DISSERTATION

A HYBRID MODEL CHECKING APPROACH TO ANALYSING RULE CONFORMANCE APPLIED TO HIPAA PRIVACY RULES

Submitted by

Phillipa Bennett

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2017

Doctoral Committee:

Advisor: James Bieman
Co-Advisor: Geri Georg

Sudipto Ghosh
Daniel Turk

ABSTRACT

A HYBRID MODEL CHECKING APPROACH TO ANALYSING RULE CONFORMANCE APPLIED TO HIPAA PRIVACY RULES

Many of today's computing systems must show evidence of conformance to rules. The rules may come from business protocol choices or from multi-jurisdictional sources. Some examples are the rules that come from the regulations in the Health Insurance Portability and Accountability Act (HIPAA) protecting the privacy of patient information and the Family Educational Rights and Privacy Act (FERPA) protecting the privacy of student education records. The rules impose additional requirements on already complex systems, and rigorous analysis is needed to show that any system implementing the rules exhibit conformance. If the analysis finds that a rule is not satisfied, we adjudge that the system fails conformance analysis and that it contains a fault, and this fault must be located in the system and fixed.

The exhaustive analysis performed by *Model Checking* makes it suitable for showing that systems satisfy conformance rules. Conformance rules may be viewed in two, sometimes overlapping, categories: *process-aware* conformance rules that dictate process sequencing, and *data-aware* conformance rules that dictate acceptable system states. Where conformance rules relate to privacy, the analysis performed in model checking requires the examination of fine-grained structural details in the system state for showing conformance to data-aware conformance rules. The analysis of these rules may cause model checking to be intractable due to a state space explosion when there are too many system states or too many details in a system state. To overcome this intractable complexity, various abstraction techniques have been proposed that achieve a smaller abstracted system state model that is more amenable to model checking. These abstraction techniques are not useful when the abstractions hide the details necessary to verify conformance. If non-conformance occurs, the abstraction may not allow isolation of the fault. In this dissertation, we introduce a Hybrid Model Checking Approach (HMCA) to analyse a system for both process- and data-aware conformance rules without abstracting the details from a system's detailed process- and data models.

Model Checking requires an analysable model of the system under analysis called a *program graph* and a representation of the rules that can be checked on the program graph. In our approach, we use connections between a process-oriented (e.g. a Unified Modelling Language (UML) activity model) and a data-oriented (e.g. UML class model) to create a unified paths-and-state system model. We represent this unified model as a UML state machine. The rule-relevant part of the state machine along with a graph-oriented formalism of the rules are the inputs to HMCA. The model checker uses an exhaustive *unfolding* of the program graph to produce a *transition system* showing all the program graph's reachable paths and states. Intractable complexity during model checking is encountered when trying to create the transition system. In HMCA, we use a divide and conquer approach that applies a slicing technique on the program graph to semi-automatically produce the transition system by analysing each slice individually, and composing its result with the results from other slices. Our ability to construct the transition system from the slices relieves a traditional model checker of that step. We then return to use model checking techniques to verify whether the transition system satisfies the rules. Since the analysis involves examining system states, if any of the rules are not satisfied, we can isolate the specific location of the fault from the details contained in the slices.

We demonstrate our technique on an instance of a medical research system whose requirements include the privacy rules mandated by HIPAA. Our technique found seeded faults for common mistakes in logic that led to non-conformance and underspecification leading to conflicts of interests in personnel relationships.

ACKNOWLEDGEMENTS

I remember when I was in high school, in sixth form as it is in the Caribbean, I decided that I would pursue getting a Ph.D. As I write this acknowledgement, on the verge of defending this dissertation, I am thankful for the fulfilment of this dream. Of course, I had a lot of inspiration along the way. For theis inspiration, I would like to say to:

- My mother Viviene Bennett, thank you for your inspiration and unswerving devotion. You are my greatest cheerleader.

- My high school math teacher, Ms. Allen who taught me from first through fifth form, thank you for teaching math in such a way that I loved it.

- My high school sixth form geography teacher (whose name I have now forgotten) who inspired me to dream this dream.

- Dr. Ezra Mugisa, affectionately known as Doc, at the University of the West Indies, Mona, where I obtained my undergraduate and masters degrees, thank you for your continued mentorship.

- My previous advisor here at Colorado State University, Dr. Robert France who passed away during my studies, I am forever indebted to you for your patience and mentoring.

- My current advisors Prof. James Bieman and Dr. Geri Georg, you both have encouraged me in more ways than in dissertation writing. I have many life lessons from relating with you.

I appreciate you all.

DEDICATION

To Moms, who stayed with her children in spite of the lure to greener pastures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

1. INTRODUCTION

Conformance analysis in systems can be non-trivial because of system size and of the complex interplay of conformance requirements from different sources. The requirements are imposed through rules that stem from business protocol choices or from legal and standards regulations. Examples of standards and regulations include the Health Insurance Portability and Accountability Act (HIPAA) [2, 3] and the Gramm-Leach-Bliley Act (GLBA) [77] that apply to the privacy of non-public health information and financial information respectively. A system that is governed either by HIPAA or by GLBA must not only show conformance to the format of information shared with others, but also to the processes accessing and updating the information. For example, the HIPAA regulations tells us:

1. that totally de-identified patient health information may be shared with researchers;
2. that total de-identification means that the patient's health information does not contain any data that can be used to identify or link to other data sets to identify the patient; and
3. which pieces of data can identify a patient.

These ensure that the patient's privacy is protected, while still allowing researchers access to medical records for conducting research.

1.1 Conformance Analysis in Practice

The main approach to conformance analysis has been to use *model checking* [17, 27, 33, 32, 52, 60, 62, 58, 59, 71]. Model checking is an exhaustive model-based verification technique [14] that relies on having an abstraction of a system represented as a *program graph* that is a unified representation for both order of process execution on paths and system structural (hereafter referred to as *state*) changes along the paths. The model checker uses an exhaustive *unfolding* of the program graph to produce a *transition system* showing all the program graph's reachable paths and states. Properties that depend on the sequencing or occurrence of system processes on paths and/or states are then verified on the transition system. The model checker can tell us whether 1) a property is satisfied, 2) it is not satisfied by producing a counterexample from the transition system, and 3) in some cases that it is neither possible to prove nor disprove a property. For the last situation, a model checker may not be able to give a definite answer due to space and time complexities or insufficient detail in the model. While the latter situation will not be examined as a part of this research, we note that:

1. if the model checker is not able to give an answer due to space and time complexities, it may mean the program graph need to be represented more abstractly (and details needed to show conformance may be los)t;

2. if there is insufficient detail in the model, it may mean that the property cannot be verified using the abstractions in the model and both the program graph and/or the property may need revisions; and

3. after the revisions the property may need to be reanalysed.

