <<
n->edge().layer().vl()->name();
     if(n->edge().layer().hl()!=NULL) f << " " <<
n->edge().layer().hl()->name();
     f << "\n";
             if(n->type()==SINK )
             {
                 CSink* s = (CSink*)n;
                 f << "\tDelay\t= " << s->delay() << " ps\n";
                 f << "\trat\t= " << s->rat() << " ps\n";
                 f << "\tDV\t= " << s->dv() << " ps\n";
              }
              f << "\tP(x,y)\t= (" << n->x() << "," << n->y() << ")\n";
              if(n->parent()!=NULL)
              {
                  n = n->parent();
                  f << "\tParent\t= " <<  ::nodeTypeStr(n->type()) << "(" << n
<< ")\n";
              }
      }
      F.close();
}


#include "cnetlist.h"
#include "ccctnet.h"
#include "ccctcomp.h"
#include "ccctimage.h"
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#include "ccctpin.h"
#include "cpla.h"
#include "clib.h"

CNetList::CNetList()
{
m_nets.setAutoDelete(true);
}
CNetList::~CNetList()
{
m_nets.clear();
}

bool CNetList::load(QString fileName)
{
extern CPla pla;
extern CLib lib;

QFile netFile(fileName);
if( !netFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open  " << fileName << "\n";
return false;
}
QTextStream net(&netFile);

QString val,var;
QString netName,pin,comp,pinName;
CCctNet* pNet = NULL;
CCctComp* pComp;
CCctImage* pImage;
CCctPin* pPin;

while( !net.atEnd() )
{
net >> var;
if( var.contains(")") ) continue;
var = var.replace( QRegExp("\\(|\\)"), "" );
if( var=="net" )
{
net >> netName;
pNet = new CCctNet( netName );
m_nets.insert( pNet->name(), pNet );
}
else if( var=="pins" )
{
QValueList<QString> pins;
do
{
net >> pin;
if( pin==")" ) break;
comp = QStringList::split("-", pin ).front();
pinName = QStringList::split("-",pin ).back();

pinName = pinName.replace( QRegExp("\\)"), "" );
pNet->addPin( comp+"-"+pinName );
pComp = pla.comp(comp);
if(pComp==NULL)
{
std::cout << "Warning : " <<
comp+"-"+pinName << " of " << netName;
std::cout << ", reference to the
component could not be found in placement\n";
}else
{
pImage = lib.image(pComp->image());
if(pImage==NULL)
{
std::cout << "Warning : " <<
comp+"-"+pinName << " of " << netName;
std::cout << ", reference to the
 component image could not be found in library\n";
}else
{
pPin = pImage->pin(pinName);
if(pPin==NULL)
{
std::cout << "Warning :
" << comp+"-"+pinName << " of " << netName;
std::cout << ",
reference to the  image pin could not be found in library\n";
}
}
}
}while( !pin.contains(")") );
}
}
netFile.close();
return true;
}

bool CNetList::save(QString fileName)
{
std::cout << "CNetList::save() not implemented yet " << fileName <<
"\n";
return true;
}
#include <defwWriter.hpp>
#include <defwWriterCalls.hpp>
#include "cnetpin.h"

CNetPin::CNetPin( const QString instance,const QString name, const int
synthesized)
:m_instance(instance)
,m_name(name)
,m_synthesized(synthesized)
{
}

CNetPin::~CNetPin()
{
}

void CNetPin::defw() const
{
defwNetConnection(m_instance, m_name, m_synthesized);
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#include "cnet.h"

#include "cnode.h"
#include "csource.h"
#include "csink.h"
#include "cbuffer.h"
#include "csubroot.h"
#include "clayerpair.h"
#include "clayer.h"

CNode::CNode(const nodeType type, const CPoint point)
:CPoint(point)
,m_type(type)
,m_parent(NULL)
,m_cap(0)
,m_delay(INF)
{
m_children.setAutoDelete(false);
}

CNode::~CNode()
{
m_children.clear();
m_moment.clear();
}

void CNode::computeCapAndR()
{
if(m_parent==NULL) return;
m_edge.computeR();

CLayerPair layer = m_edge.layer();
m_cap = 0.5 * ( m_edge.xDist() * layer.hl()->cap() + m_edge.yDist() *
layer.vl()->cap() );
QPtrListIterator<CNode> nIt(m_children);
for(nIt.toFirst();nIt.current();++nIt)
{
CEdge edge = (*nIt)->edge();
layer = edge.layer();
m_cap += 0.5 * ( edge.xDist() * layer.hl()->cap() + edge.yDist()
* layer.vl()->cap() );
}
}

void CNode::resetSubtree(const int ord)
{
m_moment.resize(2*ord);
m_edge.resizeCurrent(2*ord);
   m_moment[0] = 1.0;
   for (int i = 0; i < 2*ord; i++)
     m_edge.current(0.0, i);
QPtrListIterator<CNode> n(m_children);
for(n.toFirst();n.current();++n) if( (*n)->type()!=SUBROOT )
(*n)->resetSubtree(ord);
}

void CNode::computeCurrent(const int ord)
{
QPtrListIterator<CNode> n(m_children);

for(n.toFirst();n.current();++n) if( (*n)->type()!=SUBROOT ) (*n)->
computeCurrent(ord);
m_edge.increCurrent( cap() *  m_moment[ord-1], ord );//which cap
if( m_type!=ROOT && m_type!=SUBROOT )
m_parent->edge().increCurrent( m_edge.current(ord), ord );
}

void CNode::computeMoment(const int ord)
{
m_moment[ord] =  m_parent->moment(ord) -

m_edge.current(ord) * m_edge.res() -

m_edge.current(ord-1) * m_edge.induc();

QPtrListIterator<CNode> n(m_children);
for(n.toFirst();n.current();++n) if( (*n)->type()!=SUBROOT )
(*n)->computeMoment(ord);
}
```

```
/***************************************************************************
                          coord.cpp  -  description
                             -------------------
    begin                : Tue Jan 20 2004
    copyright            : (C) 2004 by
    email                :
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
 ***********************************************************************/

#include "coord.h"

Coord::Coord(int x, int y, int z)
:m_x(x)
,m_y(y)
,m_z(z)
{
}

Coord::~Coord()
{
}

bool Coord::operator<(const Coord &c) const
{
if(m_x < c.m_x) return true;
if(m_x > c.m_x) return false;
if(m_y < c.m_y) return true;
if(m_y > c.m_y) return false;
if(m_z < c.m_z) return true;
if(m_z > c.m_z) return false;
return false; // same node
}

bool Coord::operator==(const Coord &c) const
{
return (m_x == c.m_x) && (m_y == c.m_y) && (m_z == c.m_z);
}

#include "cpathcut.h"
#include "gwire.h"
#include "gpath.h"

CPathCut::CPathCut(gwire* wire, const  int lb, const int ub, gpath* path)
:m_wire(wire)
{
first.first = lb;
first.second = ub;
second.setAutoDelete(false);
second.append(path);
}

CPathCut::CPathCut()
:m_wire(NULL)
{
first.first = -1;
first.second = -1;
second.clear();
}

CPathCut::~CPathCut()
{
second.clear();
}
#include "cpla.h"
#include "clib.h"
#include "ccctcomp.h"
#include "ccctimage.h"

CPla::CPla()
{
m_comps.setAutoDelete(true);
}
CPla::~CPla()
{
m_comps.clear();
}

bool CPla::load(const QString fileName)
{
extern CLib lib;

QFile plaFile(fileName);
if( !plaFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open " << fileName << "\n";
return false;
}
QTextStream pla(&plaFile);

   QString val,var,imageName,compName;
CCctComp* pComp;
CCctImage* pImage;
double x,y;

while( !pla.atEnd() )
{
pla >> var;
var = var.replace( QRegExp("\\(|\\)"), "" );
if( var=="component" )
{
pla >> imageName;
}
else if( var=="place" )
{
pla >> compName;
pComp = new CCctComp( compName, imageName );
pImage = lib.image( imageName );
if( pImage==NULL )
{
std::cout << "Warning : " << ", Instance of " <<
compName << " : " << imageName;
std::cout << ", could not be found in
library\n";
}
pla >> x >> y;
pComp->place(x,y);
m_comps.insert( pComp->name(), pComp );
}
}
plaFile.close();
return true;
}
bool CPla::save(const QString fileName)
{
std::cout << "CPla::save() not implemented yet " << fileName << "\n";
```

```
return   true;
}
#include "global.h"
#include "cpoint.h"

CPoint::CPoint(const double x, const double y)
:m_x(x)
,m_y(y)
{
}
CPoint::CPoint(const CPoint& p)
:m_x(p.x())
,m_y(p.y())
{
}
CPoint::~CPoint()
{
}

bool operator==(const CPoint& p1, const CPoint& p2)
{
return fabs(p1.x() - p2.x()) < TINY && fabs(p1.y() - p2.y()) < TINY;
}

QTextStream& operator<<(QTextStream& dmp, CPoint& p)
{
dmp << p.m_x << " " << p.m_y;
return dmp;
}

#include "cpoly.h"

CPoly::CPoly()
{
}
CPoly::~CPoly()
{
clear();
}

CRect CPoly::boundingRect() const
{
if( isEmpty() ) return CRect();

CRect r( front().x(), front().y(), 0, 0 );
QValueListConstIterator<CPoint> p;
for(p=begin();p!=end();p++)
{
if( (*p).x() < r.left() ) r.left( (*p).x() ); else if ( (*p).x()
> r.right() ) r.right( (*p).x() );
if( (*p).y() < r.bottom() ) r.bottom( (*p).y() ); else if (
(*p).y() > r.top() ) r.top( (*p).y() );
}
return r;
}

#include <qwmatrix.h>
#include "crect.h"
#include "global.h"

CRect::CRect(const double left, const double bottom, const double width, const
double height)
:m_left(left)
,m_bottom(bottom)
,m_right(left+width)
,m_top(bottom+height)
{
}

CRect::CRect(const CPoint bottomLeft, const CPoint topRight)
:m_left(bottomLeft.x())
,m_bottom(bottomLeft.y())
,m_right(topRight.x())
,m_top(topRight.y())
{
}

//construct rect from QRect
CRect::CRect(const QRect r)
:m_left(r.left())
,m_bottom(r.top())
,m_right(r.left()+r.width())
,m_top(r.top()+r.height())
{
}

CRect::~CRect()
{
}

//converts this rect to QRect
QRect CRect::toQRect() const
{
QRect r;
r.setLeft((int)floor(m_left));
r.setWidth((int)floor(width()));
r.setTop((int)floor(m_bottom));
r.setHeight((int)floor(height()));
return r;
}

CRect CRect::scale(const double s) const
{
return CRect(m_left*s, m_bottom*s, (m_right-m_left)*s,
(m_top-m_bottom)*s );
}
CRect CRect::translate(const CPoint p) const
{
return CRect(m_left+p.x(), m_bottom+p.y(), m_right-m_left,
m_top-m_bottom );
}

CRect CRect::translate(const double dx, const double dy) const
{
return translate( CPoint(dx,dy) );
}

//returns true if this intersects with rect
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
bool CRect::intersects(const CRect rect) const
{
   return (  max( m_left, rect.left() ) < min( m_right, rect.right()
) &&
      max( m_bottom, rect.bottom() ) < min(
m_top, rect.top() ) );
}

//returns intersect rectangle of this and rect
CRect CRect::intersect(const CRect rect) const
{
if( !intersects(rect) ) return CRect();
    double left = max( m_left, rect.left() );
    double right = min( m_right, rect.right() );
    double bottom = max( m_bottom, rect.bottom() );
    double top = min( m_top, rect.top() );
    return CRect(left,bottom,right-left,top-bottom);
}

// return unite of the this and rect
CRect CRect::unite(const CRect rect) const
{
    double left = min( m_left, rect.left() );
    double right = max( m_right, rect.right() );
    double bottom = min( m_bottom, rect.bottom() );
    double top = max( m_top, rect.top() );
    return CRect(left,bottom,right-left,top-bottom);
}

CRect CRect::normalize() const
{
double left,right,bottom,top;
if(m_left<m_right) {left=m_left;right=m_right;}else
{right=m_left;left=m_right;}
if(m_bottom<m_top) {bottom=m_bottom;top=m_top;}else
{top=m_bottom;bottom=m_top;}
return CRect(left,bottom,right-left,top-bottom);
}
//retruns absolute rect of the def pin
CRect CRect::orientDefPin(const CPoint placement,const int orient) const
{
QRect r = (CRect( placement.x()+m_left, placement.y()+m_bottom, width(),
height() ) * RES ).toQRect();

QWMatrix wm;
wm.translate(-RES*placement.x(),-RES*placement.y());
r = wm.mapRect( r );
wm.reset();
int translate = 0;
int rotate=0;
switch(orient)
{
case FN: translate  = -2*r.left() - r.width();
case N: rotate = 0; break;
case FS: translate = 2*r.left() + r.width();
case S: rotate = 180; break;
case FW: translate =  2*r.bottom() - r.height();
case W: rotate = 90; break;
case FE: translate =  -2*r.bottom() - r.height();
case E: wm.rotate(-90); break;
default : rotate = 0;translate=0;
}
wm.rotate( rotate );
r = wm.mapRect( r ).normalize();
wm.reset();
wm.translate( translate, 0 );
r = wm.mapRect( r ).normalize();;
wm.reset();

wm.translate(RES*placement.x(),RES*placement.y());
r = wm.mapRect( r );
return CRect(r)/RES;
}

//returns absolute rect of the lef pin
CRect CRect::orientLefPin(const CRect rec, const int orient)  const
{

   QRect r = (CRect( rec.left()+m_left, rec.bottom()+m_bottom,width(),height())
* RES).toQRect();
int width = int(RES * rec.width());
int height = int(RES *rec.height());
QWMatrix wm;
wm.translate(-RES*rec.left(),-RES*rec.bottom() );
r = wm.mapRect( r );
wm.reset();
int rotate = 0;
int dx = 0; int dy = 0;
switch(orient)
{
case FN:
case N:rotate = 0;dx=0;dy=0;break;
case FS:
case S:rotate = 180;dx = width;dy =height;break;
case FW:
case W: rotate = 90;dx = height;dy=0;break;
case FE:
case E: rotate = -90;dx=0;dy=width;break;
default : rotate =0;dx=0;dy=0;
}
wm.rotate( rotate );
r = wm.mapRect( r ).normalize();
wm.reset();
wm.translate( dx, dy );
r = wm.mapRect( r ).normalize();
wm.reset();

   dx = 0;
switch(orient)
{
case FN:
case FS: dx = -2 * r.left() - r.width() + width;break;
case FW:
case FE: dx = -2 * r.left() - r.width() + height;break;
default : dx = 0;
}

wm.translate(dx,0);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
r = wm.mapRect( r ).normalize();
wm.reset();

wm.translate(RES*rec.left(),RES*rec.bottom() );
r = wm.mapRect( r ).normalize();
return CRect(r)/RES;
}



#include "crte.h"
#include "ccctnet.h"
#include "ccctwire.h"
#include "ccctpath.h"

CRte::CRte()
{
}


CRte::~CRte()
{
}

bool CRte::load(const QString fileName)
{
QFile rteFile(fileName);
if( !rteFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open " << fileName << "\n";
return false;
}
QTextStream rte(&rteFile);

QString var,val,net,layer,pin1,pin2;
int width,x1,x2,y1,y2;
CCctNet* pNet;
CCctWire* pWire=NULL;
CCctPath* pPath;
while(!rte.atEnd())
{
rte >> var;
var = var.replace( QRegExp("\\(|\\)"),"");
// std::cout << "var " << var << "\n";
if(var=="resolution")
{
rte >> val >> m_res;
// std::cout << "res " << m_res << "\n";
}else if(var=="Net" || var=="net")
{
rte >> net;
// std::cout << "net " << net << "\n";
pNet = new CCctNet(net);
m_nets.insert(pNet->name(),pNet);
}else if(var=="gwire")
{
// std::cout << "new gwire\n";
pWire = new CCctWire();
pNet->addWire(pWire);
}else if(var=="gpath")
{
// std::cout << "new gpath\n";
// pPath = new CCctPath();
// pWire->addPath(pPath);
}else if(var=="layer" && pWire!=NULL)
{
// std::cout << "layer\n";
bool done = false;
QStringList layers;
do
{
rte >> layer;
if(layer.contains(")")) done = true;
layer = layer.replace( QRegExp("\\)"),"");
layers += layer;
// std::cout << layer << "\n";
}while(!done);
rte >> width >> x1 >> y1 >> x2 >> y2;
// std::cout << width << " " << x1 << " " << y1 << " : " <<
x2 << " " << y2 << "\n";
while(!layers.isEmpty())
{
layer = layers.front();
pPath = new CCctPath(layer,width,x1,y1,x2,y2);
pWire->addPath(pPath);
layers.pop_front();
}
}else if(var=="connect")
{
rte >> val >> val >> pin1 >> val >> val >> pin2;
pin1 = pin1.replace( QRegExp("-A\\)+|\\)+"), "" );
pin2 = pin2.replace( QRegExp("-A\\)+|\\)+"), "" );
//std::cout << "pins " << pin1 << " -> " << pin2 <<
"\n";
pWire->pins(pin1,pin2);
}

}

rteFile.close();
return true;
}

bool CRte::save(const QString fileName)
{
return true;
}

#include "csink.h"

CSink::CSink( const QString name,
const QString layer,
const CPoint point,
const double load,
const double rat)
:QString(name)
,CNode(SINK,point)
,m_layer(layer)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
,m_load(load)
,m_rat(rat)
,m_sDV(INF)
{
}
CSink::~CSink()
{
}

double CSink::DVImp() const
{
    double orgDV = m_sDV;
    double newDV = dv();
    if(orgDV<0 && newDV<0 ) return 0; else return orgDV-newDV;
  // std::cout << orgDV << "\t" << newDV << "\n";
  // if(orgDV > 0 ) return 1-(newDV/orgDV); else return 0;
}
#include "global.h"
#include "csize.h"

CSize::CSize(const double width, const double height)
:m_width(width)
,m_height(height)
{
}

CSize::~CSize()
{
}

#include "csource.h"

CSource::CSource( const QString name,
const
QString layer,
const
CPoint point,
const
double res,
const
double cap)
:QString(name)
,CNode(ROOT,point)
,m_layer(layer)
,m_outRes(res)
,m_outCap(cap)
{
}
CSource::~CSource()
{
}


#include "cstr.h"
#include "ccctlayer.h"
```

```
#include "ccctkeepout.h"

CStr::CStr()
:m_hasDiePoly(false)
{
m_layers.setAutoDelete(true);
m_keepouts.setAutoDelete(true);
}

CStr::~CStr()
{
m_layers.clear();
m_keepouts.clear();
}

CCctLayer* CStr::layer(const QString name) const
{
return m_layers[name];
}

bool CStr::load(const QString fileName)
{
QFile strFile(fileName);
if( !strFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open  " << fileName << "\n";
return false;
}

QTextStream str(&strFile);


QString var,val,val1,val2;
CCctLayer* pLayer;
QString layer,type,direction;
double width,clear;
int layerInd = 0;
CRect rect;
double x1,y1,x2,y2;

while( !str.atEnd() )
{
str >> var;
var = var.replace( QRegExp("\\(|\\)"), "" );
if( var=="boundary" ) //die area
{
m_hasDiePoly =  true;
str >> val >> val >> val;
str >> val1 >> val2;
do
{
m_diePoly.append( val1.toDouble(),
val2.toDouble() );
str >> val1 >> val2;
}while( val1!=")" && val2!=")" );
}else if( var=="layer" )
{
str >> layer;
str >> val >> type;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
type = type.replace( QRegExp("\\)"),"" );
str >> val >> direction;
direction = direction.replace( QRegExp("\\)"),"" );
str >> val >> val >> width >> val >> val >> clear;
pLayer = new CCctLayer( layer , layerInd );
if( type=="signal" ) pLayer->setType( ROUTING );
pLayer->setDirection(
::layerDirection(direction.upper()) );
pLayer->setWidth( width );
pLayer->setClear( clear );
m_layers.insert( pLayer->name(), pLayer );
layerInd++;
}else if(var=="keepout" )
{
str >> val >> layer >> x1 >> y1 >> x2 >> y2;
layer = layer.upper();
rect = CRect(x1,y1,x2-x1,y2-y1);
m_keepouts.append(new CCctKeepout(layer,rect));
}
}
strFile.close();

return true;
}

bool CStr::save(const QString fileName)
{
  std::cout << "CStr::save() not implemented yet " << fileName << "\n";
return true;
}
#include "csubroot.h"

CSubRoot::CSubRoot(const CPoint point, const double res, const double cap)
:CNode(SUBROOT,point)
,m_outRes(res)
,m_outCap(cap)
,m_done(false)
{
}
CSubRoot::~CSubRoot()
{
}
#include "ctile.h"
#include "clayer.h"

CTile::CTile(CLayer* layer, const int x, const int y, const QSize size)
:QPoint(x,y)
,QSize(size)
,m_layer(layer)
{
}

CTile::CTile(const CTile& tile)
:QPoint((QPoint)tile)
,QSize((QSize)tile)
,m_layer(tile.m_layer)
{
}
```

```
CTile::CTile()
:QPoint()
,QSize()
,m_layer(NULL)
{
}

CTile::~CTile()
{
}


CTileEdge::CTileEdge(CLayer* layer, const int track, const int edge, const QSize
tileSize)
:m_layer(layer)
,m_track(track)
,m_edge(edge)
,m_size(tileSize)
{
}

CTileEdge::CTileEdge(const CTileEdge& tileEdge)
:m_layer(tileEdge.m_layer)
,m_track(tileEdge.m_track)
,m_edge(tileEdge.m_edge)
,m_size(tileEdge.m_size)
{
}

CTileEdge::CTileEdge()
:m_layer(NULL)
,m_size(QSize())
{
}


CTileEdge::~CTileEdge()
{
}

#include "ctim.h"

CTim::CTim()
{
m_riseTime.setAutoDelete(true);
m_fallTime.setAutoDelete(true);
}
CTim::~CTim()
{
m_riseTime.clear();
m_fallTime.clear();
}

pinDirectionType CTim::dir(const QString net, const QString comp) const
{
QValueList< QPair< QPair<QString,QString>, double > >* time1 =
m_riseTime[net];
QValueList< QPair< QPair<QString,QString>, double > >* time2 =
m_fallTime[net];
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
QValueListIterator< QPair< QPair<QString,QString>, double > > it;
if(time1!=NULL) for(it=time1->begin();it!=time1->end();it++)
{
if((*it).first.first==comp) return OUT;
if((*it).first.second==comp) return IN;
}
if(time2!=NULL) for(it=time2->begin();it!=time2->end();it++)
{
if((*it).first.first==comp) return OUT;
if((*it).first.second==comp) return IN;
}
return IN;
}

double CTim::rat(const QString net, const QString comp) const
{
double rat=INF;

QValueList< QPair< QPair<QString,QString>, double > >* time1 =
m_riseTime[net];
QValueList< QPair< QPair<QString,QString>, double > >* time2 =
m_fallTime[net];

QValueListIterator< QPair< QPair<QString,QString>, double > > it;
if(time1!=NULL ) for(it=time1->begin();it!=time1->end();it++)
if((*it).first.second==comp) rat = ::min( rat, (*it).second );
if(time2!=NULL) for(it=time2->begin();it!=time2->end();it++)
if((*it).first.second==comp) rat = ::min( rat, (*it).second );
return rat*2+2*rat*double(rand()%10)/100;
}


bool CTim::load(const QString fileName)
{
   QFile dataFile(fileName);
if( !dataFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open  " << fileName << "\n";
return false;
}
QTextStream data(&dataFile);

QString conn,val,net,compFrom,compTo,dir;
double delay;
QStringList connList;

data.readLine();
while( !data.atEnd() )
{
data >> conn >> val >> val >> val >> val >> val >> delay >> val
>> val >> val >> val;
delay = delay * 1000.0;
connList = QStringList::split( ":", conn);
net = connList[0];
if( net.contains("#") ) net = net.left( net.findRev("#") );
if( net.contains(".") ) net = net.mid( net.findRev(".")+1 );
if( net.contains("/") ) net = net.mid( net.findRev("/")+1 );
compFrom = connList[1];
compTo = connList[3];
dir = connList[4];

QValueList< QPair< QPair<QString,QString>, double > >* time =
new QValueList< QPair< QPair<QString,QString>, double > >::QValueList();
if(dir=="R") {if(m_riseTime[net]==NULL) m_riseTime.insert( net,
time );
else{delete time; time = m_riseTime[net];}}
if(dir=="F") {if(m_fallTime[net]==NULL) m_fallTime.insert( net,
time );
else{delete time;time = m_fallTime[net];}}

QPair<QString,QString> fromTo(compFrom,compTo);
QPair<QPair<QString,QString>,double > delayPair(fromTo, delay);
time->append( delayPair );
}
dataFile.close();
return true;
}

bool CTim::save(const QString fileName)
{
std::cout << "CTim::save() not implemented yet " << fileName << "\n";
return true;
}
#include "ctiming.h"

CTiming::CTiming()
{
m_rats.setAutoDelete(true);
}
CTiming::~CTiming()
{
m_rats.clear();
}

void CTiming::net(const QString net)
{
QValueList<QPair<QString, double> >* time = new
QValueList<QPair<QString, double> >::QValueList();
m_rats.insert(net, time);
}

void CTiming::source(const QString net, const QString source)
{
m_rats[net]->append( QPair<QString, double>::QPair(source, -INF));
}

void CTiming::sink(const QString net, const QString sink, const double rat)
{
m_rats[net]->append( QPair<QString,double>::QPair(sink,rat));
}

pinDirectionType CTiming::dir(const QString net, const QString pin) const
{
if(m_rats[net]==0) return IN;
if(m_rats[net]->first().first==pin) return OUT; else return IN;
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
double CTiming::rat(const QString net, const QString pin) const
{
if(m_rats[net]==0) return INF;
QValueList<QPair<QString,double> > tim = *m_rats[net];
QValueListIterator<QPair<QString,double> > it;
for(it=tim.begin();it!=tim.end();it++){
if((*it).first==pin){
double tim = (*it).second;
tim = tim + tim*(double(rand()%10)/100.0);
        return tim;
}
}
return INF;
}


bool CTiming::load(const QString fileName)
{
    QFile dataFile(fileName);
if( !dataFile.open(IO_ReadOnly) )
{
std::cout << "ERROR : Couldn't open  " << fileName << "\n";
return false;
}
QTextStream data(&dataFile);

while( !data.atEnd() )
{
QString net;
int count;
data >> net >> count;
//std::cout << net;
QValueList<QPair<QString, double> >* time = new
QValueList<QPair<QString, double> >::QValueList();
m_rats.insert(net, time);
for(int i=0;i<count;i++)
{
QString pin;
double rat;
data >> pin >> rat;
//std::cout << "\t" << pin << "\t" << rat;
time->append(QPair<QString,double>::QPair(pin,rat));
}
//std::cout << "\n";
}
dataFile.close();
return true;
}

bool CTiming::save(const QString fileName)
{
QFile dataFile(fileName);
if( !dataFile.open(IO_WriteOnly) )
{
std::cout << "ERROR : Couldn't open  " << fileName << "\n";
return false;
}
QTextStream data(&dataFile);
QDictIterator<QValueList<QPair<QString,double> > > nIt(m_rats);
for(nIt.toFirst();nIt.current();++nIt)
{
QString net = nIt.currentKey();
data << net;
QValueList<QPair<QString,double> > tim = *m_rats[net];
data << "\t" << tim.count();
QValueListIterator<QPair<QString,double> > tIt;
for(tIt=tim.begin();tIt!=tim.end();tIt++)
{
data << "\t" << (*tIt).first << "\t" << (*tIt).second;
}
data << "\n";
}

dataFile.close();
return true;

}
#include "cutil.h"
#include "clef.h"
#include "cdef.h"
#include "cleflayer.h"
#include "clefobs.h"
#include "clefmacro.h"
#include "clefport.h"
#include "clefpin.h"
#include "clefvia.h"
#include "cdefblock.h"
#include "cdefpin.h"
#include "cdefcomp.h"
#include "cdefnet.h"
#include "cdefpath.h"
#include "cgroute.h"
#include "cnet.h"
#include "csource.h"
#include "gwire.h"
#include "clayer.h"
#include "crte.h"
#include "cstr.h"
#include "cdsn.h"
#include "cpla.h"
#include "clib.h"
#include "cnetlist.h"
#include "ctim.h"
#include "ctiming.h"
#include "crte.h"
#include "ccctnet.h"
#include "ccctwire.h"
#include "ccctpath.h"
#include "ccctkeepout.h"
#include "ccctlayer.h"
#include "ccctcomp.h"
#include "ccctimage.h"
#include "ccctpin.h"

CUtil::CUtil()
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}

CUtil::~CUtil()
{
}

bool CUtil::lefdefCongWrite(const QString name)
{
extern CLef lef;
extern CDef def;



QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);


QPtrVector<CLefLayer> layers = lef.routingLayers();
int numLayers = layers.count();

CDefGCell gCell = def.gCell();
int numCols = gCell.numCols();
int numRows = gCell.numRows();

// int (*map)[numCols+1][numRows+1] = new
int[numLayers][numCols+1][numRows+1];
int ***map;
map = new int**[numLayers];
for(int i=0;i<numLayers;i++)
{
map[i] = new int*[numCols+1];
for(int j=0;j<=numCols;j++)
{
map[i][j] = new int[numRows+1];
for(int k=0;k<=numRows;k++)
{
map[i][j][k] = 0;
}
}
}
//add def blocakges
{
QPtrList<CDefBlock> blocks = def.blocks();
while(!blocks.isEmpty())
{
CDefBlock* blk = blocks.take();
if(!blk->hasLayer()) continue;
CLefLayer* layer = lef.layer( blk->layerName() );
if( layer->type()!=ROUTING ) continue;
QValueList<CRect> rects = blk->rectangles();
while(!rects.isEmpty())
{
CRect r =rects.front();rects.pop_front();
addRect(map, blk->layerName(), r, layers);
}
}
```

```
}
//add pdef pins
{
QDict<CDefPin> defpins = def.pins();
    QDictIterator<CDefPin> defpinIt(defpins);
for(defpinIt.toFirst();defpinIt.current();++defpinIt)
{
CDefPin* pDefPin = *defpinIt;
if(!pDefPin->hasLayer()) continue;
CLefLayer* layer = lef.layer( pDefPin->layer() );
if( layer->type()!=ROUTING ) continue;
double x1,y1,x2,y2;
pDefPin->bounds(&x1,&y1,&x2,&y2);
CRect r = CRect(x1,y1,(x2-x1),(y2-y1));
r = r.orientDefPin(CPoint(pDefPin->placementX(),
pDefPin->placementY()),
pDefPin->orient());
addRect(map, pDefPin->layer(), r, layers );
}
}

{
//add comp pins and obstructions
QDict<CDefComp> comps = def.comps();
QDictIterator<CDefComp> compIt(comps);
for(compIt.toFirst();compIt.current();++compIt)
{
CDefComp* pComp = *compIt;
CLefMacro* pMacro = lef.macro(pComp->name());
if(pMacro==NULL) continue;
          //obstructions
QPtrList<CLefObs> obs = pMacro->obs();
QPtrListIterator<CLefObs> obsIt(obs);
for(obsIt.toFirst();obsIt.current();++obsIt)
{
QValueList< QPair<geomType, void*> > items =
(*obsIt)->items();
     QValueListIterator< QPair<geomType, void*> >
itemIt;
QString layerName;
CRect rect;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
              {
              switch((*itemIt).first)
                 {
                 case GEOMLAYER:
                 layerName =
*(QString*)(*itemIt).second;
break;
              case GEOMRECT:
                   rect =
*(CRect*)(*itemIt).second;
                   if( pMacro->hasOrigin() )
rect = rect +
CPoint(
pMacro->originX(),

pMacro->originY() );
```

```
                    rect = rect.orientLefPin(
CRect(
pComp->placementX()/def.units(),

pComp->placementY()/def.units(),

pMacro->sizeX(),pMacro->sizeY() ),

pComp->placementOrient() );
addRect(map, layerName,
rect*def.units(), layers );
                    break;
                        default: std::cout << "not implemented
obstruction shape\n";
                  }

              }
}
//pins
QDict<CLefPin> pins = pMacro->pins();
QDictIterator<CLefPin> pinIt(pins);
for(pinIt.toFirst();pinIt.current();++pinIt)
{
CLefPin* pPin = *pinIt;
QPtrList<CLefPort> ports = pPin->ports();
QPtrListIterator<CLefPort> portIt(ports);
              for(portIt.toFirst();portIt.current();++portIt)
              {
              QValueList< QPair<geomType, void*> >
items = (*portIt)->items();
                 QValueListIterator< QPair<geomType,
void*> > itemIt;
QString layer;
CRect r;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
                 {
                 switch((*itemIt).first)
                    {
                    case GEOMLAYER :
                    layer =
*(QString*)(*itemIt).second;
                       break;
                   case  GEOMRECT :
                    r =
*(CRect*)(*itemIt).second;
                       if( pMacro->hasOrigin()
)
r = r +
CPoint(
pMacro->originX(),

pMacro->originY() );
                       r =
r.orientLefPin(
CRect(
pComp->placementX()/def.units(),

pComp->placementY()/def.units(),
pMacro->sizeX(),pMacro->sizeY() ),

pComp->placementOrient() );
addRect(map, layer, r,
layers );
                      break;
               default: std::cout << "not
implemented pin shape\n";
              }
}
           }
}
}
}

{
//add special nets
{QDict<CDefNet> nets = def.snets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL;
CPoint oldpoint,newpoint;
bool firstpoint = true;
double x1,x2,y1,y2,with;
with = -1;
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-with/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+with/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-with/2;
x2 = newpoint.x();
y2 =
newpoint.y()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name(),
r, layers);
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-with/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+with/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-with/2;
```

```
x2 = newpoint.x();
y2 =
newpoint.y()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name(),
r, layers);
break;}
case PATHWIDTH :
{with = *((int*)item);
break;}
default:break;
}
}
}
}}
//add nets
{QDict<CDefNet> nets = def.nets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL;
CPoint oldpoint,newpoint;
double x1,x2,y1,y2,with;
with = -1;
bool firstpoint=true;
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-def.units()*newlayer->width()/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+def.units()*newlayer->width()/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-def.units()*newlayer->width()/2;
x2 = newpoint.x();
y2 =
newpoint.y()+def.units()*newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name() ,
r, layers);
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-def.units()*newlayer->width()/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+def.units()*newlayer->width()/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-def.units()*newlayer->width()/2;
x2 = newpoint.x();
y2 =
newpoint.y()+def.units()*newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map,newlayer->name(), r
, layers );
break;}
case PATHWIDTH :
{with = *((int*)item);
break;}
default:break;
}
}
}
}}
}

int tnumOver = 0;
int tflow = 0;
int tcap = 0;
f << "\t | #Over \t\t T_Flow \t\t T_Cap\n";
f <<
"-----------------------------------------------------------------------\n";
for(int i=0;i<numLayers;i++)
{
int numOver = 0;
int flow = 0;
int cap = 0;

int scap = (layers[i]->direction()==VERTICAL)?
int(gCell.xStep()/def.units()/layers[i]->pitch()):
int(gCell.yStep()/def.units()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
flow += map[i][j][k];
cap += scap;
if(map[i][j][k]>scap) numOver++;
}
}
f << layers[i]->name();
f << " \t | " << numOver << " \t\t " << flow << " \t\t " << cap
<< "\n";
tnumOver += numOver;
tflow += flow;
tcap += cap;
}
f <<
"-----------------------------------------------------------------------\n";
f << "TOTAL";
f << " \t | " << tnumOver << " \t\t " << tflow << " \t\t " << tcap <<
"\n";
f <<
"-----------------------------------------------------------------------\n\n\n";
```

```
for(int i=0;i<numLayers;i++)
{
int scap = (layers[i]->direction()==VERTICAL)?
int(gCell.xStep()/def.units()/layers[i]->pitch()):
int(gCell.yStep()/def.units()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
f << layers[i]->name() << " " << j << " " << k;
f << " " << map[i][j][k];
if(map[i][j][k]>scap) f << " * ";f << "\n";
}
}
}


//
F.close();
for(int i=0;i<numLayers;i++)
{
for(int j=0;j<=numCols;j++)
{
delete[] map[i][j];
}
delete[] map[i];
}
delete[] map;

return true;
}

bool CUtil::cctCongWrite(const QString name)
{
extern CStr str;
extern CDsn dsn;
extern CPla pla;
extern CLib lib;
extern CNetList netlist;
extern CTim tim;
extern CTiming timing;
extern CRte rte;
extern CGRoute route;

QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);

//layers
QDict<CCctLayer> layers_helper = str.layers();
QPtrVector<CCctLayer> layers(layers_helper.count());
QDictIterator<CCctLayer> it(layers_helper);
for(it.toFirst();it.current();++it)
{
CCctLayer* layer = *it;
layers.insert(layer->index(), layer);
}
int numLayers = layers.count();
```

```
//gcell
CSize gCell = route.gCell();
CRect dieArea = str.diePoly().boundingRect();
int numCols = (int)ceil(dieArea.width()/gCell.width());
int numRows = (int)ceil(dieArea.height()/gCell.height());