1.2 Challenges in Conformance Analysis

In order to use model checking in conformance analysis, we identify six challenges below.

1. *Rule Representation*. The rules are often published informally or in legal terms and are not understandable by automated systems [27, 30, 40]. Any effort to show conformance to these rules requires a language for representing the rules to be used as input for conformance analysis. In addition, since it is the actions executed and the corresponding system state changes that are analysed to determine rule conformance, we must be able to define the rules based on the observable actions and states shown in the transition system.

2. *Changing Rules*. Changed rules [61] represent changed contexts in which to show conformance. For example, data mining techniques may be used to search for new relationships or linkages in data. If the newly revealed relationships can be used to make inferences and potentially identify subjects [12], then, especially where conformance rules address privacy concerns, rule in the system must be changed. To address this challenge, changes made to conformance rules must be re-verified on a system that previously passed conformance. In addition, applying versioning to a rule base may be important in identifying the version of a rule to which a system conforms.

3. *Rule Types.* Conformance rules may be *process-aware* [21, 58, 61] and/or *data-aware* [27, 52]. Process-aware rules are defined using regular patterns on the sequencing of processes, while data-aware rules are defined using system states to say what conformance means. Conformance analysis for each type of rule has different requirements. For example, in model checking showing conformance to data-aware rules may require additional computations and sophisticated techniques than those required for checking process-aware rules to handle large state spaces. This is because the number of processes may be considerably smaller than the size of the state space in a concrete system representation, so process-aware rules may have far less computational requirements or require less sophisticated techniques. To demonstrate conformance therefore, a requirement in this challenge is to be able to show conformance to data-aware rules without the need to use abstractions of details.

4. *System Complexity.* For conformance analysis, the complexity of a system may depend on whether the system enforces a large set of rules that may have interdependencies and conflicts, and/or a large amount of data with complex relationships. Where rules conflict, we may need additional mechanisms to prioritise conformance rule satisfaction. Such prioritisation may also be used to provide metrics that measure system conformance levels [61]. In addition, different system abstractions (representations) may hide the system complexity. For example, showing conformance on implementations reveals an aspect of complexity - unbounded and/or unanticipated executions paths and system state. These aspects may not be encountered when showing conformance on system design specifications because designs may not fully capture all of the ways software will be used. In addition, showing system conformance *a priori* by examining requirements, designs or at run-time, or *a posteriori* by examining audit logs of system executions against a model of what is expected, may all be important in system conformance analysis.

5. *Hidden Paths.* Conformance analysis failure, i.e., non-conformance to rules, due to privacy leaks occur because of the presence of hidden paths in the system. Hidden paths may exist when a system is used in non-standard ways because loopholes exist in the system, or when rules that should change in response to new and/or changed functionalities in the system are not changed. The fifth challenge therefore is to be able to identify such potential and preventable rule violations [62].

6. *Model Granularity and Analysis Results.* We need to present meaningful analysis results so that rule conformance failure can be properly linked to specific system actions and states [13, 52, 61]. For this, we need a system representation with enough granularity to find and isolate the fault causing the failure.

Showing conformance to process-aware rules is one of the strengths of model checking [58, 61, 85, 86] because large, i.e. high-level, abstractions can be applied to the states resulting in smaller computation and memory requirements. On the other hand, showing conformance to data-aware rules may cause the model checker to hang because of a state space explosion when there are too many states or details of states to consider. Conformance analysis has been successfully demonstrated for process-aware rules [61]. Approaches for showing conformance to data-aware rules may be less successful since they also employ large abstractions for the system states to overcome the intractable complexities [27, 52]. In situations where showing satisfaction to data-aware rules require analysing detailed and/or concrete system states, applying abstractions is not a feasible solution in conformance analysis because the abstractions hide the very details needed to check conformance. In order to handle the analysis of detailed system states when verifying data-aware rules this dissertation proposes and outlines a *hybrid model checking approach* (HMCA) for rule conformance analysis (RCA).

1.3 Hybrid Model Checking Approach (HMCA) to Conformance Analysis

HMCA is proposed for use in RCA to overcome the intractable analysis of current model checking tools when checking data-aware rules. We propose HMCA as a hybrid approach because it:

1. offers exhaustive analysis within a certain scope; and

2. does not use current model checking tools, but proposes the use of other modeling and analysis tools.

As with model checking, RCA using HMCA consists of constructing models, including conformance rule representations, then analysing the models, and finally providing feedback for conformance rule violations. We discuss an overview of our model construction in Section 1.3.1, analysis in Section 1.3.2, and providing feedback in Section 1.3.3. In Section 1.3.4 we outline the contributions HMCA makes in reference to the challenges discussed in Section 1.2.

1.3.1 Model Rule Conformance in terms of Model Checking

In order to use model checking in RCA, we need to construct an analysable system program graph and conformance rule representation.

1.3.1.1 Construct Program graph

For constructing our analysable system program graph, we start with both a UML activity model as a representation of (the human system interactions) as paths in the system, and a UML design class model where the operations have pre- and post conditions as a representation of the overall system state. Both these models are constructed using details provided by a domain expert for the system under analysis and by an analyst with expertise in constructing the models.

We propose a technique to add details to the the activity diagram by associating it with details from the class diagram to produce an *annotated activity model*. We then transform the annotated activity model to a UML system state machine model as the latter closely represents the semantics of transition systems [10, 9, 31, 35, 37, 42, 43, 73]. However, depending on the size of the initial activity model, its transformation may produce a complex state machine. We observe that in many cases, showing that the system conforms to a rule only requires detailed examination of some of the operations in the system state machine. In this case we may decompose the transformed system state machine to produce smaller state machine views that are rule-specific. Since rules always examine a target, i.e., states of objects of interest, we must first identify that target. These objects of interest are the focus of the decomposed state machine views, hereafter called the *entity views*. These entity views include operations that are extracted from the system state machine and states that are abstract descriptions of the system state from the class model related to the object of interest.

Each rule may reference the operations and states in more than one entity views, so our interest will be to analyse the entity views applicable to the rule. We therefore create a *rule-specific entity view* that is a single entity view, or the composition of more than one individual entity views that represents the set of all the operations and parts of the system state required to show conformance to the rule. We use each rule-specific entity view as a program graph for HMCA to analyse.