// int (*map)[numCols+1][numRows+1] = new
int[numLayers][numCols+1][numRows+1];
int ***map;
map = new int**[numLayers];
for(int i=0;i<numLayers;i++)
{
map[i] = new int*[numCols+1];
for(int j=0;j<=numCols;j++)
{
map[i][j] = new int[numRows+1];
for(int k=0;k<=numRows;k++)
{
map[i][j][k] = 0;
}
}
}
 //add str keepouts
{
QPtrList<CCctKeepout> keepouts = str.keepouts();
while(!keepouts.isEmpty())
{
CCctKeepout* blk = keepouts.take();
addRect(map,blk->layer(),blk->rect(),layers);
}
}
//add wiring
{
QValueList< QPair<QString,CRect> > wires = dsn.wiring();
while(!wires.isEmpty())
{
QPair<QString,CRect> wire = wires.front();
wires.pop_front();
addRect(map,wire.first,wire.second,layers);
}
}
{
//add comp pins and keepouts
QDict<CCctComp> comps = pla.comps();
QDictIterator<CCctComp> itComp(comps);
for(itComp.toFirst();itComp.current();++itComp)
{
CCctComp* pComp = *itComp;
CCctImage* pImage = lib.image(pComp->image());
if(pImage==NULL) continue;
//keepouts
QPtrList<CCctKeepout> keepouts = pImage->keepouts();
while(!keepouts.isEmpty())
{
CCctKeepout* pKeepout = keepouts.take();

addRect(map,pKeepout->layer(),pKeepout->rect()+pComp->placement(),layers);
}
          //pins
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
QDict<CCctPin> pins = pImage->pins();
QDictIterator<CCctPin> itPin(pins);
for(itPin.toFirst();itPin.current();++itPin)
{
CCctPin* pPin = *itPin;
QValueList< QPair<QString, CRect> > ports =
pPin->ports();
while(!ports.isEmpty())
{
QPair<QString, CRect> port =
ports.front();
ports.pop_front();
addRect(map,port.first,port.second +
pComp->placement(),layers);
}
}
}
}

{
//add nets
QDict<CCctNet> nets = rte.nets();
QDictIterator<CCctNet> itNet(nets);
for(itNet.toFirst();itNet.current();++itNet)
{
CCctNet* net = *itNet;
QPtrList<CCctWire> wires = net->wires();
while(!wires.isEmpty())
{
CCctWire* wire = wires.take();
QPtrList<CCctPath> paths = wire->paths();
while(!paths.isEmpty())
{
CCctPath* path = paths.take();
CRect
r(CPoint(path->x1()/rte.res(),path->y1()/rte.res()),

CPoint(path->x2()/rte.res(),path->y2()/rte.res()));
r = r.normalize();
CCctLayer* layer =
str.layer(path->layer());
if(path->x1()!=path->x2())
{
//std::cout << "Rect : " <<
r.left() << " " << r.bottom();
//std::cout << " : " <<
r.right() << " " << r.top() << "\n";
r.bottom(
r.bottom()-layer->width()/2);
r.top(
r.top()+layer->width()/2);

addRect(map,path->layer(),r,layers);
}else if( path->y1()!=path->y2())
{
//std::cout << "Rect : " <<
r.left() << " " << r.bottom();
//std::cout << " : " <<
r.right() << " " << r.top() << "\n";
r.left(
r.left()-layer->width()/2);
r.right(
r.right()+layer->width()/2);

addRect(map,path->layer(),r,layers);
}
//std::cout << "Rect : " << r.left() <<
" " << r.bottom();
//std::cout << " : " << r.right() << " "
<< r.top() << "\n";
//addRect(map,path->layer(),r,layers);
}
}
}
}

int tnumOver = 0;
int tflow = 0;
int tcap = 0;
f << "\t | #Over \t\t T_Flow \t\t T_Cap\n";
f <<
"----------------------------------------------------------------------\n";
for(int i=0;i<numLayers;i++)
{
int numOver = 0;
int flow = 0;
int cap = 0;

int scap = (layers[i]->dir()==VERTICAL)?
int(gCell.width()/layers[i]->pitch()):
int(gCell.height()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
flow += map[i][j][k];
cap += scap;
if(map[i][j][k]>scap) numOver++;
}
}
f << layers[i]->name();
f << " \t | " << numOver << " \t\t " << flow << " \t\t " << cap
<< "\t\t" << "\n";
tnumOver += numOver;
tflow += flow;
tcap += cap;
}
f <<
"----------------------------------------------------------------------\n";
f << "TOTAL";
f << " \t | " << tnumOver << " \t\t " << tflow << " \t\t " << tcap <<
"\n";
f <<
"----------------------------------------------------------------------\n\n\n";

{
//add nets lengths
//QFile FL("len_"+name);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//FL.open(IO_WriteOnly);
//QTextStream fl(&FL);
unsigned long int totLen = 0;
QDict<CCctNet> nets = rte.nets();
QDictIterator<CCctNet> itNet(nets);
for(itNet.toFirst();itNet.current();++itNet)
{
CCctNet* net = *itNet;
QPtrList<CCctWire> wires = net->wires();
//fl << net->name();
double len = 0;
while(!wires.isEmpty())
{
CCctWire* wire = wires.take();
QPtrList<CCctPath> paths = wire->paths();
while(!paths.isEmpty())
{
CCctPath* path = paths.take();
CRect
r(CPoint(path->x1()/rte.res(),path->y1()/rte.res()),

CPoint(path->x2()/rte.res(),path->y2()/rte.res()));
r = r.normalize();

if(str.layer(path->layer())->dir()==VERTICAL)
{
totLen += (int)r.height();
len += (int)r.height();
}
else
{
totLen += (int)r.width();
len += (int)r.width();
}
//addRect(map,path->layer(),r,layers);
}
}
//fl << "\t" << len << "\n";
}
f << "Wire Length \t: " << totLen << "\n";
//FL.close();
}

 std::cout << "GER\n";
for(int i=0;i<numLayers;i++)
{
int scap = (layers[i]->dir()==VERTICAL)?
int(gCell.width()/layers[i]->pitch()):
int(gCell.height()/layers[i]->pitch());
QFile FL(layers[i]->name()+"_"+name);
FL.open(IO_WriteOnly);
QTextStream fl(&FL);
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
//fl << layers[i]->name() << "\t"
fl << j << "\t" << k;
fl << "\t" << map[i][j][k] << "\n";
//if(map[i][j][k]>scap) f << " * ";
fl << "\n";
}
}
FL.close();
}

//
F.close();
for(int i=0;i<numLayers;i++)
{
for(int j=0;j<=numCols;j++)
{
delete[] map[i][j];
}
delete[] map[i];
}
delete[] map;

return true;
}

bool CUtil::conglefdefWrite(const QString name)
{
extern CLef lef;
extern CDef def;
extern CGRoute route;

QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);


QPtrVector<CLefLayer> layers = lef.routingLayers();
int numLayers = layers.count();

CDefGCell gCell = def.gCell();
int numCols = gCell.numCols();
int numRows = gCell.numRows();

// int (*map)[numCols+1][numRows+1] = new
int[numLayers][numCols+1][numRows+1];
int ***map;
map = new int**[numLayers];
for(int i=0;i<numLayers;i++)
{
map[i] = new int*[numCols+1];
for(int j=0;j<=numCols;j++)
{
map[i][j] = new int[numRows+1];
for(int k=0;k<=numRows;k++)
{
map[i][j][k] = 0;
}
}
}

{//add def blocakges
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
QPtrList<CDefBlock> blocks = def.blocks();
while(!blocks.isEmpty())
{
CDefBlock* blk = blocks.take();
if(!blk->hasLayer()) continue;
CLefLayer* layer = lef.layer( blk->layerName() );
if( layer->type()!=ROUTING ) continue;
QValueList<CRect> rects = blk->rectangles();
while(!rects.isEmpty())
{
CRect r =rects.front();rects.pop_front();
addRect(map, blk->layerName(), r, layers);
}
}
}

{//add pdef pins
QDict<CDefPin> defpins = def.pins();
    QDictIterator<CDefPin> defpinIt(defpins);
for(defpinlt.toFirst();defpinIt.current();++defpinIt)
{
CDefPin* pDefPin = *defpinIt;
if(!pDefPin->hasLayer()) continue;
CLefLayer* layer = lef.layer( pDefPin->layer() );
if( layer->type()!=ROUTING ) continue;
double x1,y1,x2,y2;
pDefPin->bounds(&x1,&y1,&x2,&y2);
CRect r = CRect(x1,y1,(x2-x1),(y2-y1));
r = r.orientDefPin(CPoint(pDefPin->placementX(),
pDefPin->placementY()),
pDefPin->orient());
addRect(map, pDefPin->layer(), r, layers );
}
}

{//add comp pins and obstructions
QDict<CDefComp> comps = def.comps();
QDictIterator<CDefComp> compIt(comps);
for(compIt.toFirst();compIt.current();++compIt)
{
CDefComp* pComp = *compIt;
CLefMacro* pMacro = lef.macro(pComp->name());
if(pMacro==NULL) continue;
          //obstructions
QPtrList<CLefObs> obs = pMacro->obs();
QPtrListIterator<CLefObs> obsIt(obs);
for(obsIt.toFirst();obsIt.current();++obsIt)
{
QValueList< QPair<geomType, void*> > items =
(*obsIt)->items();
    QValueListIterator< QPair<geomType, void*> >
itemIt;
QString layerName;
CRect rect;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
             {
             switch((*itemIt).first)
               {
                 case GEOMLAYER:
                 layerName =
*(QString*)(*itemIt).second;
break;
             case GEOMRECT:
                   rect =
*(CRect*)(*itemIt).second;
                   if( pMacro->hasOrigin() )
rect = rect +
CPoint(
pMacro->originX(),

pMacro->originY() );
                   rect = rect.orientLefPin(
CRect(
pComp->placementX()/def.units(),

pComp->placementY()/def.units(),

pMacro->sizeX(),pMacro->sizeY() ),

pComp->placementOrient() );
addRect(map, layerName,
rect*def.units(), layers );
                   break;
                       default: std::cout << "not implemented
obstruction shape\n";
                 }

             }
}
//pins
QDict<CLefPin> pins = pMacro->pins();
QDictIterator<CLefPin> pinIt(pins);
for(pinIt.toFirst();pinIt.current();++pinIt)
{
CLefPin* pPin = *pinIt;
QPtrList<CLefPort> ports = pPin->ports();
QPtrListIterator<CLefPort> portIt(ports);
              for(portIt.toFirst();portIt.current();++portIt)
              {
              QValueList< QPair<geomType, void*> >
items = (*portIt)->items();
                 QValueListIterator< QPair<geomType,
void*> > itemIt;
QString layer;
CRect r;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
                 {
                 switch((*itemIt).first)
                   {
                   case GEOMLAYER :
                   layer =
*(QString*)(*itemIt).second;
                      break;
                  case  GEOMRECT :
                   r =
*(CRect*)(*itemIt).second;
```

```
                        if( pMacro->hasOrigin()
)
r = r +
CPoint(
pMacro->originX(),

pMacro->originY() );
                        r =
r.orientLefPin(
CRect(
pComp->placementX()/def.units(),

pComp->placementY()/def.units(),

pMacro->sizeX(),pMacro->sizeY() ),

pComp->placementOrient() );
addRect(map, layer, r,
layers );
                        break;
        default: std::cout << "not
implemented pin shape\n";
            }
}
        }
}
}
}

{//add special nets
{QDict<CDefNet> nets = def.snets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL ;
CPoint oldpoint,newpoint;
bool firstpoint = true;
double x1,x2,y1,y2,with;
with = -1;
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-with/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+with/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-with/2;
x2 = newpoint.x();
y2 =
newpoint.y()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name(),
r, layers);
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
x1 =
oldpoint.x()-with/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+with/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-with/2;
x2 = newpoint.x();
y2 =
newpoint.y()+with/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name(),
r, layers);
break;}
case PATHWIDTH :
{with = *((int*)item);
break;}
default:break;
}
}
}
}}
//add nets
{QDict<CDefNet> nets = def.nets();
QDictIterator<CDefNet> netIt(nets);
for(netIt.toFirst();netIt.current();++netIt)
{
CDefNet* net = *netIt;
int numPaths = net->numPaths();
for(int i=0;i<numPaths;i++)
{
CLefLayer* oldlayer;
CLefLayer* newlayer = NULL ;
CPoint oldpoint,newpoint;
double x1,x2,y1,y2,with;
with = -1;
bool firstpoint=true;
CDefPath* path = net->path(i);
QValueList< QPair<pathType, void*> > items =
path->items();
QValueListIterator< QPair<pathType, void*> >
itemIt;

for(itemIt=items.begin();itemIt!=items.end();itemIt++)
{
pathType type = (*itemIt).first;
void* item = (*itemIt).second;
switch(type)
{
case PATHLAYER :
{newlayer = lef.layer(
*((QString*)item) );
break;}
case PATHVIA :
{oldlayer = newlayer;
CLefVia* via = lef.via(
*((QString*)item) );
QValueList<QPair<QString,
QValueList<CRect> > > data;
data = via->data();

QValueListIterator<QPair<QString, QValueList<CRect> > > dataIt;

for(dataIt=data.begin();dataIt!=data.end();dataIt++)
{
CLefLayer* canlayer =
lef.layer( (*dataIt).first );
if(
canlayer->type()!=ROUTING ) continue;
if(
canlayer->name()==oldlayer->name() ) continue;
newlayer = canlayer;
}
break;}
case PATHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
x1 =
oldpoint.x()-def.units()*newlayer->width()/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+def.units()*newlayer->width()/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-def.units()*newlayer->width()/2;
x2 = newpoint.x();
y2 =
newpoint.y()+def.units()*newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map, newlayer->name() ,
r, layers);
break;}
case PATHFLUSHPOINT :
{oldpoint = newpoint;
newpoint = *((CPoint*)item);
if(firstpoint)
{firstpoint=false;break;}

if(newlayer->direction()==VERTICAL)
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
x1 =
oldpoint.x()-def.units()*newlayer->width()/2;
y1 = oldpoint.y();
x2 =
newpoint.x()+def.units()*newlayer->width()/2;
y2 = newpoint.y();
}else if(
newlayer->direction()==HORIZONTAL)
{
x1 = oldpoint.x();
y1 =
oldpoint.y()-def.units()*newlayer->width()/2;
x2 = newpoint.x();
y2 =
newpoint.y()+def.units()*newlayer->width()/2;
}else break;
CRect r(x1,y1,x2-x1,y2-y1);
r = r.normalize();
addRect(map,newlayer->name(), r
, layers );
break;}
case PATHWIDTH :
{with = *((int*)item);
break;}
default:break;
}
}
}
}}
}

{//add routed nets
QDict<CNet> nets = route.nets();
QDictIterator<CNet> it(nets);
for(it.toFirst();it.current();++it)
{
CNet* net = *it;
QPtrListIterator<gwire> wIt(*net);
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
QValueList<CTileEdge> edges = wire->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge e = *eIt;
CLayer* layer = e.layer();
int track = e.track();
int edge = e.edge();
int col =(layer->dir()==VERTICAL)? edge
: track;
int row =(layer->dir()==HORIZONTAL)?
edge : track;
if(col>=numCols) continue;
if(row>=numRows) continue;
map[layer->index()][col][row] += 1;

}
}
}

}

int tnumOver = 0;
int tflow = 0;
int tcap = 0;
f << "\t | #Over \t\t T_Flow \t\t T_Cap\n";
f <<
"----------------------------------------------------------------------\n";
for(int i=0;i<numLayers;i++)
{
int numOver = 0;
int flow = 0;
int cap = 0;

int scap = (layers[i]->direction()==VERTICAL)?
int(gCell.xStep()/def.units()/layers[i]->pitch()):
int(gCell.yStep()/def.units()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
flow += map[i][j][k];
cap += scap;
if(map[i][j][k]>scap) numOver++;
}
}
f << layers[i]->name();
f << " \t | " << numOver << " \t\t " << flow << " \t\t " << cap
<< "\n";
tnumOver += numOver;
tflow += flow;
tcap += cap;
}
f <<
"----------------------------------------------------------------------\n";
f << "TOTAL";
f << " \t | " << tnumOver << " \t\t " << tflow << " \t\t " << tcap <<
"\n";
f <<
"----------------------------------------------------------------------\n\n\n";

for(int i=0;i<numLayers;i++)
{
int scap = (layers[i]->direction()==VERTICAL)?
int(gCell.xStep()/def.units()/layers[i]->pitch()):
int(gCell.yStep()/def.units()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
f << layers[i]->name() << " " << j << " " << k;
f << " " << map[i][j][k];
if(map[i][j][k]>scap) f << " * ";f << "\n";
}
}
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
//
F.close();
for(int i=0;i<numLayers;i++)
{
for(int j=0;j<=numCols;j++)
{
delete[] map[i][j];
}
delete[] map[i];
}
delete[] map;

return true;
}


bool CUtil::congcctWrite(const QString name)
{
extern CStr str;
extern CDsn dsn;
extern CPla pla;
extern CLib lib;
extern CNetList netlist;
extern CTim tim;
extern CRte rte;
extern CGRoute route;

QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);

//layers
QDict<CCctLayer> layers_helper = str.layers();
QPtrVector<CCctLayer> layers(layers_helper.count());
QDictIterator<CCctLayer> it(layers_helper);
for(it.toFirst();it.current();++it)
{
CCctLayer* layer = *it;
layers.insert(layer->index(), layer);
}
int numLayers = layers.count();

//gcell
CSize gCell = route.gCell();
CRect dieArea = str.diePoly().boundingRect();
int numCols = (int)ceil(dieArea.width()/gCell.width());
int numRows = (int)ceil(dieArea.height()/gCell.height());

// int (*map)[numCols+1][numRows+1] = new
int[numLayers][numCols+1][numRows+1];
int ***map;
map = new int**[numLayers];
for(int i=0;i<numLayers;i++)
{
map[i] = new int*[numCols+1];
for(int j=0;j<=numCols;j++)
{
map[i][j] = new int[numRows+1];
for(int k=0;k<=numRows;k++)
{
map[i][j][k] = 0;
}
}
}
//add str keepouts
{
QPtrList<CCctKeepout> keepouts = str.keepouts();
while(!keepouts.isEmpty())
{
CCctKeepout* blk = keepouts.take();
addRect(map,blk->layer(),blk->rect(),layers);
}
}
//add wiring
{
QValueList< QPair<QString,CRect> > wires = dsn.wiring();
while(!wires.isEmpty())
{
QPair<QString,CRect> wire = wires.front();
wires.pop_front();
addRect(map,wire.first,wire.second,layers);
}
}
{
//add comp pins and keepouts
QDict<CCctComp> comps = pla.comps();
QDictIterator<CCctComp> itComp(comps);
for(itComp.toFirst();itComp.current();++itComp)
{
CCctComp* pComp = *itComp;
CCctImage* pImage = lib.image(pComp->image());
if(pImage==NULL) continue;
//keepouts
QPtrList<CCctKeepout> keepouts = pImage->keepouts();
while(!keepouts.isEmpty())
{
CCctKeepout* pKeepout = keepouts.take();

addRect(map,pKeepout->layer(),pKeepout->rect()+pComp->placement(),layers);
}
        //pins
QDict<CCctPin> pins = pImage->pins();
QDictIterator<CCctPin> itPin(pins);
for(itPin.toFirst();itPin.current();++itPin)
{
CCctPin* pPin = *itPin;
QValueList< QPair<QString, CRect> > ports =
pPin->ports();
while(!ports.isEmpty())
{
QPair<QString, CRect> port =
ports.front();
ports.pop_front();
addRect(map,port.first,port.second +
pComp->placement(),layers);
}
```