1.3.1.2 Construct Conformance Rule Representation

We use a graph formalism, a non-deterministic finite automaton (NFA) [14, see Chapter 4], to define conformance rules based on the elements in each rule-specific entity view, i.e., the sequencing or occurrence of operations and states to define process- and data-aware rules respectively.

1.3.2 Conformance Analysis

We analyse each rule-specific entity view to determine rule conformance to it applicable rule. However, our analysis of data-aware rules means that we are likely to encounter intractable complexity because we will not apply abstractions to the state beyond those used in the class diagram. We therefore construct a transition system using the following steps.

1. *Model reduction.* Recall that intractable complexity is usually encountered when the state space being analysed is too large. Model reduction techniques, such as model slicing [81, 82] for UML class and object models has been shown to reduce space and time complexities in model analysis for structural and operational constraints. Therefore, we adapt this technique to slice our class diagram according to operations, i.e., we create smaller class models, each containing the class model elements referenced in each operation in the rule-specific entity view. Each small class model is called a slice. This allows us to produce smaller state analysis sub-problems.

2. *Local analysis, i.e., slice analysis.* We use each slice as an intermediate model that we analyse in a semi-automated way. We transform each slice to an equivalent Alloy language specification [1, 47, 49]. We then use the Alloy Analyser, whose strength is in analysing structural models within a certain scope to determine potential final states.

3. *Construct Transition System.* We use the states from each slice to construct the transition system.

4. *Evaluate Property on the Transition System.* We check whether the NFA rule representation is satisfied in the transition system using model checking techniques.

1.3.3 Provide Feedback

If the transition system shows a rule violation, then this represents a failure in the system. In this case, we identify the slice of the class model in which the non-conformance occurs, and extract from it the evidence of a fault in the system. We pinpoint the location of the fault to aid in fixing the fault.

1.3.4 Addressing Challenges and HMCA Contributions

HMCA makes the following contributions by addressing the challenges in Section 1.2; each numbered item corresponds to the same numbered challenge:

1. Construct *rule representation* from details in entity views - we construct conformance rule representations so that they can be checked on our system program graph. The emphasis here is that we can represent the conformance rules using elements from our system models. The details are in sections 5.1.3, 6.2, and 7.1.2.

2. Analyse *changed rules* - HMCA can be used in the analysis of changed rules and for metrics to judge the level of system conformance. However we relegate specific techniques to streamline analysis of changed rules to future work.

3. Analyse and get results for *rule types* - the focus is on being able to analyse data-aware rules at the level of detail required where current model checking tools fail. We analyse data-aware rules using slicing techniques on the program graph to create the transition system. The details are in in sections 5.2, 6.3, and 7.1.4.

4. No need to apply large abstractions to handle *system complexity* - we handle a large amount of data in the state by using entity view as decompositions of the system state machine. The details are in sections 5.1.2, 6.2, and 7.1.1

5. Finding *hidden paths* - HMCA can provide hidden path analysis by examining how:

   (a) the results from slices may be recombined to create paths not documented in the system activity diagram; and

(b) the segments or elements of the class model that are not used in the analysis of any rule and whether these segments can lead to rule violations because they create a way to traverse a path not checked by the rules under analysis.

However we relegate specific techniques to to find hidden paths to future work.

6. *Model Granularity and Analysis Results* - We provide meaningful and useful feedback on faults that cause rule conformance failures by using connections between activity and class models. The details are in sections 5.3, 6.4, and 7.2 through 7.6.

1.4 Evaluation

Our evaluation of HMCA involves examining a real world system possessing all the challenges outlined in Section 1.2. For this, we use the National Jewish Health (NJH) medical research system (see Section 3.4 for more details) where conformance rules come from the Health Insurance and Portability Act 1996 (HIPAA). It is important that systems like those at the NJH undergo conformance analysis because the penalties for non-conformance are severe, and the public's perception of the trustworthiness of the organisations involved in a rule violation can decline.

HIPAA rules include both process- and data-aware rules. For example, HIPAA mandates that to control privacy leaks, health information maintained and stored by an organisation should only be shared with (trusted) associates. In the NJH's system, this is enforced as a process-aware rule to verify that researchers are qualified, or have been approved through a qualification process, before they are allowed to apply for specific permissions for accessing patient health information. In work prior [23] to this dissertation, we examined how a system-wide state machine view of the NJH system can allow us to verify conformance to process-aware rules. This view required using large abstractions to represent the system state, and could become complex without abstractions when verifying data-aware rules.

Since HIPAA rules also mandate the formats of patient health information that is shared, the NJH system must enforce them using data-aware rules. We will examine two of these rules. The first requires that shared patient health information has *all identifying data removed*, i.e., data is *de-identified* using the

HIPAA *deidentified* conformance rule[1]. The second requires that that shared patient health information has *no identifying data removed*, i.e., data is *identified*. For these, HIPAA outlines the specific data about a patient that can be used to identify a patient. Within the scope of both these conformance rules, we must consider the ways that data for special populations, i.e., pregnant women and neonates, prisoners, and children, may be shared. This is because these special patient populations that have additional rules that further protect their privacy. In our evaluation we included the children protected population.

Firstly, we provide an initial demonstration of HMCA by constructing, analysing, and providing feedback on conformance rule violations related to HIPAA *de-identified* conformance rule. The details are in Chapter 5 and Sections 7.2 through 7.6.

Secondly, we include the HIPAA *identified* conformance rule and evaluate *how well HMCA is able to find faults*:

1. through fault seeding - first by inserting a logic error in a rule and second by adding a transition to the system state machine that is not considered in the rule; and

2. highlight that these seeded faults correspond to real-world problems - the logic fault causes non-conformance to the previously verified HIPAA de-identified conformance rule and the second fault causes conflicts of interest.

The details of this validation is in Chapter 8.

Finally, we model the rules governing access to data for children as this is important to the NJH and we argue that doing so will work as a proof of concept for the other protected populations. The details are in Chapter 9.

1.5 Document Organisation

We describe related work in Chapter 2, and background tools and techniques, and give more details on the NJH system in Chapter 3. In Chapter 4 we provide motivation for HMCA by examining RCA in specific model checking tools to show how intractable complexity results when using design class models. In addition to the specifics of HMCA already highlighted in sections 1.3.4 and 1.4, we describe how HMCA is applied

[1]Since the focus of this dissertation is not on HIPAA regulations, we used simplifications of them in order to demonstrate HMCA.

to the NJH system in Chapter 5, we describe HMCA in Chapter 6, an expansion of the feedback stage of HMCA in Chapter 7, and the analysis for the children protected population in Chapter 9. In Chapter 10 we describe how HMCA can be applied in other domains requiring RCA. We follow with some insights that may be helpful when applying HMCA in Chapter 11. Our final chapter, Chapter 12, gives our conclusions and future directions.