```
}
}
}

{
//add nets
QDict<CCctNet> nets = rte.nets();
QDictIterator<CCctNet> itNet(nets);
for(itNet.toFirst();itNet.current();++itNet)
{
CCctNet* net = *itNet;
QPtrList<CCctWire> wires = net->wires();
while(!wires.isEmpty())
{
CCctWire* wire = wires.take();
QPtrList<CCctPath> paths = wire->paths();
while(!paths.isEmpty())
{
CCctPath* path = paths.take();
CRect
r(CPoint(path->x1()/rte.res(),path->y1()/rte.res()),

CPoint(path->x2()/rte.res(),path->y2()/rte.res()));
r = r.normalize();
addRect(map,path->layer(),r,layers);
}
}
}
}

{//add routed nets
QDict<CNet> nets = route.nets();
QDictIterator<CNet> it(nets);
for(it.toFirst();it.current();++it)
{
CNet* net = *it;
QPtrListIterator<gwire> wIt(*net);
for(wIt.toFirst();wIt.current();++wIt)
{
gwire* wire = *wIt;
QValueList<CTileEdge> edges = wire->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge e = *eIt;
CLayer* layer = e.layer();
int track = e.track();
int edge = e.edge();
int col =(layer->dir()==VERTICAL)? edge
: track;
int row =(layer->dir()==HORIZONTAL)?
edge : track;
if(col>=numCols) continue;
if(row>=numRows) continue;
map[layer->index()][col][row] += 1;

}
}
}
```

```
}

int tnumOver = 0;
int tflow = 0;
int tcap = 0;
f << "\t | #Over \t\t T_Flow \t\t T_Cap\n";
f <<
"----------------------------------------------------------------------\n";
for(int i=0;i<numLayers;i++)
{
int numOver = 0;
int flow = 0;
int cap = 0;

int scap = (layers[i]->dir()==VERTICAL)?
int(gCell.width()/layers[i]->pitch()):
int(gCell.height()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
flow += map[i][j][k];
cap += scap;
if(map[i][j][k]>scap) numOver++;
}
}
f << layers[i]->name();
f << " \t | " << numOver << " \t\t " << flow << " \t\t " << cap
<< "\n";
tnumOver += numOver;
tflow += flow;
tcap += cap;
}
f <<
"----------------------------------------------------------------------\n";
f << "TOTAL";
f << " \t | " << tnumOver << " \t\t " << tflow << " \t\t " << tcap <<
"\n";
f <<
"----------------------------------------------------------------------\n\n\n";

for(int i=0;i<numLayers;i++)
{
int scap = (layers[i]->dir()==VERTICAL)?
int(gCell.width()/layers[i]->pitch()):
int(gCell.height()/layers[i]->pitch());
for(int j=0;j<=numCols;j++)
{
for(int k=0;k<=numRows;k++)
{
//f << layers[i]->name() << " " ;
f << j << " " << k;
f << " " << map[i][j][k];
//if(map[i][j][k]>scap) f << " * ";
f << "\n";
}
}
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
//
F.close();
for(int i=0;i<numLayers;i++)
{
for(int j=0;j<=numCols;j++)
{
delete[] map[i][j];
}
delete[] map[i];
}
delete[] map;

return true;

}

bool CUtil::lengthWrite(const QString name)
{

extern CGRoute route;

QFile F(name);
F.open(IO_WriteOnly);
QTextStream f(&F);

QDict<CNet> nets = route.nets();
double len = 0;
QDictIterator<CNet> it(nets);
for(it.toFirst();it.current();++it) len += (*it)->length();
f << "Total Length " << len << "\n";


F.close();
return true;

}

bool CUtil::cctTimWrite(const QString name)
{
extern CGRoute route;
extern CRte rte;

double maxDV = -INF;
double totDV = 0;
double totLen = 0;
int numDV=0;
int numNet=0;
int numBuf=0;
QString maxDVNet;

QFile F(name);
F.open(IO_WriteOnly | IO_Raw);
QTextStream f(&F);
//QFile FL("net_"+name);
//FL.open(IO_WriteOnly | IO_Raw);
//QTextStream fl(&FL);
```

```
//route.randtim();

QDict<CNet> nets = route.nets();
QDictIterator<CNet> nIt(nets);
std::cout << "Here\n";
for(nIt.toFirst();nIt.current();++nIt)
{
        std::cout << "tehre\n";
CNet* pNet = *nIt;
//f << "net " << QString(*pNet) << "\n";
std::cout << "net " << QString(*pNet) << "\n";
CCctNet* cctNet = rte.net(QString(*pNet));
if(cctNet==0) continue;
f << "net from rte " << cctNet->name() << "\n";
//fl << cctNet->name();
QPtrList<CCctPath> P;
        QPtrList<CCctWire> wires = cctNet->wires();
QPtrListIterator<CCctWire> wIt(wires);
for(wIt.toFirst();wIt.current();++wIt)
{
QPtrList<CCctPath> paths = (*wIt)->paths();
        QPtrListIterator<CCctPath> pIt(paths);
for(pIt.toFirst();pIt.current();++pIt)
P.append( *pIt );
}
begin:
QPtrListIterator<CCctPath> it1(P);
QPtrListIterator<CCctPath> it2(P);
 f << "original paths\n";
for(it1.toFirst();it1.current();++it1)
{
CCctPath* p = *it1;
f << "gpath (layer " << p->layer() << " ) ";
f << p->pin1() << "(" << p->x1() << " " << p->y1() << ")
: ";
f << p->pin2() << "(" << p->x2() << " " << p->y2() <<
")\n";
}
for(it1.toFirst();it1.current();++it1)
{
CCctPath* p1 = *it1;
it2 = it1;++it2;
for(;it2.current();++it2)
{
CCctPath* p2 = *it2;
f << "two path to compare\n";
f << "p1 gpath (layer " << p1->layer() << " ) ";
f << p1->pin1() << "(" << p1->x1() << " " <<
p1->y1() << ") : ";
f << p1->pin2() << "(" << p1->x2() << " " <<
p1->y2() << ")\n";
f << "p2 gpath (layer " << p2->layer() << " ) ";
f << p2->pin1() << "(" << p2->x1() << " " <<
p2->y1() << ") : ";
f << p2->pin2() << "(" << p2->x2() << " " <<
p2->y2() << ")\n";
if(p1==p2) continue;
f << "p1==p2\n";
if(p1->layer()!=p2->layer()) continue;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
f << "p1->layer()!=p2->layer()\n";
if(!p1->intersects(*p2)) continue;
f << "!p1->intersects(*p2)\n";
// if(p1->width()>1 && p2->widht()>1 &&
p1->intersect(*p2).width()==1 ) continue;
    //      if(p1->height()>1 && p2->height()>1 &&
p1->intesect(*p2).height()==1 ) continue;
P.remove(p1);
P.remove(p2);
QPtrList<CCctPath> nonInPaths =
CCctPath::nonInPaths(*p1,*p2);
QPtrListIterator<CCctPath> it(nonInPaths);
if(nonInPaths.isEmpty()) f << "ERROR\n";
assert(!nonInPaths.isEmpty());
f << "two paths compared\n";
for(it.toFirst();it.current();++it)
{
f << "p gpath (layer " << (*it)->layer()
<< " ) ";
f << (*it)->pin1() << "(" << (*it)->x1()
<< " " << (*it)->y1() << ") : ";
f << (*it)->pin2() << "(" << (*it)->x2()
<< " " << (*it)->y2() << ")\n";
P.append( *it );
}
goto begin;
}
}
f << "nonintersecting paths\n";
for(it1.toFirst();it1.current();++it1)
{
CCctPath* p = *it1;
f << "gpath (layer " << p->layer() << " ) ";
f << p->pin1() << "(" << p->x1() << " " << p->y1() << ")
: ";
f << p->pin2() << "(" << p->x2() << " " << p->y2() <<
")\n";
}

CSource* rootN = (CSource*)pNet->root();
QString root = (QString)(*rootN);
int x,y;
QPtrList<CCctPath> T;
QPtrListIterator<CCctPath> itP(P);
for(itP.toFirst();itP.current();++itP)
{
CCctPath* path = *itP;
//std::cout << root << "==" << path->pin1() << " | " <<
path->pin2() << "\n";
if(path->pin1()==root)
{
x = path->x1();
y = path->y1();
break;
}else if(path->pin2()==root)
{
x = path->x2();
y = path->y2();
break;
```

```
}
}
for(itP.toFirst();itP.current();++itP)
{
CCctPath* path = *itP;
if( (x==path->x1() && y==path->y1()) ||
(x==path->x2() && y==path->y2()))
T.append(path);
}
QPtrListIterator<CCctPath> itT(T);
for(itT.toFirst();itT.current();++itT)
{
CCctPath* path = *itT;
P.remove(path);
}
pNet->buildTree(rootN, root,x,y,T,&P, f);
//std::cout << "Inserting Buffers \n";
route.awe(pNet);
//pNet->dump("");
//while(route.insertBuffer(pNet)) std::cout << numBuf++ << "\n";
//std::cout << "Inserted Buffers \n";
//if(numBuf>0) exit(1);
// route.awe(pNet);
//pNet->dump("");
// std::cout << "\t Delay Violation = " << pNet->maxDV() << "\n";
// f1 << "\t" << pNet->maxDV() << "\t" << pNet->length();
//f1 << "\t" << pNet->numBuffers() << "\n";
// numNet++;
//double DV = pNet->maxDV();
//if(DV>0) { numDV++;totDV+=DV;}
// if(DV>maxDV) {maxDV = DV;maxDVNet = (QString)*pNet;}
// totLen += pNet->length();
//pNet->dump("");
}


//FL.close();
F.close();
//std::cout << "Number of Nets = " << numNet << "\n";
//std::cout << "Number of Delay Violations = " << numDV << "\n";
//std::cout << "Total Delay Violation = " << totDV << "\n";
//std::cout << "Maximum Delay Violation = " << maxDV << " (" << maxDVNet
<< ")\n";
//std::cout << "Number of Buffers = " << numBuf << "\n";
//std::cout << "Total Length = " << totLen << "\n";
return true;
}


void CUtil::addRect(int ***map, QString layerName, CRect r,
QPtrVector<CLefLayer> layers)
{
// std::cout << layerName << " ";
// std::cout << r.left() << " " << r.right() << " : " << r.bottom() << " "
<< r.top() << "\n";
extern CLef lef;
extern CDef def;
}
int numLayers = layers.count();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CDefGCell gCell = def.gCell();

CLefLayer* layer = lef.layer( layerName );
if( layer->type()!=ROUTING ) return;


r = r & def.dieArea();

int mnt = 0;

int indx = -1;
for(int i=0;i<numLayers;i++)
if(layers[i]->name()==layer->name()) {indx = i;break;}
int xCap = (layer->direction()==VERTICAL)?
int(gCell.xStep()/def.units()/layer->pitch()):0;
int yCap = (layer->direction()==HORIZONTAL)?
int(gCell.yStep()/def.units()/layer->pitch()):0;
// std::cout << "xCap = " << xCap << " yCap = " << yCap << "\n";

int colS = (int)floor((r.left()-gCell.x())/gCell.xStep());
int colE = (int)floor((r.right()-gCell.x())/gCell.xStep());
int rowS = (int)floor((r.bottom()-gCell.y())/gCell.yStep());
int rowE = (int)floor((r.top()-gCell.y())/gCell.yStep());
// std::cout << "bnd " << colS << " " << colE << " " << rowS << " " << rowE
<< "\n";
for(int
col=(layer->direction()==HORIZONTAL)?colS+1:colS;col<=colE;col++)
{
double left = col * gCell.xStep() + gCell.x();
double right = left;
right += (layer->direction()==VERTICAL)?gCell.xStep():0;
left = ::max(left, r.left());
right = ::min(right, r.right());
int blkcol = (int)ceil(xCap*(right-left)/gCell.xStep());
for(int
row=(layer->direction()==VERTICAL)?rowS+1:rowS;row<=rowE;row++)
{
double bottom = row * gCell.yStep() + gCell.y();
double top = bottom;
top += (layer->direction()==HORIZONTAL)?gCell.yStep():0;
bottom = ::max(bottom, r.bottom());
top = ::min(top, r.top());
int blkrow = (int)ceil(yCap*(top-bottom)/gCell.yStep());
map[indx][col][row] += blkcol + blkrow;
mnt += blkcol + blkrow;
// std::cout << "left right : bottom top ";
// std::cout << left << " " << right << " : " << bottom <<
" " << top << "\n";
// std::cout << "col row : blkcol blkrow ";
// std::cout << col << " " << row << " : " << blkcol << " "
<< blkrow << "\n";
}
}
// std::cout << "Added " << mnt << "\n";

}

void CUtil::addRect(int ***map, QString layerName, CRect r,
QPtrVector<CCctLayer> layers)
{
// std::cout << layerName << " ";
// std::cout << r.left() << " " << r.right() << " : " << r.bottom() << " "
<< r.top() << "\n";
extern CGRoute route;
extern CStr str;

int numLayers = layers.count();
CSize gCell = route.gCell();
CRect dieArea = str.diePoly().boundingRect();

CCctLayer* layer = str.layer( layerName );

r = r & dieArea;

int mnt = 0;

int indx = -1;
for(int i=0;i<numLayers;i++)
if(layers[i]->name()==layer->name()) {indx = i;break;}
int xCap = (layer->dir()==VERTICAL)?
int(gCell.width()/layer->pitch()):0;
int yCap = (layer->dir()==HORIZONTAL)?
int(gCell.height()/layer->pitch()):0;
// std::cout << "xCap = " << xCap << " yCap = " << yCap << "\n";

int colS = (int)floor((r.left()-dieArea.left())/gCell.width());
int colE = (int)floor((r.right()-dieArea.left())/gCell.width());
int rowS = (int)floor((r.bottom()-dieArea.bottom())/gCell.height());
int rowE = (int)floor((r.top()-dieArea.bottom())/gCell.height());
//std::cout << "bnd " << colS << " " << colE << " " << rowS << " " <<
rowE << "\n";
for(int col=(layer->dir()==HORIZONTAL)?colS+1:colS;col<=colE;col++)
{
double left = col * gCell.width() + dieArea.left();
double right = left;
right += (layer->dir()==VERTICAL)?gCell.width():0;
left = ::max(left, r.left());
right = ::min(right, r.right());
int blkcol = (int)ceil(xCap*(right-left)/gCell.width());
for(int
row=(layer->dir()==VERTICAL)?rowS+1:rowS;row<=rowE;row++)
{
double bottom = row * gCell.height() + dieArea.bottom();
double top = bottom;
top += (layer->dir()==HORIZONTAL)?gCell.height():0;
bottom = ::max(bottom, r.bottom());
top = ::min(top, r.top());
int blkrow =
(int)ceil(yCap*(top-bottom)/gCell.height());
map[indx][col][row] += blkcol + blkrow;
mnt += blkcol + blkrow;
//std::cout << "left right : bottom top ";
//std::cout << left << " " << right << " : " << bottom
<< " " << top << "\n";
//std::cout << "col row : blkcol blkrow ";
//std::cout << col << " " << row << " : " << blkcol << "
" << blkrow << "\n";
}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
}
// std::cout << "Added " << mnt << "\n";

}
#include "global.h"

layerType layer( const QString layerType )
{
if( layerType=="CUT" ) return CUT;
if( layerType=="MASTERSLICE" ) return MASTERSLICE;
if( layerType=="OVERLAP" ) return OVERLAP;
if( layerType=="ROUTING" ) return ROUTING;
return UNKNOWNLAYERTYPE;
}

QString layer(const layerType layer)
{
if( layer==CUT ) return "CUT";
if( layer==MASTERSLICE ) return "MASTERSLICE";
if( layer==OVERLAP ) return "OVERLAP";
if( layer==ROUTING ) return "ROUTING";
return "UNKNOWLAYERTPE";
}

layerDirectionType layerDirection(const QString layerDirection)
{
if( layerDirection=="VERTICAL" ) return VERTICAL;
if( layerDirection=="HORIZONTAL" ) return HORIZONTAL;
return UNKNOWNLAYERDIRECTION;
}

QString layerDirection(const layerDirectionType layerDirection)
{
if( layerDirection==VERTICAL ) return "VERTICAL";
if( layerDirection==HORIZONTAL ) return "HORIZONTAL";
return "UNKNOWNLAYERDIR";
}

geomType geomType(const int geomType)
{
if(geomType==lefiGeomLayerE) return GEOMLAYER;
if(geomType==lefiGeomRectE) return GEOMRECT;
return UNKNOWNGEOM;
}

pinDirectionType pinDirection(const QString pinDir)
{
if( pinDir=="INPUT" ) return IN;
if( pinDir=="OUTPUT" ) return OUT;
if( pinDir=="INOUT" ) return INOUT;
if( pinDir=="FEEDTHRU" ) return FEEDTHRU;
return UNKNOWNPINDIR;
}

QString pinDirection(const pinDirectionType pinDir)
{
  if( pinDir==IN ) return "INPUT";
if( pinDir==OUT ) return "OUTPUT";
if( pinDir==INOUT ) return "INOUT";
if( pinDir==FEEDTHRU ) return "FEEDTHRU";
return "UNKNOWNPINDIR";
}

useType use(const QString use)
{
if( use=="ANALOG" ) return ANALOG;
if( use=="CLOCK" ) return CLOCK;
if( use=="GROUND" ) return GROUND;
if( use=="POWER" ) return POWER;
if( use=="RESET" ) return RESET;
if( use=="SCAN" ) return SCAN;
if( use=="SIGNAL" ) return SIGNAL;
if( use=="TIEOFF" ) return TIEOFF;
return UNKNOWNUSE;
}

QString use(const useType use)
{
if( use==ANALOG ) return "ANALOG";
if( use==CLOCK ) return "CLOCK";
if( use==GROUND ) return "GROUND";
if( use==POWER ) return "POWER";
if( use==RESET ) return "RESET";
if( use==SCAN ) return "SCAN";
if( use==SIGNAL ) return "SIGNAL";
if( use==TIEOFF ) return "TIEOFF";
return "UNKNOWNUSE";
}

placementStatusType placementStatus(const int placementStatus)
{
switch(placementStatus)
{
case DEFI_COMPONENT_UNPLACED : return UNPLACED;
    case DEFI_COMPONENT_PLACED : return PLACED;
case DEFI_COMPONENT_FIXED : return FIXED;
case DEFI_COMPONENT_COVER : return COVER;
}
return UNKNOWNPLACEMENTSTATUS;
}

QString  placementStatus(const placementStatusType placementStatus )
{
switch(placementStatus)
{
case UNPLACED : return "UNPLACED";
case PLACED : return "PLACED";
case FIXED : return "FIXED";
case COVER : return "COVER";
default : return "UNKNOWNPLACEMENTSTATUS";
}
return "UNKNOWNPLACEMENTSTATUS";
}
placementOrientType placementOrient(const int placementOrient)
{
switch(placementOrient)
{
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
case 0 : return N;
   case 1 : return W;
case 2 : return S;
case 3 : return E;
case 4 : return FN;
case 5 : return FW;
case 6 : return FS;
case 7 : return FE;
}
return UNKNOWNORIENT;
}

int placementOrient(const placementOrientType placementOrient )
{
switch(placementOrient)
{
case N : return 0;
   case W : return 1;
case S : return 2;
case E : return 3;
case FN : return 4;
case FW : return 5;
case FS : return 6;
case FE : return 7;
default : return UNKNOWNORIENT;
}
return UNKNOWNORIENT;
}

pathType path(const int type)
{
if(type==DEFIPATH_DONE) return PATHDONE;
if(type==DEFIPATH_LAYER) return PATHLAYER;
if(type==DEFIPATH_VIA) return PATHVIA;
if(type==DEFIPATH_VIAROTATION) return PATHVIAROTATION;
if(type==DEFIPATH_WIDTH) return PATHWIDTH;
if(type==DEFIPATH_POINT) return PATHPOINT;
if(type==DEFIPATH_FLUSHPOINT) return PATHFLUSHPOINT;
if(type==DEFIPATH_TAPER) return PATHTAPER;
if(type==DEFIPATH_SHAPE) return PATHSHAPE;
if(type==DEFIPATH_TAPERRULE) return PATHTAPERRULE;
if(type==DEFIPATH_VIADATA) return PATHVIADATA;
return UNKNOWNPATH;
}

QString pathStr(const pathType type)
{
switch(type)
{
case PATHDONE : return "PATHDONE";
case PATHLAYER : return "PATHLAYER";
case PATHVIA : return "PATHVIA";
case PATHVIAROTATION : return "PATHVIAROTATION";
case PATHWIDTH : return "PATHWIDTH";
case PATHPOINT : return "PATHPOINT";
case PATHFLUSHPOINT : return "PATHFLUSHPOINT";
case PATHTAPER : return "PATHTAPER";
case PATHSHAPE : return "PATHSHAPE";
case PATHTAPERRULE: return "PATHTAPERRULE";
case PATHVIADATA : return "PATHVIADATA";
case UNKNOWNPATH : return "UNKNOWNPATH";
default: return "UNKNOWNPATH";
}
return "UNKNOWNPATH";
}

wireType wire( const QString type )
{
if( type=="COVER" ) return WIRECOVER;
if( type=="FIXED" ) return WIREFIXED;
if( type=="ROUTED" ) return WIREROUTED;
if( type=="NOSHIELD" ) return WIRENOSHIELD;
if( type=="UNKNOWNWIRETYPE" ) return UNKNOWNWIRETYPE;
return UNKNOWNWIRETYPE;
}

QString wire(const wireType type)
{
if( type==WIRECOVER ) return "COVER";
if( type==WIREFIXED ) return "FIXED";
if( type==WIREROUTED ) return "ROUTED";
if( type==WIRENOSHIELD ) return "NOSHIELD";
if( type==UNKNOWNWIRETYPE ) return "UNKNOWNWIRETYPE";
return "UNKNOWNWIRETYPE";
}

QString nodeTypeStr(const nodeType type)
{      enum nodeType {DRIVER,ROOT,SINK,BUFFER,SUBROOT,STEINER,AWE};
switch(type)
{
case DRIVER : return "DRIVER";
case ROOT : return "ROOT";
case SINK : return "SINK";
case BUFFER : return "BUFFER";
case SUBROOT : return "SUBROOT";
case STEINER : return "STEINER";
case AWE : return "AWE";
default : return "UNKNOWN";
}
return "UNKNOWN";
}

//clips r1 against r2,
//r2 stays same clipped portions of the r1 is returned in the list
//clipped rectangles doesn't touch the main rectangle
//inp : r1, rectangle to clip
//inp : r2, rectanngel to clip against
//out : list of clipped portions of the r1
QValueList<QRect> clipRect(QRect r1, const QRect r2)
{
QValueList<QRect> rects;
QRect rect;
//if they don't intersecs, noting to clip r1 is the result
//if r2 is inside r1, noting to clip r1 is the result
if( !r1.intersects(r2) || r1.contains(r2) ) {rects<<r1;return rects;}
//if r1 is inside r2, no result, empty list returned
if( r1.bottom() > r2.bottom() )
```

```
{
rect = r1;
rect.setTop( r2.bottom()+1 );
r1.setBottom( r2.bottom() );
rects << rect;
}
if( r1.top() < r2.top() )
{
rect = r1;
rect.setBottom( r2.top()-1 );
r1.setTop( r2.top() );
rects << rect;
}
if( r1.left() < r2.left() )
{
rect = r1;
rect.setRight( r2.left()-1 );
r1.setLeft( r2.left() );
rects << rect;
}
if( r1.right() > r2.right() )
{
rect = r1;
rect.setLeft( r2.right()+1 );
r1.setRight( r2.right() );
rects << rect;
}
return  rects;
}
#include "gpath.h"
#include "clayer.h"

gpath::gpath(const CLayerPair layer, const QRect from, const QRect to, const
bool routed)
:QRect(from | to)
,m_layer(layer)
,m_from(from)
,m_to(to)
,m_routed(routed)
{
}

gpath::gpath(const gpath& path)
:QRect((QRect)path)
,m_layer(path.m_layer)
,m_from(path.m_from)
,m_to(path.m_to)
,m_routed(path.m_routed)
{
}

gpath::gpath()
:QRect()
,m_layer(CLayerPair())
,m_from(QRect())
,m_to(QRect())
,m_routed(false)
{
}

gpath::~gpath()
{
}

QValueList<CTile> gpath::tiles(QSize tileSize) const
{
QValueList<CTile> tiles;
CLayer* vl = m_layer.vl();
CLayer* hl = m_layer.hl();
int x1 = (int)floor(m_from.center().x()/tileSize.width());
int y1 = (int)floor(m_from.center().y()/tileSize.height());
int x2 = (int)floor(m_to.center().x()/tileSize.width());
int y2 = (int)floor(m_to.center().y()/tileSize.height());
if(x1!=x2) //horizontal path
{
assert(hl!=NULL);
CValueList<int> list(x1,x2,true,true);
QValueListIterator<int> it;
for(it=list.begin();it!=list.end();it++) tiles.append(
CTile(hl,*it,y1,tileSize));
}
if(x1!=x2 && y1!=y2)//there is bend
{
assert(hl!=NULL && vl!=NULL);
CPtrList<CLayer> list(hl,vl,false,false);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it) tiles.append(
CTile(*it,x2,y1,tileSize));
}
if(y1!=y2) //vertical path
{
assert(vl!=NULL);
CValueList<int> list(y1,y2,true,true);
QValueListIterator<int> it;
for(it=list.begin();it!=list.end();it++) tiles.append(
CTile(vl,x2,*it,tileSize));
}
if(x1==x2 && y1==y2)//there is pad
{
CPtrList<CLayer> list(m_layer.first,m_layer.second,true,true);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it) tiles.append(
CTile(*it,x1,y1,tileSize));
}
return tiles;
}

QValueList<CTileEdge> gpath::edges(const QSize tileSize) const
{
QValueList<CTileEdge> edges;
CLayer* vl = m_layer.vl();
CLayer* hl = m_layer.hl();
int x1 = (int)floor(m_from.center().x()/tileSize.width());
int y1 = (int)floor(m_from.center().y()/tileSize.height());
int x2 = (int)floor(m_to.center().x()/tileSize.width());
int y2 = (int)floor(m_to.center().y()/tileSize.height());
if(x1!=x2) //horizontal path
{
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
assert(hl!=NULL);
CValueList<int> list(x1,x2);
QValueListIterator<int> it;
for(it=list.begin();it!=list.end();it++) edges.append(
CTileEdge(hl,*it,y1,tileSize));
}
if(y1!=y2) //vertical path
{
assert(vl!=NULL);
CValueList<int> list(y1,y2);
QValueListIterator<int> it;
for(it=list.begin();it!=list.end();it++) edges.append(
CTileEdge(vl,*it,x2,tileSize));
}
return edges;
}

void gpath::incTileFlows(const QSize tileSize) const
{
QValueList<CTileEdge> edges = this->edges(tileSize);
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->incTileFlow((*eIt).track(),(*eIt).edge());
}
void gpath::decTileFlows(const QSize tileSize) const
 {
QValueList<CTileEdge> edges = this->edges(tileSize);
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->decTileFlow((*eIt).track(),(*eIt).edge());
}
void gpath::incCellFlows() const
{
if(!m_routed) return;
QValueList<CTileEdge> edges = this->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->incCellFlow((*eIt).track(),(*eIt).edge());
}

void gpath::decCellFlows() const
{
if(!m_routed) return;
QValueList<CTileEdge> edges = this->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->decCellFlow((*eIt).track(),(*eIt).edge());
}

bool gpath::cutsAndBounds(CLayer* layer,const int track,int& lb,int& ub) const
{
if(layer==m_layer.vl() || layer==m_layer.hl())
{
if(layer->dir()==VERTICAL && track>top() && track<=bottom() )
{
lb = left();ub = right();return true;
}else if(layer->dir()==HORIZONTAL && track>left() &&
track<=right() )
{
     lb = top(); ub = bottom(); return true;
}else return false;
}else return false;
}

void gpath::rteWrite(QTextStream& f)
{
f << "\t(gpath";
int x1 = m_from.center().x();
int y1 = m_from.center().y();
int x2 = m_to.center().x();
int y2 = m_to.center().y();
f << " (layer";
double width = 0;
CSize gCell = m_layer.ll()->gCell();
CRect dieArea = m_layer.ll()->dieArea();
if(x1!=x2) {f << " " << m_layer.hl()->name();width = ::
max(width,gCell.height()); }
if(y1!=y2) {f << " " << m_layer.vl()->name();width = ::
max(width,gCell.width()); }
if(x1==x2 && y1==y2)
{
if(m_layer.ll()!=NULL )
{
f << " " << m_layer.ll()->name();
if(m_layer.ll()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ll()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
if(m_layer.ul()!=NULL && m_layer.ul()!=m_layer.ll())
{
f << " " << m_layer.ul()->name();
if(m_layer.ul()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ul()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
}
f << ") " << (int)(1000*width);
//if(x1==x2-1 || x1==x2+1) x1=x2;
//if(y1==y2-1 || y1==y2+1) y1=y2;

double m1 = 1000.0*(x1*gCell.width() + gCell.width()/2.0 +
dieArea.left());
double n1 = 1000.0*(y1*gCell.height() + gCell.height()/2.0 +
dieArea.bottom());
double m2 = 1000.0*(x2*gCell.width() + gCell.width()/2.0 +
dieArea.left());
double n2 = 1000.0*(y2*gCell.height() + gCell.height()/2.0 +
dieArea.bottom());

//if(m1==m2-1 || m1==m2+1) m1=m2;
//if(n1==n2-1 || n1==n2+1) n1=n2;

f << " " << 10*(int)nearbyint(m1/10) << " " << 10*(int)nearbyint(n1/10);
f << " " << 10*(int)nearbyint(m2/10) << " " << 10*(int)nearbyint(n2/10);
f << ")\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// f << m_routed << "\n";
// f << m_from.left() << " " << m_from.top() << " " <<  m_from.right() << "
" <<  m_from.bottom() << "\n";
// f << m_to.left() << " " << m_to.top() << " " <<  m_to.right() << " " <<
m_to.bottom() << "\n";
}

void gpath::grteWrite(QTextStream& f)
{
f << "\t(gpath";
int x1 = m_from.center().x();
int y1 = m_from.center().y();
int x2 = m_to.center().x();
int y2 = m_to.center().y();
f << " (layer";
double width = 0;
CSize gCell = m_layer.ll()->gCell();
CRect dieArea = m_layer.ll()->dieArea();
if(x1!=x2) {f << " " << m_layer.hl()->name();width = ::
max(width,gCell.height()); }
if(y1!=y2) {f << " " << m_layer.vl()->name();width = ::
max(width,gCell.width()); }
if(x1==x2 && y1==y2)
{
if(m_layer.ll()!=NULL )
{
f << " " << m_layer.ll()->name();
if(m_layer.ll()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ll()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
if(m_layer.ul()!=NULL && m_layer.ul()!=m_layer.ll())
{
f << " " << m_layer.ul()->name();
if(m_layer.ul()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ul()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
}
f << ") ";// << (int)(1000*width);

f << " " << (int)x1 << " " << (int)y1;
f << " " << (int)x2 << " " << (int)y2;
f << ") ";
// f << m_routed << "\n";
// f << m_from.left() << " " << m_from.top() << " " <<  m_from.right() << "
" <<  m_from.bottom() << "\n";
// f << m_to.left() << " " << m_to.top() << " " <<  m_to.right() << " " <<
m_to.bottom() << "\n";

QValueList<CTileEdge> edges = this->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{
CTileEdge edge = *eIt;
CLayer* l = edge.layer();
int t = edge.track();
int e = edge.edge();
if(l->cellOverflows(t,e))
{
f << " (*" << l->name() << " " << t << " " << e << ")";
}
}
f << "\n";
}

void gpath::paint(QPainter* p)
{
int x1 = m_from.center().x();
int y1 = m_from.center().y();
int x2 = m_to.center().x();
int y2 = m_to.center().y();
double width = 0;
CSize gCell = m_layer.ll()->gCell();
CRect dieArea = m_layer.ll()->dieArea();
if(x1!=x2)
{
if(m_layer.hl()->name()=="METAL3") p->setPen("black");
else if(m_layer.hl()->name()=="METAL4") p->setPen("yellow");
else if(m_layer.hl()->name()=="METAL5") p->setPen("red");
else if(m_layer.hl()->name()=="METAL6") p->setPen("green");
else if(m_layer.hl()->name()=="METAL7") p->setPen("blue");
}
if(y1!=y2)
{
if(m_layer.vl()->name()=="METAL3") p->setPen("black");
else if(m_layer.vl()->name()=="METAL4") p->setPen("yellow");
else if(m_layer.vl()->name()=="METAL5") p->setPen("red");
else if(m_layer.vl()->name()=="METAL6") p->setPen("green");
else if(m_layer.vl()->name()=="METAL7") p->setPen("blue");
}
if(x1==x2 && y1==y2)
{
if(m_layer.ll()!=NULL )
{
if(m_layer.ll()->name()=="METAL3") p->setPen("black");
else if(m_layer.ll()->name()=="METAL4")
p->setPen("yellow");
else if(m_layer.ll()->name()=="METAL5")
p->setPen("red");
else if(m_layer.ll()->name()=="METAL6")
p->setPen("green");
else if(m_layer.ll()->name()=="METAL7")
p->setPen("blue");
if(m_layer.ll()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ll()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
if(m_layer.ul()!=NULL && m_layer.ul()!=m_layer.ll())
{
if(m_layer.ul()->name()=="METAL3") p->setPen("black");
else if(m_layer.ul()->name()=="METAL4")
p->setPen("yellow");
else if(m_layer.ul()->name()=="METAL5")
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
p->setPen("red");
else if(m_layer.ul()->name()=="METAL6")
p->setPen("green");
else if(m_layer.ul()->name()=="METAL7")
p->setPen("blue");
if(m_layer.ul()->dir()==VERTICAL ) width = ::
max(width,gCell.width());
else if(m_layer.ul()->dir()==HORIZONTAL ) width = ::
max(width,gCell.height());
}
}
p->drawLine((int)x1*10+5,(int)y1*10+5,(int)x2*10+5,(int)y2*10+5);
}
```

```
#include "gwire.h"
#include "gpath.h"
#include "cnode.h"
#include "clayer.h"

gwire::gwire(CNode* from, CNode* to)
:m_from(from)
,m_to(to)
,m_routed(false)
{
setAutoDelete(true);
}

gwire::gwire(const gwire& wire)
:QPtrList<gpath>::QPtrList()
,m_from(wire.from())
,m_to(wire.to())
,m_routed(wire.routed())
{
setAutoDelete(true);
QPtrListIterator<gpath> pIt(wire);
for(pIt.toFirst();pIt.current();++pIt)
append( new gpath(**pIt) );
}

gwire::gwire()
:m_from(NULL)
,m_to(NULL)
,m_routed(false)
{
setAutoDelete(true);
}
```

```
gwire::~gwire()
{
clear();
}

//void gwire::append(gpath* path)
//{
// if(!isEmpty() && !last()->layer().isNull() && !path->layer().isNull())
// {
// QRect r = last()->to() & path->from();
// if(*last()->layer().ul()<*path->layer().ll())
// {
// CPtrList<CLayer>
list(last()->layer().ul(),path->layer().ll(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->append( new gpath(CLayerPair(*it,NULL),r,r));
// }if(*last()->layer().ll()>*path->layer().ul())
// {
// CPtrList<CLayer>
list(last()->layer().ll(),path->layer().ul(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->append( new gpath(CLayerPair(*it,NULL),r,r));
// }
// }
// ((QPtrList<gpath>*)this)->append( path );
//}

//void gwire::prepend(gpath* path)
//{
// if(!isEmpty() && !first()->layer().isNull() && !path->layer().isNull())
// {
// QRect r = first()->from() & path->to();
// if(*first()->layer().ul()<*path->layer().ll())
// {
// CPtrList<CLayer>
list(first()->layer().ul(),path->layer().ll(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->prepend( new gpath(CLayerPair(*it,NULL),r,r));
// }if(*first()->layer().ll()>*path->layer().ul())
// {
// CPtrList<CLayer>
list(first()->layer().ll(),path->layer().ul(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->prepend( new gpath(CLayerPair(*it,NULL),r,r));
// }
// }
// ((QPtrList<gpath>*)this)->prepend( path );
//}


QValueList<CTile> gwire::tiles(const QSize tileSize) const
{
QValueList<CTile> tiles;
QPtrListIterator<gpath> it(*this);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
for(it.toFirst();it.current();++it) tiles += (*it)->tiles(tileSize);
return tiles;
}

QValueList<CTileEdge> gwire::edges(const QSize tileSize) const
{
QValueList<CTileEdge> edges;
QPtrListIterator<gpath> it(*this);
for(it.toFirst();it.current();++it) edges += (*it)->edges(tileSize);
return edges;
}

void gwire::incTileFlows(const QSize tileSize) const
{
if(!m_routed) return;
QValueList<CTileEdge> edges = this->edges(tileSize);
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->incTileFlow((*eIt).track(),(*eIt).edge());
}

void gwire::decTileFlows(const QSize tileSize) const
{
if(!m_routed) return;
QValueList<CTileEdge> edges = this->edges(tileSize);
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->decTileFlow((*eIt).track(),(*eIt).edge());
}

void gwire::incCellFlows() const
{
QValueList<CTileEdge> edges = this->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->incCellFlow((*eIt).track(),(*eIt).edge());
}

void gwire::decCellFlows() const
{
QValueList<CTileEdge> edges = this->edges();
QValueListIterator<CTileEdge> eIt;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
(*eIt).layer()->decCellFlow((*eIt).track(),(*eIt).edge());
}

bool gwire::cutPathsAndBounds(CLayer* layer, const int track,
QValueList<CPathCut>& cutpaths)
{
QValueList<CPathCut> newcutpaths;
QValueListIterator<CPathCut> it;
QPtrListIterator<gpath> pIt(*this);
for(pIt.toFirst();pIt.current();++pIt)
{
gpath* path = *pIt;
if(path->routed())
{
if(!newcutpaths.isEmpty()) cutpaths += newcutpaths;
newcutpaths.clear();
continue;
}
int lb,ub;
if( path->cutsAndBounds(layer,track,lb,ub) )
{
bool exist = false;
for(it=newcutpaths.begin();it!=newcutpaths.end();it++)
{
int lbold = (*it).lb();
int ubold = (*it).ub();
if( (lb>=lbold && lb<=ubold) || (ub>=lbold
&& ub<=ubold) ||
(lbold>=lb &&lbold<=ub) || (ubold>=lb
&& ubold<=ub) )
{
(*it).append( path );
(*it).lb( ::min(lbold,lb) );
(*it).ub( ::max(ubold,ub) );
exist = true;break;
}
}
if(!exist) newcutpaths << CPathCut(this,lb,ub,path);
}
}
if(!newcutpaths.isEmpty()) cutpaths += newcutpaths;
newcutpaths.clear();
return !cutpaths.isEmpty();
}