2. RELATED WORK

In this chapter, we summarise related work in rule conformance analysis (RCA). The key areas of discussion are (1) the approaches to RCA in Section 2.1, (2) the types of conformance rules in Section 2.2 , and (3) how conformance rules are specified in Section 2.3. We give a summary and some open problems in Section 2.4.

In our discussion compliance and conformance are interchangeable concepts, and each is used based on their use by the authors. However, in the rest of the dissertation we will use conformance.

2.1 RCA Approaches

2.1.1 General Complexity Handling in RCA

In this section we describe approaches to RCA that uses decomposition and/or distributed processing to handle the complexity in RCA.

2.1.1.1 Odessa

Montanari et al. [67] developed the Odessa environment that uses network distribution in RCA. The rules being analysed are specified in a security policy and the data needed for each rule may be distributed on different servers to the network. The observation in Odessa is that rule parts may be analysed at the server where the data they need are located. For example, given a rule

$rule1\colon\ r1 \rightarrow r2$

where $\rightarrow$ means implies, then if *r1* can be checked on server *S1* and *r2* on server *S2*, then we can assign *r1* to *S1* and send *r1*'s results to *S2*; *S2* then evaluates *r2* and *rule1*. Since *S1* and *S2* evaluate parts of *rule1*, they are assigned to a group called a *predicate* group. In addition, there may also be replication of data on separate servers, so for resilience *rule1* may also be evaluated in another predicate group. The distribution of the rule parts to different servers and rules to different groups enables evaluation of large-scale policies.

2.1.1.2 System Logs and Petri Net Decomposition

Broucke et al. [85] describe an approach to RCA that compares event logs that are replayed as streams against a system expected behaviour that is modelled as a Petri Net. To address scalability, the Petri Net is decomposed into subprocesses. Each log is then replayed and matched to subprocess(es) that are enabled by events in the Petri Net. An event that cannot be executed on the sub-model may identify an illegal or missing process.

2.1.2 Bottlenecks in Weak and Strong Conformance

In addition to their specification language (see Section 2.3.3), Chowdhury et al. [27] provide a demonstration that system actions show weak or strong conformance to the encoded rules. Their notions of *weak conformance* and *strong conformance* are redefinitions of those originally proposed by Barth et al.[17]. When applied to system actions, a contemplated action shows weak conformance to a policy if it does not violate the present requirements and can be checked on finite traces of past events. These requirements are specified using conjunctions and disjunctions and no future operators. An action shows strong conformance to a policy if its obligatory requirement is consistent with the current requirements and can be checked by concatenating a finite trace that fulfils weak conformance with an infinite trace satisfying the obligation. These requirements are specified using implications and future temporal operators. Strong conformance also means that the contemplated action neither prevents obligations nor causes unsatisfiable obligations. During analysis, the authors found that the policy became a bottleneck, so they propose slicing (see Section 3.3) based on obligations. However their slicing technique only reduces the bottlenecks if obligations do not depend on each other.

2.1.3 Compliance Monitoring and Conformance Checking

In order to analyse rules, another approach is to use *a priori* conformance monitoring [5, 13, 19, 18, 24, 26, 62, 63, 66, 71, 86] or *a posteriori* conformance checking [46, 76, 84, 85, 86]. Conformance rule monitoring (CRM) requires continuous polling in systems to detect where rules are satisfied, violated, or violable so that measures could be taken to disallow rule violation [62] during execution. CRM also applies to checking designs for conformance rule violations [21]. Conformance rule checking (CRC) requires verifying that process logs conform to process models and rules, and includes having models of fitness to measure levels of conformance.

CRM approaches may be further identified as 1) automaton based monitoring, 2) logic based monitoring, or 3) violation pattern based monitoring according to the formalism used to specify the rules. Automaton based monitoring [63, 71] uses linear temporal logic (LTL) that is transformed to an automaton. In this approach patterns [32, 33] may be used to hide the complexity of LTL. Logic based monitoring [5, 66, 71] makes use of logic formalisms. Violation pattern based monitoring is used to query design models [21] and partial execution traces for rule violation patterns [13, 24, 50, 86].

Both CRM and CRC require exhaustive exploration using model checking to extract conformance evidence from the paths, states, events, or system logs (as seen in Section 2.1.1.1). While model checking has been the main method used in RCA, Petri Nets [72, 71, 85] have also been used. Petri Nets is a mathematical modelling language used to describe distributed systems and therefore can be used to easily model and analyse concurrent systems. Model checking on the other hand must use interleaving semantics to reason about concurrent systems executions. Model checking has good tool support, so petri-net practitioners have been using model checking techniques to analyse systems modelled as Petri Nets [53, 54, 55].

2.2 Process and Data-aware rules

RCA separates *process-aware* from *data-aware* conformance rules because of the different memory requirements for each type of rule. Knuplesch et al. [52] describe an approach to 1) identify and monitor individual activations of a conformance rule, 2) proactively prevent rule violations by using techniques that are able to identify rules that could be violated in the future, and 3) provide root cause identification in case of rule violations. A rule is modelled as a *compliance rule graph* and is instantiated each time a rule is to be checked. Monitoring is accomplished by using pattern matching of events in the compliance rule graph. In addition, events trigger an instance of compliance rule graphs for each applicable data item. Rule matching uses antecedents that wait for consequents: if consequents are observed then the rule is satisfied but the rule can also be violable if the consequent has an event that must not occur (checked on future events). The intervention to prevent violations is semi-automatic such that it can enable (and force execution of) events that should be observed or disable events that should not be observed before the process preceding the event ends. Since event observation is per rule activation, and rule activation is per applicable data item, rule enabling or disabling is used to provide feedback when violations occur.

While model checking is used in their RCA, Knuplesch et al. also identify that data-aware rules may cause a state-space explosion in a large domain. The authors propose to minimise the state explosion by:

1. applying an automatic pre-processing step that reduces concrete data values to abstraction classes based on the data values that appear in the rules - this step produces an abstract process model and abstract data-aware compliance rules;

2. using the abstract process model and the abstract data-aware compliance rules to perform conformance analysis and produce a conformance report; and

3. applying an automatic post-processing step that converts the abstract process model back to a concrete model - this only occurs where conformance rules have not been satisfied in order to provide user feedback.

The authors demonstrate this automatic pre- and post-processing for numerical data.