QRect gwire::farfrom(gpath* path, gpath& farfrompath)
{
if(findRef(path)==-1) return QRect();
while(!prev()->routed());
farfrompath = *current();
return current()->to();
}

QRect gwire::farto(gpath* path, gpath& fartopath)
{
if(findRef(path)==-1) return QRect();
while(!next()->routed());
fartopath = *current();
return current()->from();
}

void gwire::rewire(QPtrList<gpath> paths, CLayer* layer, const int track, const
int edge)
{

// extern QTextStream lg;
// lg << "\tbefore rewire wire " << this << "\n";
// lg << layer->name() << ", " << track << ", " << edge << "\n";
// lg << m_from->name() << " --> " << m_to->name() << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
```

```
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }
//
// lg << "\tcuts\n";
// paths.first();
// while(paths.current())
// {
// gpath* path = paths.current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// paths.next();
// }


QPair<QRect,QRect> layerRects = layer->divideAtTrack(track);
QRect left = layerRects.first;
QRect right = layerRects.second;
gpath tmp;

QRect farfrom = this->farfrom(paths.first(),tmp);
QRect farto = this->farto(paths.last(),tmp);

QRect fromside,toside;
if(left.contains(farfrom)) fromside=left;
else if(right.contains(farfrom)) fromside=right;
else assert(false);
if(left.contains(farto)) toside=left;
else if(right.contains(farto)) toside=right;
else assert(false);

// {
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// farfrom.coords(&fx1,&fy1,&fx2,&fy2);
// farto.coords(&tx1,&ty1,&tx2,&ty2);
// lg << "farfrom to farto\n";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
//    }
//
//
// {
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// fromside.coords(&fx1,&fy1,&fx2,&fy2);
// toside.coords(&tx1,&ty1,&tx2,&ty2);
// lg << "from to to\n";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
//    }

gpath* newpath = NULL;
if(edge!=-1)
{
assert(fromside!=toside);
if(layer->dir()==VERTICAL)
newpath = new
gpath(CLayerPair(),QRect(edge,track-1,1,2)&fromside,QRect(edge,track-1,1,
2)&toside);
else if(layer->dir()==HORIZONTAL)
newpath = new
gpath(CLayerPair(),QRect(track-1,edge,2,1)&fromside,QRect(track-1,edge,2,
1)&toside);
}else assert(fromside==toside);

QPtrListIterator<gpath> it(paths);
for(it.toFirst();it.current();++it)
{
gpath* path = *it;
assert(!path->routed());
QRect from = path->from();
QRect to = path->to();
CLayerPair layerP = path->layer();
assert(layerP.vl()!=NULL && layerP.hl()!=NULL);

if(edge==-1)
{
path->from( from & fromside );
path->to( to & toside );
if(path->from()==path->to() &&
path->from().size()==QSize(1,1) ) removeRef(path);
}else if(newpath->routed())
{
path->from( from&toside );
path->to( to&toside );
    if(path->from()==path->to() && path->from().size()==QSize(1,1) )
removeRef(path);
}else if(path->contains(*newpath))
{
int pos = findRef(path);
remove(pos);
if((from&fromside)!=newpath->from() )
{insert(pos,new
gpath(layerP,from&fromside,newpath->from()));pos++;}
insert(pos,newpath);pos++;
if(newpath->to()!=(to&toside))
{insert(pos, new
gpath(layerP,newpath->to(),to&toside));pos++;}
newpath->layer(layerP);
newpath->routed(true);
newpath->incCellFlows();
}else
{
path->from( from&fromside );
path->to( to&fromside );
if(path->from()==path->to() &&
path->from().size()==QSize(1,1) ) removeRef(path);
}
}
if(edge!=-1) assert(newpath->routed());
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// lg << "\tafter rewire wire " << this << "\n";
// lg << layer->name() << ", " <<  track << ", " << edge << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }

}


void gwire::mergeStraightPaths()
{
//// extern QTextStream lg;
//// lg << "\tbefore merge wire " << this << "\n";
//// first();
//// while(current())
//// {
//// gpath* path = current();
//// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
//// path->from().coords(&fx1,&fy1,&fx2,&fy2);
//// path->to().coords(&tx1,&ty1,&tx2,&ty2);
//// if(path->routed()) lg << "routed"; else lg << "notrouted";
//// lg << " " << path << " \t";
//// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
//// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
//// next();
//// }

gpath* p1 = first();
gpath* p2 = next();
if(p2==0) return;
while(current())
{
if(!p1->routed()) removeRef(p1);
else if(!p1->edges().isEmpty() && !p2->edges().isEmpty())
{
CTileEdge e1 = p1->edges().last();
CTileEdge e2 = p2->edges().first();
if(*e1.layer()==*e2.layer() && e1.edge()==e2.edge())
{
assert((int)fabs(e1.track()-e2.track())<=1);
p2->from( p1->from() );
removeRef(p1);
}
}
p1=p2;
p2 = next();
}
```

```
// lg << "\tafter merge wire " << this << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }

}

void gwire::mergeExpandUnroutedPaths(const QSize gTile)
{

// extern QTextStream lg;
// lg << "\tbefore merge unrouted wire " << this << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }

//merge
gpath* p1 = first();
gpath* p2 = next();
if(p2==0) return;
while(current())
{
if(!p1->routed() && !p2->routed())
{
p2->from( p1->from() );
removeRef(p1);
}
p1=p2;
p2 = next();
}

// lg << "\tafter merge unrouted, before expand wire " << this << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }

//expand
gpath* p = first();
while(current())
{
p = current();
if( p->routed() || p==getFirst() && p==getLast() )
{next();continue;}
if(  (int)fabs(p->from().center().x()-p->to().center().x()) <
gTile.width() &&

(int)fabs(p->from().center().y()-p->to().center().y()) < gTile.height() )
{
int pos = at();
bool merged = false;
if(at(pos-1)!=getFirst())
   {
at(pos-1)->decCellFlows();
p->from( at(pos-1)->from() );
remove( pos-1 );
pos--;
merged = true;
}
if(at(pos+1)!=getLast())
{
at(pos+1)->decCellFlows();
p->to( at(pos+1)->to() );
remove( pos+1 );
merged = true;
}
p = at(pos);
if(merged) continue;
}
next();
}

//merge again (for merging paths that expanded and became neighbors)
p1 = first();
p2 = next();
if(p2==0) return;
while(current())
{
if(!p1->routed() && !p2->routed())
{
p2->from( p1->from() );
removeRef(p1);
}
p1=p2;
p2 = next();
}
//
// lg << "\tafter expand wire " << this << "\n";
// first();
```

```
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << " \t";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
// next();
// }

}

gwire& gwire::operator+=(gwire& wire)
{
if(!m_routed && m_from==NULL  && m_to==NULL)
{
m_from = wire.from();
m_to = wire.to();
}
if(m_from->type()==STEINER && !isEmpty()) removeFirst();
   wire.last();
if(wire.to()->type()==STEINER) wire.prev();
do
{
prepend(new gpath(*wire.current()));
wire.prev();
}while(wire.current());
m_from = wire.from();
return *this;
}


bool gwire::pp1(gpath* path)
{
CLayerPair oldpp = path->layer();
CLayerPair newpp = oldpp;
bool donep = false;
CLayerPair oldpm = path->layer();
CLayerPair newpm = oldpm;
bool donem = false;
do
{
if(!donep && pp1(path, newpp)) return true;
if(!donem && pp1(path, newpm)) return true;

oldpp = newpp;
if(!donep) newpp++;
if(oldpp==newpp) donep =true;

        oldpm = newpm;
if(!donem) newpm--;
if(oldpm==newpm) donem = true;

}while(!donep || !donem);
return false;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
}

bool gwire::pp1(gpath* path, CLayerPair layerp)
{
int pos = findRef(path);
gpath* prev = at(pos-1);
gpath* next = at(pos+1);
QPoint from = prev->from().center();
QPoint to = next->to().center();
//temporarly lay the wire
//first set of paths
CValueList<int> path1(from.x(),to.x(),false,true);
QValueListIterator<int> it;
for(it=path1.begin();it!=path1.end();it++)
{
gwire* wire = new gwire();
if(from.y()!=to.y())
{
//wire->append(new
gpath(layerp,grect(gpoint(*up)),grect(*it,from.y())));
wire->append(new
gpath(layerp,QRect(from,QSize(1,1)),QRect(*it,from.y(),1,1)));
if(*it!=to.x()) wire->append(new
gpath(layerp,QRect(*it,from.y(),1,1),QRect(*it,to.y(),1,1)));
else wire->append(new
gpath(layerp,QRect(*it,from.y(),1,1),QRect(to,QSize(1,1))));
//if(*it!=to.x() ) wire->append(new
gpath(layerp,grect(*it,to.y()),grect(gpoint(*down))));
if(*it!=to.x() ) wire->append(new
gpath(layerp,QRect(*it,to.y(),1,1),QRect(to,QSize(1,1))));
}else wire->append(new
gpath(layerp,QRect(from,QSize(1,1)),QRect(to,QSize(1,1))));
QValueList<CTileEdge> edges = wire->edges();
QValueListIterator<CTileEdge> eIt;
bool full = false;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){full=true;break;}
}
if(!full)
{
wire->setAutoDelete(false);
path->decCellFlows();
remove(path);
wire->first();
while(wire->current())
{
wire->current()->routed(true);
wire->current()->incCellFlows();
insert(pos,wire->current());
pos++;
wire->next();
}
if(prev!=first()) {prev->decCellFlows();remove(prev);}
if(next!=last()) {next->decCellFlows();remove(next);}
at(pos);
            delete wire;
return true;
}
delete wire;
if(from.y()==to.y()) break;//don't let more than one
}

//second set of paths
CValueList<int> path2(from.y(),to.y(),false,true);
for(it=path2.begin();it!=path2.end();it++)
{
gwire* wire = new gwire();
if(from.x()!=to.x())
{
//wire->append(new
gpath(layerp,grect(gpoint(*up)),grect(from.x(),*it)));
wire->append(new
gpath(layerp,QRect(from,QSize(1,1)),QRect(from.x(),*it,1,1)));
if(*it!=to.y()) wire->append(new
gpath(layerp,QRect(from.x(),*it,1,1),QRect(to.x(),*it,1,1)));
else  wire->append(new
gpath(layerp,QRect(from.x(),*it,1,1),QRect(to,QSize(1,1))));
//if(*it!=to.y()) wire->append(new
gpath(layerp,grect(to.x(),*it),grect(gpoint(*down))));
if(*it!=to.y()) wire->append(new
gpath(layerp,QRect(to.x(),*it,1,1),QRect(to,QSize(1,1))));
}else wire->append(new
gpath(layerp,QRect(from,QSize(1,1)),QRect(to,QSize(1,1))));
QValueList<CTileEdge> edges = wire->edges();
QValueListIterator<CTileEdge> eIt;
bool full = false;
for(eIt=edges.begin();eIt!=edges.end();eIt++)
{

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){full=true;break;}
}
if(!full)
{
wire->setAutoDelete(false);
path->decCellFlows();
remove(path);
wire->first();
while(wire->current())
{
wire->current()->routed(true);
wire->current()->incCellFlows();
insert(pos,wire->current());
pos++;
wire->next();
}
if(prev!=first()) {prev->decCellFlows();remove(prev);}
if(next!=last()) {next->decCellFlows();remove(next);}
at(pos);
            delete wire;
return true;
}
delete wire;
if(from.x()==to.x()) break;//don't let more than one same path
}
at(pos);
return false;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
}

bool gwire::pp2(gpath* path)
{
CLayerPair oldpp = path->layer();
CLayerPair newpp = oldpp;
bool donep = false;
CLayerPair oldpm = path->layer();
CLayerPair newpm = oldpm;
bool donem = false;
do
{
if(!donep && pp2(path, newpp)) return true;
if(!donem && pp2(path, newpm)) return true;

oldpp = newpp;
if(!donep) newpp++;
if(oldpp==newpp) donep =true;

        oldpm = newpm;
if(!donem) newpm--;
if(oldpm==newpm) donem = true;

}while(!donep || !donem);
return false;
}

bool gwire::pp2(gpath* path, CLayerPair layerp)
{

//extern QTextStream lg;
//lg << "trying ";
//lg << path->from().center().x() << " " << path->from().center().y();
//lg << " : " << path->to().center().x() << " " << path->to().center().y();
//lg << " at " << layerp.ll()->name() << " " << layerp.ul()->name() << "\n";

    int pos = findRef(path);

    QValueList<CTileEdge> edges = path->edges();
    QValueListIterator<CTileEdge> eIt;
    bool hasLeft = false;
    bool hasRight = false;
    CTileEdge left,right;

    for(eIt=edges.begin();eIt!=edges.end() && !hasLeft;eIt++)
    {
         CTileEdge edge = *eIt;
//lg << edge.layer()->name() << " " << edge.track() << "" << edge.edge();
         if(edge.layer()->cellOverflows(edge.track(),edge.edge()))
         {
//lg << " overflows";
             left=edge;
             hasLeft = true;
         }
//lg << "\n";
    }

    eIt=edges.end();
    do
    {
        eIt--;
        CTileEdge edge = *eIt;
//lg << edge.layer()->name() << " " << edge.track() << " " << edge.edge();
        if(edge.layer()->cellOverflows(edge.track(),edge.edge()))
        {
//lg << " overflows";
            right=edge;
            hasRight = true;
        }
//lg << "\n";
    }while(eIt!=edges.begin() && !hasRight);
    assert(hasLeft && hasRight);

    QPoint from,to;
    if(path->dir()==VERTICAL)
    {
        from.setX( left.edge() );
        from.setY( left.track() );
        to.setX( right.edge() );
        to.setY( right.track() );
        if(path->from().center().y()<path->to().center().y())
            from.setY( from.y()-1 );
        else if(path->to().center().y()<path->from().center().y())
            to.setY( to.y()-1 );
        else assert(false);
    }
    else if(path->dir()==HORIZONTAL)
    {
        from.setX( left.track() );
        from.setY( left.edge() );
        to.setX( right.track() );
        to.setY( right.edge() );
        if(path->from().center().x()<path->to().center().x())
            from.setX( from.x()-1 );
        else if(path->to().center().x()<path->from().center().x())
            to.setX( to.x()-1 );
        else assert(false);
    }
    else assert(false);
//lg << "from : " << from.x() << " " << from.y() << "\n";
//lg << "to : " << to.x() << " " << to.y() << "\n";

    if(path->dir()==VERTICAL)
    {
        int d=0;
        int maxd = (int)fabs(from.y()-to.y());
        do
        {
            gwire* wire = new gwire();
            if(d!=0) wire->append( new
gpath(layerp,QRect(from,QSize(1,1)),QRect(from.x()+d,from.y(),1,1)));
            wire->append( new
gpath(layerp,QRect(from.x()+d,from.y(),1,1),QRect(to.x()+d,to.y(),1,1)));
            if(d!=0) wire->append( new
gpath(layerp,QRect(to.x()+d,to.y(),1,1),QRect(to,QSize(1,1))));
            QValueList<CTileEdge> edges = wire->edges();
     QValueListIterator<CTileEdge> eIt;
//lg << "delta " << d << "\n";
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      bool full = false;
      for(eIt=edges.begin();eIt!=edges.end();eIt++)
      {
//lg << (*eIt).layer()->name() << " " <<  (*eIt).track() << " " <<
(*eIt).edge();

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){/*lg <<
"full\n";*/full=true;break;}
//lg << "\n";
      }
             if(!full)
             {
//lg << "inserting\n";
                 wire->setAutoDelete(false);
                 path->decCellFlows();
                 if(path->from().center()!=from) wire->prepend(new
gpath(path->layer(),path->from(),QRect(from,QSize(1,1))));
                 if(path->to().center()!=to) wire->append(new
gpath(path->layer(),QRect(to,QSize(1,1)),path->to()));
                 remove(path);
     wire->first();
     while(wire->current())
     {
//lg << " " << wire->current()->from().center().x() << " " <<
wire->current()->from().center().y();
//lg << " " << wire->current()->to().center().x() << " " <<
wire->current()->to().center().y() << "\n";
     wire->current()->routed(true);
     wire->current()->incCellFlows();
     insert(pos,wire->current());
     pos++;
     wire->next();
     }
     at(pos);
                 delete wire;
     return true;
             }
             delete wire;

             wire = new gwire();
             if(d!=0)wire->append( new
gpath(layerp,QRect(from,QSize(1,1)),QRect(from.x()-d,from.y(),1,1)));
             wire->append( new
gpath(layerp,QRect(from.x()-d,from.y(),1,1),QRect(to.x()-d,to.y(),1,1)));
             if(d!=0)wire->append( new
gpath(layerp,QRect(to.x()-d,to.y(),1,1),QRect(to,QSize(1,1))));
             edges = wire->edges();
      full = false;
//lg << "delta " << d << "\n";
      for(eIt=edges.begin();eIt!=edges.end();eIt++)
      {
//lg << (*eIt).layer()->name() << " " <<  (*eIt).track() << " " <<
(*eIt).edge();

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){/*lg <<
"full\n";*/full=true;break;}
//lg << "\n";
      }
              if(!full)
             {
//lg << "inserting\n";
                 wire->setAutoDelete(false);
                 path->decCellFlows();
                 if(path->from().center()!=from) wire->prepend(new
gpath(path->layer(),path->from(),QRect(from,QSize(1,1))));
                 if(path->to().center()!=to) wire->append(new
gpath(path->layer(),QRect(to,QSize(1,1)),path->to()));
                 remove(path);
     wire->first();
     while(wire->current())
     {
//lg << " " << wire->current()->from().center().x() << " " <<
wire->current()->from().center().y();
//lg << " " << wire->current()->to().center().x() << " " <<
wire->current()->to().center().y() << "\n";
     wire->current()->routed(true);
     wire->current()->incCellFlows();
     insert(pos,wire->current());
     pos++;
     wire->next();
     }
     at(pos);
                 delete wire;
     return true;
             }
             delete wire;
         }while(d++<maxd);
     }else if(path->dir()==HORIZONTAL)
     {
         int d=0;
         int maxd = (int)fabs(from.x()-to.x());
         do
         {
             gwire* wire = new gwire();
             if(d!=0)wire->append( new
gpath(layerp,QRect(from,QSize(1,1)),QRect(from.x(),from.y()+d,1,1)));
             wire->append( new
gpath(layerp,QRect(from.x(),from.y()+d,1,1),QRect(to.x(),to.y()+d,1,1)));
             if(d!=0)wire->append( new
gpath(layerp,QRect(to.x(),to.y()+d,1,1),QRect(to,QSize(1,1))));
             QValueList<CTileEdge> edges = wire->edges();
      QValueListIterator<CTileEdge> eIt;
      bool full = false;
//lg << "delta " << d << "\n";
      for(eIt=edges.begin();eIt!=edges.end();eIt++)
      {
//lg << (*eIt).layer()->name() << " " <<  (*eIt).track() << " " <<
(*eIt).edge();