Ly et al. [59, 60, 62] propose the use of patterns as a compliance rule graph for specifying path rules based on activity occurrences and sequences using first-order logic. The patterns use precedence and antecedent activities that must happen, cannot happen, etc. This is in effect a language for specifying rules that is a simplification for the non-technical user, yet has formal semantics that can be analysed, similar to Dwyer's temporal patterns discussed in Section 2.3.2.1.

As an extension of Ly et al.[60], Ly et al. [58, 61] describe specifying and analysing both process-aware and data-aware conformance rules by supporting loop-free process models, using abstractions of data conditions from Knuplesch et al. [52]. They further state that data-aware rules include those where 1) process-aware rules include examining data, 2) rules imply that a data condition needs to be checked, and 3) data conditions are included directly in the rules.

2.3 Conformance Rules

This section describes approaches to specifying and/or analysing conformance rules.

2.3.1 Checklists in Rule Conformance

Checklists can show conformance to the *Standards for Safeguarding Customer Information* (Safeguards Rule) [36] and the *Volcker Rule* [34] for financial transactions as mandated by the Federal Trade Commission. The Safeguards Rule checklist allows companies to assess and address operational risks related to customer information. The Volcker Rule is an improvement to the Gramm-Leach-Bliley Act in relation to covered funds, investment activity and affiliated transactions.

Rashidi-Tabrizi et al. [74] describes a framework for expressing legal requirements for compliance as goals that includes decomposing, attaching importance, conditions and exceptions. The framework uses the *Goal-oriented Requirement Language* (GRL) to formalise legal text in order to make it amenable to conformance analysis. This formalisation yields a goal model. The framework can be used as an analysis tool when auditing a system for compliance based on answering questions. Conformance is given a 100 value if compliant, and otherwise a value of 0-99 that indicates the level of compliance. The framework is not mapped to a system implementation so it is in effect a checklist.

Though checklists are important, they do not enable us to extract evidence of conformance from computerised systems that support an organisation's operations. However they may be more understandable than their corresponding laws and regulations and may be useful as requirements for specifying more formalised conformance rules.

2.3.2 Generalised Rule Specifications

2.3.2.1 Dwyer's Patterns

Conformance rules based on actions or a sequence of actions may be specified using first order logic (FOL) and first order temporal logic (FOTL) based on Dwyer's patterns [32, 33]. Dwyer identifies that the main hindrance to specifying and using tools that analyse a system of paths may be unfamiliarity with specifications, specification notations, and specification strategies. Dwyer proposes eight common patterns based on temporal logic. We summarise them in Table 2.1. For example, in a system that uses permissions to restrict access to sensitive data, an applicable rule is that permissions must be granted prior to access. In this case, we use the *Precedence* pattern in the *Order* pattern group to specify that permission approval must always precede the access. Every path in the system that accesses data must be shown to satisfy this rule in order for the system to show conformance to the rule. While the example given uses actions, the patterns may also be used with states and events.

Table 2.1: Dwyer's Patterns [32, 33] for Specifying Conformance Rules, adapted.

Pattern Group	**Pattern Name**	**Pattern Description**
Occurrence	Absence	A given system *state/action/event* does not occur within a scope.
	Existence	A given system *state/action/event* must occur within a scope.
	Universality	A given system *state/action/event* must exist throughout a scope.
	Bounded Existence	A given system *state/action/event* must occur k times within a scope.
Order	Precedence	A system *state/action/event* P must always be preceded by another system state/action/event Q within a scope.
	Response	A system *state/action/event* P must always be followed by another system *state/action/event* Q within a scope.
	Chain Precedence	A sequence of system *states/actions/events* $P_1, \ldots, P_n$ must always be preceded by a sequence of system *states/actions/events* $Q_1, , \ldots, Q_n$.
	Chain Response	A sequence of system *states/actions/events* $P_1, \ldots, P_n$ must always be followed by a sequence of system *states/actions/events* $Q_1, \ldots, Q_n$.

2.3.2.2 *Reference Architectures as Rules*

Buchgeher and WeinReich [26] propose focusing on the reuse of reference architectures in conformance analysis. A reference architecture is a set of rules that consist of roles together with the constraints on the roles and role relationships for a particular domain. RCA is made possible when a system realises the reference architecture and inherits its rules. The realisation involves making bindings from the architecture to specific roles in the actual implementation. This allows evaluation of the reference architecture rules.

2.3.3 Formal Languages to Encode Legal Requirements.

The following are examples of formal languages to encode rules from laws and regulations:

1. May et al. [65] present a formalism, called *Privacy APIs*, to encode HIPAA 2000 and 2003 consent rules which relate to when health care providers must obtain patient consent before performing treatment, payment, and other activities related to health care operations. HIPAA 2003 consent rules are a simplification of the HIPAA 2000 consent rules. After encoding the rules, May et al. convert their

formalism into the specification language of the SPIN model checker and check whether the formalism satisfies desired invariants as well as to explore the differences between the two versions of the rules.

2. Barth et al. [17] propose *C1*, a language for specifying policies based on a fixed set of predefined predicates using propositional linear temporal logic (pLTL).

3. Basin et al. [18, 19] use metric first-order temporal logic to specify rules, and they also developed a monitoring algorithm for the rules.

4. DeYoung et al. [30] develop an improvement to *C1*, a policy language called *PrivacyLFP* as a specification language for HIPAA and GLBA.

5. Garg et al. [38] propose a first-order logic-based privacy policy specification language that can encode HIPAA policies. They present an auditing algorithm that incrementally inspects the system log against a policy and detects violations.

6. Chowdhury et al. [27] propose a policy (rule) specification language based on first-order temporal logic as an improvement over *C1* that they use to encode all 84 disclosure-related clauses of HIPAA.

7. Becker et al. [20, 21, 22, 29, 80] describe a business process graph-based query language and matching algorithm. The query language is pattern based and can be used to specify infringement patterns, legal requirement identification patterns, risk management patterns, and change management patterns. The checking algorithm is able to analyse for conformance rule violations for all these patterns. The query language is applicable to arbitrary graph-based modelling languages for both simple and complex conformance rules.

2.4 Summary and Open Problems

The results of our related work are summarised as a matrix in Table 2.2.

RCA uses either a conformance rule monitoring approach or a conformance rule checking approach. Each RCA approach considers process-aware rules and/or data-aware rules with only two of the approaches considering data-aware rules. This is due to the additional considerations required to minimise intractable

Table 2.2: Related Work Summary

TABLE NOTES
CRC - Conformance Rule Checking
CRM - Conformance Rule Monitoring
DAR - Data Aware Rule
MC - Model Checking
PAR - Process Aware Rule
PN - Petri Net
Cell entries - Y means yes, an empty cell entry means no.