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){/*lg <<
"full\n";*/full=true;break;}
//lg << "\n";
      }
             if(!full)
             {
//lg << "inserting\n";
                 wire->setAutoDelete(false);
                 path->decCellFlows();
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
                if(path->from().center()!=from) wire->prepend(new
gpath(path->layer(),path->from(),QRect(from,QSize(1,1))));
                if(path->to().center()!=to) wire->append(new
gpath(path->layer(),QRect(to,QSize(1,1)),path->to()));
                remove(path);
    wire->first();
    while(wire->current())
    {
//lg << " " << wire->current()->from().center().x() << " " <<
wire->current()->from().center().y();
//lg << " " << wire->current()->to().center().x() << " " <<
wire->current()->to().center().y() << "\n";
    wire->current()->routed(true);
    wire->current()->incCellFlows();
    insert(pos,wire->current());
    pos++;
    wire->next();
    }
    at(pos);
                delete wire;
    return true;
            }
            delete wire;


            wire = new gwire();
            if(d!=0)wire->append( new
gpath(layerp,QRect(from,QSize(1,1)),QRect(from.x(),from.y()-d,1,1)));
            wire->append( new
gpath(layerp,QRect(from.x(),from.y()-d,1,1),QRect(to.x(),to.y()-d,1,1)));
            if(d!=0)wire->append( new
gpath(layerp,QRect(to.x(),to.y()-d,1,1),QRect(to,QSize(1,1))));
            edges = wire->edges();
     full = false;
//lg << "delta " << d << "\n";
     for(eIt=edges.begin();eIt!=edges.end();eIt++)
     {
//lg << (*eIt).layer()->name() << " " <<  (*eIt).track() << " " <<
(*eIt).edge();

if((*eIt).layer()->cellFull((*eIt).track(),(*eIt).edge())){/*lg <<
"full\n";*/full=true;break;}
//lg << "\n";
     }
            if(!full)
            {
//lg << "inserting\n";
                wire->setAutoDelete(false);
                path->decCellFlows();
                if(path->from().center()!=from) wire->prepend(new
gpath(path->layer(),path->from(),QRect(from,QSize(1,1))));
                if(path->to().center()!=to) wire->append(new
gpath(path->layer(),QRect(to,QSize(1,1)),path->to()));
                remove(path);
    wire->first();
    while(wire->current())
    {
//lg << " " << wire->current()->from().center().x() << " " <<
wire->current()->from().center().y();
//lg << " " << wire->current()->to().center().x() << " " <<
wire->current()->to().center().y() << "\n";
    wire->current()->routed(true);
    wire->current()->incCellFlows();
    insert(pos,wire->current());
    pos++;
    wire->next();
    }
    at(pos);
                delete wire;
    return true;
            }
            delete wire;
        }while(d++<maxd);
    }
    else assert(false);
    return false;
}

void gwire::pp3()
{
first();
gpath* path = NULL;
while(current())
{
if(current()!=getFirst())
{
if(!path->layer().isNull() &&
!current()->layer().isNull())
{
bool viol = false;
QRect r = path->to() & current()->from();

if(*path->layer().ul()<=*current()->layer().ll())
{
bool include1,include2;

if(path->dir()!=path->layer().ul()->dir()) include1 = true; else include1 =
false;
if(path->dir()==UNKNOWNLAYERDIRECTION)
include1 = false;

if(current()->dir()!=current()->layer().ll()->dir() ) include2 = true; else
include2 = false;

if(current()->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
CPtrList<CLayer>
list(path->layer().ul(),current()->layer().ll(),include1,include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol =
```

```
true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol =
true;break;
}
}
list.next();
}
}else
if(*path->layer().ll()>=*current()->layer().ul())
{
bool include1,include2;

if(path->dir()!=path->layer().ll()->dir()) include1 = true; else include1 =
false;
if(path->dir()==UNKNOWNLAYERDIRECTION)
include1 = false;

if(current()->dir()!=current()->layer().ul()->dir()) include2 = true; else
include2 = false;

if(current()->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
CPtrList<CLayer>
list(path->layer().ll(),current()->layer().ul(),include1,include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol=true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol=true;break;
}
}
list.next();
}
}
if(viol) pp3(path,current());
}
}
path = current();
next();
}
}

void gwire::pp3(gpath* prev, gpath* next)
{
assert(prev->to()==next->from());
CPtrList<CLayer> listp,listn;
if(prev->dir()!=UNKNOWNLAYERDIRECTION &&
next->dir()!=UNKNOWNLAYERDIRECTION &&
prev->dir()==next->dir())
{
// QRect r = prev->to() & next->from();
if(*prev->layer().ul()<=*next->layer().ll())
{
bool include1,include2;
  if(prev->dir()!=prev->layer().ul()->dir()) include1 =
true; else include1 = false;
  if(next->dir()!=next->layer().ll()->dir() ) include2 =
true; else include2 = false;
  listp =
CPtrList<CLayer>::CPtrList(prev->layer().ul(),next->layer().ll(),include1,
include2);
listn = listp;
//std::cout << prev->layer().ul()->name() << include1;
//std::cout << " " << next->layer().ll()->name() << include2 << "\n";
  }else if(*prev->layer().ll()>=*next->layer().ul())
  {
  bool include1,include2;
  if(prev->dir()!=prev->layer().ll()->dir()) include1 =
true; else include1 = false;
  if(next->dir()!=next->layer().ul()->dir()) include2 =
true; else include2 = false;
  listp =
CPtrList<CLayer>::CPtrList(prev->layer().ll(),next->layer().ul(),include1,
include2);
listn = listp;
//std::cout << prev->layer().ll()->name() << include1;
//std::cout << " " << next->layer().ul()->name() << include2 << "\n";
  }else assert(false);
}else if( prev->dir()!=UNKNOWNLAYERDIRECTION &&
next->dir()!=UNKNOWNLAYERDIRECTION &&
prev->dir()!=next->dir())
{
// QRect r = prev->to() & next->from();
if(*prev->layer().ul()<=*next->layer().ll())
{
bool include1,include2;
  if(prev->dir()!=prev->layer().ul()->dir()) include1 =
true; else include1 = false;
  if(prev->dir()!=next->layer().ll()->dir() ) include2 =
true; else include2 = false;
  listp =
CPtrList<CLayer>::CPtrList(prev->layer().ul(),next->layer().ll(),include1,
include2);
//std::cout << prev->layer().ul()->name() << include1;
//std::cout << " " << next->layer().ll()->name() << include2 << "\n";
  }else if(*prev->layer().ll()>=*next->layer().ul())
  {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
  bool include1,include2;
  if(prev->dir()!=prev->layer().ll()->dir()) include1 =
true; else include1 = false;
  if(prev->dir()!=next->layer().ul()->dir()) include2 =
true; else include2 = false;
  listp =
CPtrList<CLayer>::CPtrList(prev->layer().ll(),next->layer().ul(),include1,
include2);
//std::cout << prev->layer().ll()->name() << include1;
//std::cout << " " << next->layer().ul()->name() << include2 << "\n";
  }else assert(false);

if(*prev->layer().ul()<=*next->layer().ll())
{
bool include1,include2;
  if(next->dir()!=prev->layer().ul()->dir()) include1 =
true; else include1 = false;
  if(next->dir()!=next->layer().ll()->dir() ) include2 =
true; else include2 = false;
  listn =
CPtrList<CLayer>::CPtrList(prev->layer().ul(),next->layer().ll(),include1,
include2);
//std::cout << prev->layer().ul()->name() << include1;
//std::cout << " " << next->layer().ll()->name() << include2 << "\n";
  }else if(*prev->layer().ll()>=*next->layer().ul())
  {
  bool include1,include2;
  if(next->dir()!=prev->layer().ll()->dir()) include1 =
true; else include1 = false;
  if(next->dir()!=next->layer().ul()->dir()) include2 =
true; else include2 = false;
  listn =
CPtrList<CLayer>::CPtrList(prev->layer().ll(),next->layer().ul(),include1,
include2);
//std::cout << prev->layer().ll()->name() << include1;
//std::cout << " " << next->layer().ul()->name() << include2 << "\n";
  }else assert(false);
}
// std::cout << "\n" << listp.count() << "\n";
// std::cout << listn.count() << "\n\n";
// std::cout.flush();
// listn.first();
// while(listn.current()){std::cout << listn.current()->name() <<
"\n";listn.next();}
// std::cout << "\n";
// listp.first();
// while(listp.current()){std::cout << listp.current()->name() <<
"\n";listp.next();}
// std::cout.flush();

QValueList<CTile> tilesp = prev->tiles();
QValueList<CTile> tilesn = next->tiles();
QValueListIterator<CTile> itp,itn;
itp = tilesp.end();
itn = tilesn.begin();
bool donep = false;
bool donen = false;
do
{
```

```
if(itp!=tilesp.begin()) itp--;

CTile tilep = *itp;
CTile tilen = *itn;
if(!donep && !listp.isEmpty())
{
bool full = false;
listp.first();
while(listp.current())
{
CLayer* l = listp.current();
if(l->dir()==VERTICAL)
{
if(
l->cellBlocked(tilep.y(),tilep.x()) &&

l->cellBlocked(tilep.y()+1,tilep.x()))
{
full=true;break;
}
}else if(l->dir()==HORIZONTAL)
{
if(
l->cellBlocked(tilep.x(),tilep.y()) &&

l->cellBlocked(tilep.x()+1,tilep.y()))
{
full=true;break;
}
}
listp.next();
}
if(!full)
{
if(prev->dir()==next->dir())
{
prev->decCellFlows();
next->decCellFlows();
prev->to(tilep.rect());
next->from(tilep.rect());
prev->incCellFlows();
next->incCellFlows();
}else
{
prev->decCellFlows();
prev->to(tilep.rect());
gpath* path = new
gpath(next->layer(),tilep.rect(),next->from(),true);
insert(at(),path);
path->incCellFlows();
prev->incCellFlows();
this->next();
}
donep = true;
donen = true;
break;
}

}
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if(!donen && !listn.isEmpty())
{
listn.first();
bool full = false;
while(listn.current())
{
CLayer* l = listn.current();
if(l->dir()==VERTICAL)
{
if(
l->cellBlocked(tilep.y(),tilep.x()) &&

l->cellBlocked(tilep.y()+1,tilep.x()))
{
full=true;break;
}
}else if(l->dir()==HORIZONTAL)
{
if(
l->cellBlocked(tilep.x(),tilep.y()) &&

l->cellBlocked(tilep.x()+1,tilep.y()))
{
full=true;break;
}
}
listn.next();
}
if(!full)
{
if(prev->dir()==next->dir())
{
prev->decCellFlows();
next->decCellFlows();
prev->to(tilen.rect());
next->from(tilen.rect());
prev->incCellFlows();
next->incCellFlows();
}else
{
next->decCellFlows();
next->from(tilen.rect());
gpath* path = new
gpath(prev->layer(),prev->to(),tilen.rect(),true);
insert(at(),path);
path->incCellFlows();
next->incCellFlows();
this->next();
}
donep = true;
donen = true;
break;
}
}
if(itn!=tilesn.end()) itn++;
if(itp==tilesp.begin()) donep = true;
if(itn==tilesn.end()) donen = true;
}while(!donep || !donen );
}
```

```
void gwire::pp4()
{
if(!first()->layer().isNull() && m_from->type()!=STEINER)
{
CPtrList<CLayer> list;
gpath* from = first();
gpath* path = next();
assert(from->dir()==UNKNOWNLAYERDIRECTION);
bool viol = false;
QRect r = from->to() & path->from();
if(*from->layer().ul()<=*path->layer().ll())
{
bool include1,include2;
include1 = false;
if(path->dir()!=path->layer().ll()->dir() ) include2 =
true; else include2 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
list =
CPtrList<CLayer>::CPtrList(from->layer().ul(),path->layer().ll(),include1,
include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol = true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol = true;break;
}
}
list.next();
}
}else if(*from->layer().ll()>=*path->layer().ul())
{
bool include1,include2;
include1 = false;
if(path->dir()!=path->layer().ul()->dir()) include2 =
true; else include2 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
list =
CPtrList<CLayer>::CPtrList(from->layer().ll(),path->layer().ul(),include1,
include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol=true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol=true;break;
}
}
list.next();
}
}
if(viol) pp4from(list);
}


if(!last()->layer().isNull() && m_to->type()!=STEINER)
{
CPtrList<CLayer> list;
gpath* to = last();
gpath* path = prev();
assert(to->dir()==UNKNOWNLAYERDIRECTION);
bool viol = false;
QRect r = to->to() & path->from();
if(*path->layer().ul()<=*to->layer().ll())
{
bool include1,include2;
if(path->dir()!=path->layer().ul()->dir() ) include1 =
true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1 = false;
include2 = false;
list =
CPtrList<CLayer>::CPtrList(path->layer().ul(),to->layer().ll(),include1,
include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol = true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol = true;break;
}
}
list.next();
}
}else if(*path->layer().ll()>=*to->layer().ul())
{
bool include1,include2;
if(path->dir()!=path->layer().ll()->dir()) include1 =
true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1 = false;
include2 =false;
list =
CPtrList<CLayer>::CPtrList(path->layer().ll(),to->layer().ul(),include1,
include2);
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
viol=true;break;
}
}else if(l->dir()==HORIZONTAL)
{

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
viol=true;break;
}
}
list.next();
}
}
if(viol) pp4to(list);
}
}

void gwire::pp4from(CPtrList<CLayer> list)
{
first();
gpath* next = this->next();
QValueList<CTile> tiles = next->tiles();
QValueListIterator<CTile> itt;
itt = tiles.begin();
bool done = false;
do
{
CTile tile = *itt;
if(!done && !list.isEmpty())
{
bool full = false;
list.first();
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{
if(
l->cellBlocked(tile.y(),tile.x()) &&

l->cellBlocked(tile.y()+1,tile.x()))
{
full=true;break;
}
}else if(l->dir()==HORIZONTAL)
{
if(
l->cellBlocked(tile.x(),tile.y()) &&

l->cellBlocked(tile.x()+1,tile.y()))
{
full=true;break;
}
}
list.next();
}
if(!full)
{
next->decCellFlows();
next->from(tile.rect());

CLayer* l1 = first()->layer().l1();
CLayer* l2 = list.first();
gpath* path = new
gpath(CLayerPair(l1,l2),first()->to(),tile.rect(),true);
insert(1,path);
path->incCellFlows();
next->incCellFlows();
done = true;
break;
}
}
if(itt!=tiles.end()) itt++;
if(itt==tiles.end()) done = true;
}while(!done);
}


void gwire::pp4to(CPtrList<CLayer> list)
{
last();
gpath* prev = this->prev();
QValueList<CTile> tiles = prev->tiles();
QValueListIterator<CTile> itt;
itt = tiles.end();
bool done = false;
do
{
if(itt!=tiles.begin()) itt--;
CTile tile = *itt;
if(!done && !list.isEmpty())
{
bool full = false;
list.first();
while(list.current())
{
CLayer* l = list.current();
if(l->dir()==VERTICAL)
{
if(
l->cellBlocked(tile.y(),tile.x()) &&

l->cellBlocked(tile.y()+1,tile.x()))
{
full=true;break;
}
}else if(l->dir()==HORIZONTAL)
{
if(
l->cellBlocked(tile.x(),tile.y()) &&

l->cellBlocked(tile.x()+1,tile.y()))
{
full=true;break;
}
}
list.next();
}
if(!full)
{
prev->decCellFlows();
prev->to(tile.rect());

CLayer* l1 = last()->layer().l1();
CLayer* l2 = list.last();
gpath* path = new
gpath(CLayerPair(l1,l2),tile.rect(),last()->from(),true);
insert(count()-1,path);
path->incCellFlows();
prev->incCellFlows();
done = true;
break;
}
}
if(itt==tiles.begin()) done = true;
}while(!done);
}


void gwire::movePath(gpath* path, CLayerPair layer, const  int dx, const int dy)
{
if(!layer.isNull())
{
int pos = findRef(path); assert(pos>0);
    path->layer( layer );
    }
if(dx!=0 || dy!=0)
{
int pos = findRef(path); assert(pos>0);
```