Reference	**Approach**		**Rule Formalism**		**How Rules Specified**			**Method**	
	CRM	CRC	PAR	DAR	Automaton	Logic	Pattern	MC	PN
Alberti et al. [5], Montali et al. [66]	Y					Y			
Awad & Weske[13], Birukou et al. [24]	Y						Y		
Barth et al. [17]	Y					Y		Y	
Basin et al. [18, 19]		Y				Y		Y	
Becker et al. [21]	Y						Y	Y	
Buchgeher & Weinreich [26]	Y						Y		
Chowdhury et al. [27]	Y		Y			Y		Y	
DeYoung et al.[30]	Y					Y		Y	
Dwyer et al. [32, 33]			Y	Y	Y	Y	Y		
Earnest & Young [34], FTC [36], Rashidi-Tabrizi et al. [74]		Y							
Garg et al. [38]	Y					Y		Y	
Huynh & Le [46], Rozinat et al. [76], Van der Aalst et al. [84]		Y							
Jacobsen et al. [50]	Y	Y			Y			Y	
Knuplesch et al. [52], Ly et al. [58]	Y		Y	Y			Y	Y	
Ly et al. [59, 60, 61, 62]	Y		Y				Y	Y	
Maggi et al. [63]	Y					Y			
May et al. [65]	Y		Y					Y	
Montanari et al. [67]	Y								
Pesic & Van der Aalst [71]	Y				Y	Y			Y
Broucke et al. [85]		Y							Y
Weidlich et al. [86]		Y	Y				Y		

complexity when the state explosion problem occurs for data-aware rules. While different specification formalisms exist for specifying rules, all the formalisms outlined are useful in both conformance rule monitoring and conformance rule checking. More approaches use model checking than Petri Nets for RCA.

Conformance reflects that a system adheres to governing rules. The rules are requirements that may be available before a system is developed and can be incorporated into the development process. A conformance rule monitoring approach may be used and instituted at design-time or at run-time. In contrast, conformance rule checking on system process logs allows one to check conformance in existing systems, or when rules are not available or included in the development process.

Conformance rules are based on laws and regulations and are usually not in a form that is analysable in computerised systems. Most of these rules are formalised using automata, logic, or patterns. Rules based on patterns may be the easiest for non-technical analysts to use. However, pattern-based rules may not be as expressive as the policies specified using automata and logic because automata and logic allow more fine-grained specifications. Further, rules specified using patterns must be transformed into more formal representations before analysis.

Both model checking and Petri Nets allow us to represent systems under conformance test and to perform exhaustive analysis to show rule satisfaction. Process-aware rules are easier to test because they often do not encounter the state explosion problem since process representations can be largely abstracted without loss of generality. However, where the details for showing conformance lie in examining detailed system structure, the analysis of data-aware rules using the large abstractions proposed are insufficient. We have seen that decomposition and/or network distribution have been used as a scalability technique in RCA. These techniques are also useful in minimising the state explosion problem.

One of the areas not examined in RCA is where hidden paths cause RCA to fail. Hidden paths may exist because path possibilities are not well understood or constrained. Representations may focus on the paths that are allowed and on restricting path possibilities that should not be allowed, however hidden paths in either of these categories may exist. While normal operations may not execute actions that constitute hidden paths, they may be started through other channels such as a backdoor into the system. Hidden path analysis is very important in evaluating systems for security leaks, particularly for network security algorithms that could violate privacy.

Another area of RCA not examined is rule interactions. There is often overlap in the elements of a system that analysis for each rule examines. In addition, concurrent activation of rules may mean that the same element instantiation is shared among rules, and sequencing of analysis may be important. In this case the approaches based on Petri Nets hold great promise for tractable and scalable analysis for rule interactions because we can adopt and adapt the techniques used to prove properties about interacting processes.

3. BACKGROUND

In this chapter, we give the background required to understand HMCA. We use both Alloy and model checking to specify and verify conformance to rules. Model slicing reduces the size of the models to be checked. Our evaluation applies HMCA to validate the NJH Research System against HIPAA rules.

3.1 Alloy

Alloy [1, 47, 49] is a formal specifications language that is described as a relational logic because it combines the quantifiers of first-order logic and the operators of relational calculus. In Alloy, a specification is made up of elements that are *atoms* and *relations*. Atoms are a modelling abstraction used to define entities and are indivisible, immutable, and uninterpreted. Relations, also called *fields*, define the relationships between two or more atoms. Both atoms and relations are viewed as, or a part of, *signatures* in the Alloy language. Constraints are included in the model as *facts*. *Predicates* are parameterised constraints that can be used to simulate instances of the model or as a part of other facts or other predicates. Though strictly not a part of the model, *assertions* may be used to define constraints that should follow from the facts.

Alloy is well supported by the Alloy Analyzer that has an embedded SAT-solver used to evaluate Alloy expressions. The Alloy Analyzer is able to simulate model examples of predicates or find counter-examples of assertions using user-defined scopes that are upper bounds on the number of each element.

Both predicates and assertions may be used to check invariants of UML class models [64, 81]. To do this, we use Alloy to specify an equivalent representation of the class diagram using atoms and relations. Any additional invariants from the class diagram that use Object Constraint Language (OCL) may be specified using facts, and object models may be specified using predicates. The Alloy Analyzer is able to check that a predicate is consistent with an Alloy model by generating an instance of the model according to the constraints in the predicate. In the case where we want to check that certain instances of a model are never possible, we use assertions. When simulating predicates, if an instance cannot be generated then we know that the object model defined by the predicate is inconsistent with the model. When checking assertions, if an instance, i.e. a counter-example, is found then we know that the object model defined by the assertion is inconsistent with the model.

Unlike model checking (see Section 3.2), the Alloy Analyzer is in the class of *model finder* tools because of its use of simulation to experiment with a restrictive set of scenarios compared to model checking that uses exhaustive exploration to verify properties. However, the Alloy Analyzer is still able to produce good results due to its reliance on the *small scope hypothesis* that justifies testing models with small scopes because a high proportion of faults may be uncovered when testing a program for all test inputs using small scopes [8, 48].

3.2 Model Checking

Model checking is a model-based verification technique [14] that performs an exhaustive brute-force exploration of system models to show that a property is satisfied. The system model describes how the system behaves, while the property specifies what the system should or should not do. The exploration performed in model checking examines all the possible states of a system in a systematic way to truly show that the properties are satisfied.

Both the system model of possible behaviours and the property of interest must be defined in a mathematically precise and unambiguous manner. The system model is often expressed as a Transition System (TS) model or as models whose executions may be transformed to TSs. For example, UML activity diagrams have semantics that are closely represented in transition system models [9, 35, 37, 42, 73, 31]. The properties of interest in this research are in the class of *safety properties* because they represent invariants in the system. These invariants may be described using a linear temporal property represented as a non-deterministic finite automaton (NFA) to be checked on the system model. We provide examples and further explanations of both the TS and the NFA in Chapter 5.

Model checking can have any of three outcomes: the property is valid in the model, the property is not valid in the model, or the memory required to enumerate the states in the model is larger that the physical limits of the computer's memory. If a state is encountered that violates a property, model checking uses simulation to replay the violation as a counter-example that shows how the behaviour is reachable in the system model. The simulation may also contain useful state information from the model that can be used to debug or adapt the model or property in order to reverify the property.

The outcome that exceeds the physical limits of the computer's memory requires that we revisit the model and apply abstraction techniques to reduce the state-space required. These abstractions must preserve the validity or non-validity of the properties. Alternatively, the abstractions may reduce the precision in the model and in the case of a property violation critical state information may be lost.

3.3 Slicing

Model Slicing [11, 25, 56, 57, 78, 79], analogous to program slicing [87], is a technique for decomposing models as a way to handle complexity in model analysis. The observation is that not all elements of a model are required for the analysis of each property (e.g. constraint). Therefore, we can create slices of a model that contain only those elements required for a local analysis. Slicing requires defining *slicing criteria* in order to perform the decomposition.

Our interest in slicing is specifically with the technique to slice UML class models and object models as described in Sun et al. [81, 82] as a way to promote scalable and rigorous analysis. In their work, a system represented using a class diagram is sliced by OCL invariants, or operation contracts written using the *Object Constraint Language* (OCL) [68].

The slicing allows each invariant and operation contract to be checked individually for scopes (using the Alloy Analyzer) well beyond those that would be allowed if the model were not sliced. Sun et al. also describe a technique to sequence slices for operations to check that invariants are not violated when operations are executed. In addition to the smaller memory requirement for analysing each slice, the authors also show that the technique significantly reduces analysis time and preserves analysis results such that the sliced models showed the same results as the unsliced model.

3.4 NJH System

NJH has a system, here after refered to as the NJH system, for sharing patient health information with researchers. This research system implements rules to maintain privacy of patient health information that stem from HIPAA regulations. In order to have access to patient health information from the NJH data sources, individual researchers or projects must first apply for, and have approved permission. Each approval defines pre-approved queries, and rules that dictate the format of the query results and whether the query results may only be viewed or if they can also be downloaded. The process of applying for a permission, setting up rules for each permission, querying the data sources using an approved permission, and delivering the query results according to a permission's predetermined format is used by NJH to help determine that it is conforming to the rules from HIPAA regulations. However, NJH currently does not verify HIPAA rules in their system, but assumes that their process is sufficient to satisfy them. In addition, their system uses a combination of manual and automatic steps in this process, so that automatically showing conformance to rules is not always possible. In order to automatically show conformance we need to describe the NJH system using formal techniques that create analysable models.

3.4.1 System Components of Interest

Structurally, the NJH system may be viewed in terms of the access control system that it implements, the patient information that it creates and manages, and the conformance rules it verifies. Specifically relating to HIPAA, the access control scheme assigns the following permissions to researchers and projects to access patient health information:

1. *Fishing License*: allows access to only *counts* of requested data.
2. *Prep License*: requested data is *viewable and not downloadable.*
3. *Access ticket*: data is *downloadable* according to any the following formats:
    (a) totally *De-identified* where
        i. columns having one of the 18 types of HIPAA-defined identifiers such as patient names are removed;
        ii. date columns are modified to show year only;
        iii. ages of 90 years or older are grouped into a single value;
        iv. geographic locations shown are only states or larger geographic subdivisions; and

v. geographic codes are modified to show only leftmost three digits of zip codes where the total population of those zip codes is $> 20,000$ or else display zip code 000.

The *De-identified* access ticket will hereafter be referred to as the *DeIDed* access ticket.

(b) *coded* or *linked* [70] where personal identifiers are substituted with codes so as to make them indirectly identifiable. This is different from anonymous, anonymised, or de-identified such that the link between the code and the personal identifier is maintained but not known to the researcher.

(c) a *limited data set* (LDS) where the following are removed from the query result

i. columns having one of the 15 types of *HIPAA LDS identifiers*, and

ii. any geographic locations smaller than town or city or zipcode.

(d) *identified* where the results are displayed without alteration.

Conformance rules include process-aware rules that specify that sequential processes are followed, e.g., application and approval before querying, and data-aware rules, e.g., patient health information in a query's result does not violate the kind of permission issued and used to execute the query.

4. MOTIVATING HMCA: NAïVE RCA

Since HMCA proposes to handle complexity using model slicing to decompose the analysis tasks, we will evaluate conformance analysis on unsliced models in order to highlight the limitations of current model checking/finding tools. Specifically, we will use the Spin model checking tool and the Alloy Analyzer model finding tool. We conducted analysis on process models of the NJH system using the UPPAAL [4] model checking tool [23]. However, this analysis was preliminary in a bid to understand process sequencing and interleaving for process-aware rules. We were able to verify the process models to be free of deadlock and not to violate any of the process-aware rules. We identified that RCA requires us to produce a more complete state model beyond the use of numeric symbolic representations in UPPAAL. The analysis we perform here will cover both data- and process-aware rules in a single model using the NJH system.

We discuss the design in Section 4.1 and the verification for the Alloy Analyzer and Spin in sections 4.2 and 4.3 respectively. We end this chapter with a discussion of the results and summary in Section 4.4.

4.1 Evaluation Design

4.1.1 Questions

From our evaluations of the tools, we wish to answer the following questions:

Question 1: What kinds of rules are best suited for each tool?

Question 2: What are the space and time measures when using the tools and how can we use these measures to motivate HMCA?

4.1.2 NJH System Operations and Data of Interest

For our analysis we will highlight operations where:

1. a researcher may:
   (a) apply to be qualified;
   (b) apply for a fishing licence; and

(c) execute queries using a fishing licence.

An approved Item 1a is the prerequisite for Item 1b, and an approved Item 1b is the prerequisite for Item 1c.

2. a project may:

   (a) apply for an access ticket; and

   (b) execute queries using an access ticket.