```
QRect from  = path->from();
QRect to = path->to();
from.moveBy(dx,-dy);to.moveBy(dx,-dy);
path->from( from );
path->to( to );
gpath* prev = at(pos-1);
if(prev->dir()!=path->dir() &&
prev->dir()!=UNKNOWNLAYERDIRECTION ) prev->to( path->from() );
else{((QPtrList<gpath>*)this)->insert( pos, new
gpath(path->layer(),prev->to(),path->from(),true));}
gpath* next = at(pos+1);
if(next->dir()!=path->dir() &&
next->dir()!=UNKNOWNLAYERDIRECTION) next->from( path->to() );
else{((QPtrList<gpath>*)this)->insert( pos+1, new
gpath(path->layer(),path->to(),next->from(),true));}
}
}

void gwire::rteWrite(QTextStream& f)
{
  static int via = 0;
f << "\t(gwire ";
first();
gpath* path = NULL;
while(current())
{
if(current()!=getFirst())
{
QRect r = path->to() & current()->from();
if(*path->layer().ul()<=*current()->layer().ll())
{
bool include1,include2;
if(path->dir()!=path->layer().ul()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
= false;

if(current()->dir()!=current()->layer().ll()->dir() ) include2 = true; else
include2 = false;
if(current()->dir()==UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ul(),current()->layer().ll(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it)
{
CLayerPair layer(*it,NULL);
gpath(layer,r,r,true).rteWrite(f);
/*
{
  CLayer* l = layer.ll();
  if(l->dir()==VERTICAL)
  {
      if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
       {
           via++;
           f << "#" << via << " VIA OVERFLOW "  << l->name() << " " <<
r.center().x() << " " << r.center().y() << "\n";
       }
  }else if(l->dir()==HORIZONTAL)
  {
      if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
       {
           via++;
           f << "#"<<via << " VIA OVERFLOW "  << l->name() << " " <<
r.center().x() << " " << r.center().y() << "\n";
       }
  }
}*/
}
}else if(*path->layer().ll()>=*current()->layer().ul())
{
bool include1,include2;
if(path->dir()!=path->layer().ll()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
= false;

if(current()->dir()!=current()->layer().ul()->dir()) include2 = true; else
include2 = false;
if(current()->dir()==UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ll(),current()->layer().ul(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it)
{
CLayerPair layer(*it,NULL);
gpath(layer,r,r,true).rteWrite(f);
/*{
  CLayer* l = layer.ll();
  if(l->dir()==VERTICAL)
  {
      if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
       {
           via++;
           f << "#"<<via << " VIA OVERFLOW "  << l->name() << " " <<
r.center().x() << " " << r.center().y() << "\n";
       }
  }else if(l->dir()==HORIZONTAL)
  {
      if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
       {
           via++;
           f << "#"<<via << " VIA OVERFLOW "  << l->name() << " " <<
r.center().x() << " " << r.center().y() << "\n";
       }
  }
}*/
}
}
}
path = current();
path->rteWrite(f);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
next();
}
f << "\t(connect";
f << " (terminal pin " << m_from->name() << "-A)";
f << " (terminal pin " << m_to->name() << "-A)))\n";
}

void gwire::grteWrite(QTextStream& f)
{
f << "\t(gwire ";
first();
gpath* path = NULL;
while(current())
{
if(current()!=getFirst())
{
QRect r = path->to() & current()->from();
if(*path->layer().ul()<=*current()->layer().ll())
{
bool include1,include2;
if(path->dir()!=path->layer().ul()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
= false;
if(current()->dir() !=
current()->layer().ll()->dir() ) include2 = true; else include2 = false;
if(current()->dir() == UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ul(),current()->layer().ll(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it){CLayerPair
layer(*it,NULL); gpath(layer,r,r,true).grteWrite(f);}
}if(*path->layer().ll()>=*current()->layer().ul())
{
bool include1,include2;
if(path->dir()!=path->layer().ll()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
= false;

if(current()->dir()!=current()->layer().ul()->dir()) include2 = true; else
include2 = false;
if(current()->dir()==UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ll(),current()->layer().ul(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it){CLayerPair
layer(*it,NULL);gpath(layer,r,r,true).grteWrite(f);}
}
}
path = current();
path->grteWrite(f);
next();
}
f << "\t(connect";
f << " (terminal pin " << m_from->name() << "-A)";
f << " (terminal pin " << m_to->name() << "-A)))\n";
}

int gwire::numPadViolations()
{
// extern QTextStream lg;
// lg << "\n\nWIRE\n";
// lg << m_from->name() << " --> " << m_to->name() << "\n";

int via = 0;
first();
gpath* path = NULL;
while(current())
{
//lg << "\nPATH\n";
if(current()!=getFirst())
{
//lg << "RECT1 " << path->to().center().x() << " " << path->to().center().y() <<
"\n";
if(!path->layer().isNull() &&
!current()->layer().isNull())
{
QRect r = path->to() & current()->from();
//lg << "RECT2 " << r.center().x() << " " << r.center().y() << "\n";

if(*path->layer().ul()<=*current()->layer().ll())
{
bool include1,include2;

if(path->dir()!=path->layer().ul()->dir()) include1 = true; else include1 =
false;
if(path->dir()==UNKNOWNLAYERDIRECTION)
include1 = false;

if(current()->dir()!=current()->layer().ll()->dir() ) include2 = true; else
include2 = false;

if(current()->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
CPtrList<CLayer>
list(path->layer().ul(),current()->layer().ll(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it)
{
CLayerPair layer(*it,NULL);
CLayer* l = layer.ll();
//lg << "Layer " << l->name() << "\n";
if(l->dir()==VERTICAL)
{
//lg << "track, edge " << r.center().y() << " , " << r.center().x() << "\n";
//if(l->cellBlocked(r.center().y(),r.center().x())) lg << "* ";
//lg << l->cellFlow(r.center().y(),r.center().x());
//lg << " + " << l->cellBlock(r.center().y(),r.center().x());
//if(l->cellBlocked(r.center().y()+1,r.center().x())) lg << "* ";
//lg << " " << l->cellFlow(r.center().y()+1,r.center().x());
//lg << " + " << l->cellBlock(r.center().y()+1,r.center().x()) << "\n";

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
//lg << "via viol " << via+1 << "\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
via++;
}
}else if(l->dir()==HORIZONTAL)
{
//lg << "track, edge " << r.center().x() << " , " << r.center().y() << "\n";
//if(l->cellBlocked(r.center().x(),r.center().y())) lg << "* ";
//lg << l->cellFlow(r.center().x(),r.center().y());
//lg << " + " << l->cellBlock(r.center().x(),r.center().y());
//if(l->cellBlocked(r.center().x()+1,r.center().y())) lg << "* ";
//lg << " " << l->cellFlow(r.center().x()+1,r.center().y());
//lg << " + " << l->cellBlock(r.center().x()+1,r.center().y()) << "\n";

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
//lg << "via viol " << via+1 << "\n";
via++;
}
}
}
}else
if(*path->layer().ll()>=*current()->layer().ul())
{
bool include1,include2;

if(path->dir()!=path->layer().ll()->dir()) include1 = true; else include1 =
false;
if(path->dir()==UNKNOWNLAYERDIRECTION)
include1 = false;

if(current()->dir()!=current()->layer().ul()->dir()) include2 = true; else
include2 = false;

if(current()->dir()==UNKNOWNLAYERDIRECTION) include2 = false;
CPtrList<CLayer>
list(path->layer().ll(),current()->layer().ul(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it)
{
CLayerPair layer(*it,NULL);
CLayer* l = layer.ll();
//lg << "Layer " <<  l->name() << "\n";
if(l->dir()==VERTICAL)
{
//lg << "track, edge " << r.center().y() << " , " << r.center().x() << "\n";
//if(l->cellBlocked(r.center().y(),r.center().x())) lg << "* ";
//lg << l->cellFlow(r.center().y(),r.center().x());
//lg << " + " << l->cellBlock(r.center().y(),r.center().x());
//if(l->cellBlocked(r.center().y()+1,r.center().x())) lg << "* ";
//lg << " " << l->cellFlow(r.center().y()+1,r.center().x());
//lg << " + " << l->cellBlock(r.center().y()+1,r.center().x()) << "\n";

if(l->cellBlocked(r.center().y(),r.center().x()) &&
l->cellBlocked(r.center().y()+1,r.center().x()))
{
//lg << "via viol " << via+1 << "\n";
via++;
}
}else if(l->dir()==HORIZONTAL)
{
//lg << "track, edge " << r.center().x() << " , " << r.center().y() << "\n";
//if(l->cellBlocked(r.center().x(),r.center().y())) lg << "* ";
//lg << l->cellFlow(r.center().x(),r.center().y());
//lg << " + " << l->cellBlock(r.center().x(),r.center().y());
//if(l->cellBlocked(r.center().x()+1,r.center().y())) lg << "* ";
//lg << " " << l->cellFlow(r.center().x()+1,r.center().y());
//lg << " + " << l->cellBlock(r.center().x()+1,r.center().y()) << "\n";

if(l->cellBlocked(r.center().x(),r.center().y()) &&
l->cellBlocked(r.center().x()+1,r.center().y()))
{
//lg << "Layer " <<  l->name() << "\n";
via++;
}
}
}
}
}
}
path = current();
next();
}
return via;
}


void gwire::paint(QPainter* p)
{
first();
gpath* path = NULL;
while(current())
{
if(current()!=getFirst())
{
QRect r = path->to() & current()->from();
if(*path->layer().ul()<=*current()->layer().ll())
{
bool include1,include2;
if(path->dir()!=path->layer().ul()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
= false;

if(current()->dir()!=current()->layer().ll()->dir() ) include2 = true; else
include2 = false;
if(current()->dir()==UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ul(),current()->layer().ll(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it){CLayerPair
layer(*it,NULL); gpath(layer,r,r,true).paint(p);}
}if(*path->layer().ll()>=*current()->layer().ul())
{
bool include1,include2;
if(path->dir()!=path->layer().ll()->dir())
include1 = true; else include1 = false;
if(path->dir()==UNKNOWNLAYERDIRECTION) include1
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
= false;

if(current()->dir()!=current()->layer().ul()->dir()) include2 = true; else
include2 = false;
if(current()->dir()==UNKNOWNLAYERDIRECTION)
include2 = false;
CPtrList<CLayer>
list(path->layer().ll(),current()->layer().ul(),include1,include2);
QPtrListIterator<CLayer> it(list);
for(it.toFirst();it.current();++it){CLayerPair
layer(*it,NULL);gpath(layer,r,r,true).paint(p);}
}
}
path = current();
path->paint(p);
next();
}
}


//void gwire::append(gpath* path)
//{
// if(!isEmpty() && !last()->layer().isNull() && !path->layer().isNull())
// {
// QRect r = last()->to() & path->from();
// if(*last()->layer().ul()<*path->layer().ll())
// {
// CPtrList<CLayer>
list(last()->layer().ul(),path->layer().ll(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->append( new gpath(CLayerPair(*it,NULL),r,r));
// }if(*last()->layer().ll()>*path->layer().ul())
// {
// CPtrList<CLayer>
list(last()->layer().ll(),path->layer().ul(),false,false);
// QPtrListIterator<CLayer> it(list);
// for(it.toFirst();it.current();++it)
((QPtrList<gpath>*)this)->append( new gpath(CLayerPair(*it,NULL),r,r));
// }
// }
// ((QPtrList<gpath>*)this)->append( path );
//}


//extern QTextStream lg;
// lg << "before merge wire " << this << "\n";
// first();
// while(current())
// {
// gpath* path = current();
// int fx1,fy1,fx2,fy2,tx1,ty1,tx2,ty2;
// path->from().coords(&fx1,&fy1,&fx2,&fy2);
// path->to().coords(&tx1,&ty1,&tx2,&ty2);
// if(path->routed()) lg << "routed"; else lg << "notrouted";
// lg << " " << path << "\n";
// lg << fx1 << " " << fy1  << " " << fx2 << " " << fy2 << " --> ";
// lg << tx1 << " " << ty1  << " " << tx2 << " " << ty2 << "\n";
```

```
// next();
// }


#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <qapplication.h>
#include "global.h"
#include "util.h"
#include "clef.h"
#include "cdef.h"
#include "cstr.h"
#include "clib.h"
#include "cpla.h"
#include "cnetlist.h"
#include "cdsn.h"
#include "ctim.h"
#include "ctiming.h"
#include "crte.h"
#include "cgroute.h"
#include "cutil.h"

CLef lef;
CDef def;
CStr str;
CLib lib;
CPla pla;
CNetList netlist;
CDsn dsn;
CTim tim;
CTiming timing;
CRte rte;
CGRoute route;
CUtil util;
bool hasLefDef;
bool hasCct;
bool hasTim;
bool hasTiming;
QFile logFile;
QTextStream lg;


int main(int argc, char *argv[])
{
QApplication app(argc, argv,true);
init();
if(argc>1) load(argv[1]);
std::cout << "exiting global router.\n";
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
dispose();
app.exit();
return 0;
// return app.exec();
}


#include "cgroute.h"
#include "clayer.h"
#include "ctile.h"
#include "gwire.h"
#include "gpath.h"
#include "cnode.h"
#include "cmaze.h"

void CGRoute::maze()
{
QPtrList<gwire> wires = this->wires();
wires.first();
while(wires.current())
{
gwire* wire = wires.current();
QValueList<CTileEdge> edges = wire->edges();
QValueListIterator<CTileEdge> eit;
bool overflows = false;
for(eit=edges.begin();eit!=edges.end() && !overflows;eit++)

if((*eit).layer()->cellOverflows((*eit).track(),(*eit).edge())) overflows =
true;
if(overflows || wire->numPadViolations()>0 ) maze(wire);
wires.next();
}
}


void CGRoute::maze(gwire* wire)
{
QValueList<CTile> tiles = wire->tiles();
CTile from = tiles.first();
CTile to = tiles.last();
CMaze* maze = new CMaze(from,to,m_layers,wire);
maze->layerChangeCost(m_layerChangeCost);
std::cout << "\n------------------------------------------\n";
  std::cout << "over \t:" <<  numCellOverflows() << "\n";
  std::cout << "pad \t:" << numPadViolations() << "\n";
  std::cout << "maze ";
if(maze->route()) std::cout << " succ";
std::cout << "\n";
std::cout << "over \t:" <<  numCellOverflows() << "\n";
  std::cout << "pad \t:" << numPadViolations() << "\n";
  std::cout << "------------------------------------------\n\n";

delete maze;

}
```

```
#include "openset.h"

#include "util.h"
#include "clef.h"
#include "cdef.h"
#include "cstr.h"
#include "clib.h"
#include "cpla.h"
#include "cnetlist.h"
#include "cdsn.h"
#include "ctim.h"
#include "ctiming.h"
#include "crte.h"
#include "cgroute.h"
#include "clayer.h"
#include "cutil.h"

void init()
{
extern bool hasLefDef;
extern bool hasCct;
hasLefDef= false;
hasCct = false;
extern QFile logFile;
logFile.setName("gr.log");
logFile.open(IO_ReadOnly | IO_Append);
extern QTextStream lg;
lg.setDevice(&logFile);
}

void dispose()
{
extern QFile logFile;
logFile.close();
}

bool load(const QString fileName)
{
QFile doFile(fileName);
if(!doFile.open(IO_ReadOnly)) return false;
QTextStream doStr(&doFile);
while(!doStr.atEnd()) parseCmd(doStr.readLine());
doFile.close();
return true;
}

bool parseCmd(const QString cmd)
{
extern CLef lef;
extern CDef def;
extern CStr str;
extern CLib lib;
extern CPla pla;
extern CNetList netlist;
extern CDsn dsn;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


```
extern CTim tim;
extern CTiming timing;
extern CRte rte;
extern CGRoute route;
extern bool hasLefDef;
extern bool hasCct;
extern bool hasTim;
extern bool hasTiming;
extern CUtil util;

if(cmd.isNull() || cmd.isEmpty() || cmd[0]=='#') return true;
QStringList params = QStringList::split(" ",cmd.simplifyWhiteSpace());
std::cout << cmd << "\n";
bool succ=false;
if(params[0]=="exit")
{
exit(0);
//app.quit();
return true;
}else if(params[0]=="include")
{
succ = load(params[1]);
}else if(params[0]=="read")
{
QString what = params[1];
if(what=="lef"){succ = lef.load(params[2]);if(succ) hasLefDef =
true;}
else if(what=="def"){succ = def.load(params[2]);if(succ)
hasLefDef = true;}
else if(what=="str"){succ = str.load(params[2]);if(succ) hasCct
= true;}
else if(what=="lib"){succ = lib.load(params[2]);if(succ) hasCct
= true;}
else if(what=="pla"){succ = pla.load(params[2]);if(succ) hasCct
= true;}
else if(what=="net"){succ = netlist.load(params[2]);if(succ)
hasCct = true;}
else if(what=="dsn"){succ = dsn.load(params[2]);if(succ) hasCct
= true;}
else if(what=="tim"){succ = tim.load(params[2]);if(succ) hasTim
= true;}
else if(what=="timing"){succ = timing.load(params[2]);if(succ)
hasTiming = true;}
else if(what=="rte"){ succ = rte.load(params[2]);}
}else if(params[0]=="write")
{
QString what = params[1];
if(what=="lef")  succ = lef.save(params[2]);
else if(what=="def") succ = def.save(params[2]);
else if(what=="str") succ = str.save(params[2]);
else if(what=="lib") succ = lib.save(params[2]);
else if(what=="pla") succ = pla.save(params[2]);
else if(what=="net") succ = netlist.save(params[2]);
else if(what=="dsn") succ = dsn.save(params[2]);
else if(what=="tim") succ = tim.save(params[2]);
else if(what=="timing") succ = timing.save(params[2]);
else if(what=="block") succ = route.saveBlk(params[2]);
else if(what=="rte") succ = route.rteWrite(params[2]);
else if(what=="grte") succ = route.grteWrite(params[2]);
else if(what=="flw") succ = route.saveFlow(params[2]);
else if(what=="pic") succ = route.savePic(params[2], params[3]);
}else if(params[0]=="report")
{
QString what = params[1];
if(what=="lefdefcong") succ = util.lefdefCongWrite(params[2]);
else if(what=="cctcong") succ = util.cctCongWrite(params[2]);
else if(what=="conglefdef") succ =
util.conglefdefWrite(params[2]);
else if(what=="congcct") succ = util.congcctWrite(params[2]);
else if(what=="length") succ = util.lengthWrite(params[2]);
else if(what=="ccttim") succ = util.cctTimWrite(params[2]);
}else if(params[0]=="set")
{
QString what = params[1];
if(what=="diearea")
{
double x1 = params[2].toDouble();
double y1 = params[3].toDouble();
double x2 = params[4].toDouble();
double y2 = params[5].toDouble();
route.dieArea(x1,y1,x2,y2);
succ = true;
}else if(what=="gcell")
{
double width = params[2].toDouble();
double height = params[3].toDouble();
route.gCell(width,height);
succ = true;
}else if(what=="gtile")
{
int width = params[2].toInt();
int height = params[3].toInt();
route.gTile(width,height);
succ = true;
}else if(what=="order")
{
int order = params[2].toInt();
route.order(order);
succ = true;
}else if(what=="lump")
{
double lump = params[2].toDouble();
route.lump(lump);
succ=true;
}else if(what=="driver")
{
double res = params[2].toDouble();
double cap = params[3].toDouble();
route.driver(res,cap);
succ = true;
}else if(what=="buffer")
{
double load = params[2].toDouble();
double res = params[3].toDouble();
double cap = params[4].toDouble();
route.buffer(load,res,cap);
succ = true;
}else if(what=="load")
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
{
double load = params[2].toDouble();
route.load(load);
succ = true;
}else if(what=="layer")
{
QString layer = params[2];
QString param = params[3];
CLayer* pLayer = route.layer(layer);
if(pLayer!=NULL)
{
if(param=="dir")
pLayer->dir(::layerDirection(params[4]));
else if(param=="pitch")
pLayer->pitch(params[4].toDouble());
else if(param=="res")
pLayer->res(params[4].toDouble());
else if(param=="cap")
pLayer->cap(params[4].toDouble());
else if(param=="usage")
{
pLayer->len(params[4].toDouble());
pLayer->bUsage(params[5].toDouble());
pLayer->nUsage(0);
}else if(param=="cellBlock")
route.addCellBlock(layer, params[4] + " " + params[5] + " " + params[6]);
else if(param=="tileBlock")
route.addTileBlock(layer, params[4] + " " + params[5] + " " + params[6]);
succ = true;
}else succ = false;
}else if(what=="maze")
{
QString param = params[2];
if(param=="dir_change_cost")
route.dirChangeCost(params[3].toInt());
else if(param=="layer_change_cost")
route.layerChangeCost(params[3].toInt());
else if(param=="step_cost")
route.stepCost(params[3].toInt());
else if(param=="wrong_dir_cost")
route.wrongDirCost(params[3].toInt());
}
}else if(params[0]=="select")
{
QString what = params[1];
if(what=="layer")
{
QString layer = params[2];
route.selectLayer(layer);
succ = true;
}
}else if(params[0]=="unselect")
{
QString what = params[1];
if(what=="net")
{
QString net = params[2];
route.unselectNet(net);
succ = true;
}else if( what=="layer")
{
QString layer = params[2];
route.unselectLayer(layer);
succ=true;
}
}else if(params[0]=="init")
{
succ = route.init();

}else if(params[0]=="gen_rnd_tim")
{
succ = route.gen_rnd_tim();
}else if(params[0]=="groute")
{
succ = true;
int numIter = 1;if(params.count()>1) numIter =
params[1].toInt();
for(int i=0;i<numIter;i++) if(!route.route()) {succ =
false;break;}
}
if(!succ) std::cout << "WARNING : Failed to execute, " << cmd << "\n";
return succ;
}
```