   In order to have an approved Item 2a, one of the requirements is that all the researchers assigned to the project must have an approved Item 1b, and an approved Item 2a is the prerequisite for Item 2b.

3. process-aware rules and data-aware rules are checked (see Section 4.1.3 below).

The UML class model supporting these operations is shown in Figure 4.1. It shows 61 classes, 26 associations, and 7 operations.

4.1.3 Rules

The rules of interest are to:

1. enforce operation sequencing for all the operation sequences implied in Section 4.1.2, e.g. a researcher has to be qualified before they can have a fishing licence approved;

2. check whether a query's result conforms to the required transformations, e.g. that the results are de-identified in accordance with the $DeIDed$ access ticket annotated with G in Figure 4.1; and

3. check whether a query's result conforms to additional rules, i.e., inclusion/exclusion based on patient consent.

The first is a process-aware rule rule and the others data-aware rules.

4.2 RCA using the Alloy Analyzer

4.2.1 Overview ofAlloy

Refer to Section 3.1 for a description of the Alloy Analyzer.

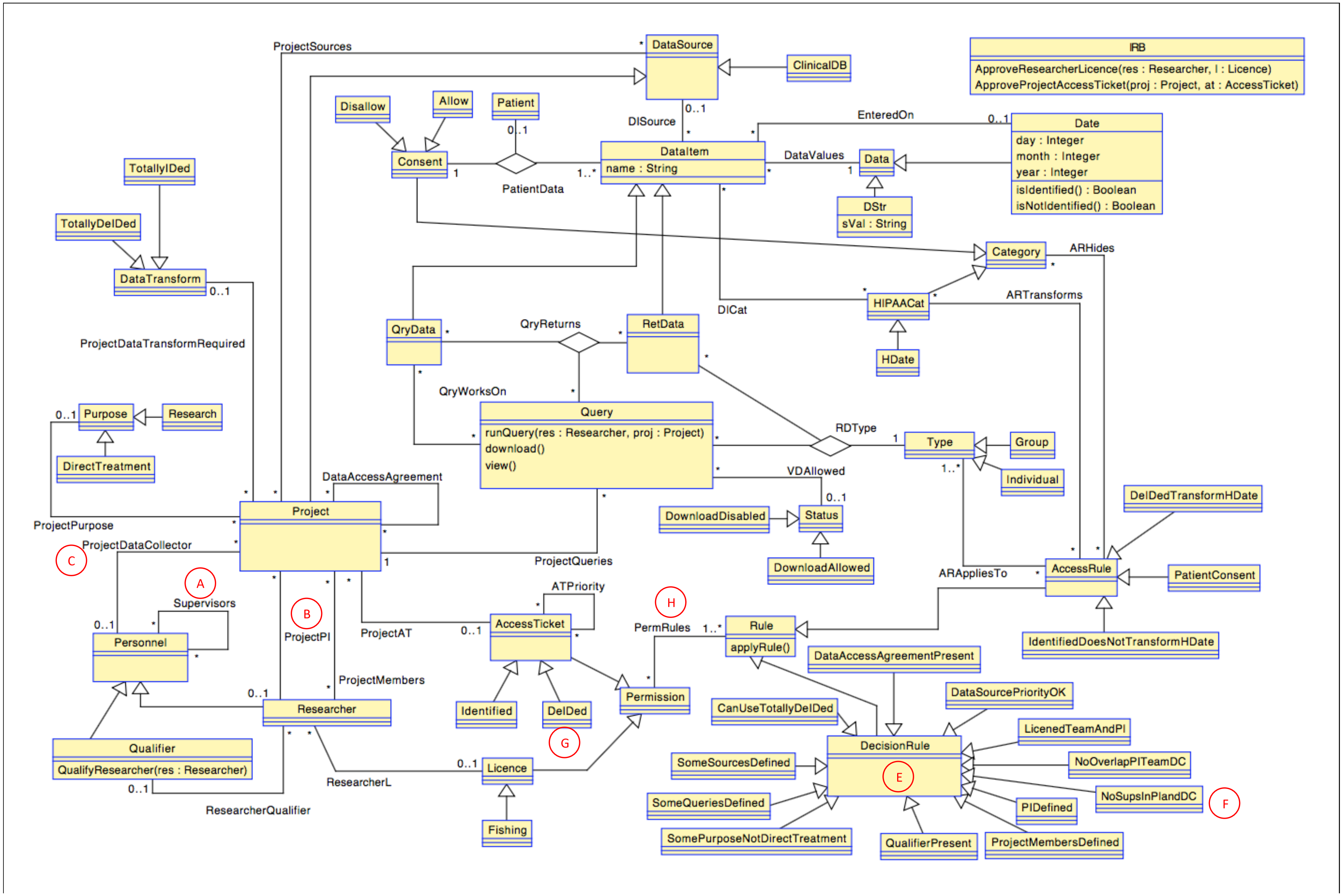


Figure 4.1: Class Model for the NJH system supporting the operations in Section 4.1.2

4.2.2 Alloy Specifications

We have created Alloy specifications to:

1. represent all the structural details in Figure 4.1;
2. include operation specifications for each of the operations in Section 4.1.2;
3. include assertions for process-aware ruless using Dwyer's *chain precedence* pattern to specify:
   (a) if the operation to approve a researcher's licence application is successful in the current state, then it must be that the operation to qualify the same researcher was successful prior to the operation to approve the licence;
   (b) if the operation to approve a project's access ticket application is successful in the current state, then it must be that all of its associated researchers have prior approved licences;
   (c) if the operation to execute any of a project's queries is successful in the current state, then it must be that the operation to approve the (same) project's access ticket was successful prior to the execution of the query; and
   (d) if the operation to check whether a query's return data conforms its associated project's access ticket is successful in the current state, then we know that the operations to execute the query was successful prior to the conformance check;
4. include as an assertion a data-aware rule using Dwyer's *absence* pattern to specify that no data that a query returns is identified when a *DeIDed* access ticket is used, and the converse, that if the data returned is de-identified then a *DeIDed* access ticket was used is also true.

The full Alloy model is in Appendix A.2.

4.2.3 Model Execution Results in the Alloy Analyzer

The Alloy Analyzer is limited to use 4GB of memory for analysis. This will have an impact on the scope for analysis and the time taken to perform the analysis. We show the analysis results in Table 4.1. The

Table 4.1: Verification Details for Alloy Predicates and Assertions in Table Notes

TABLE NOTES
The following IDs are used in the table to identify the predicates/assertions
[13] ltl_ApproveResLicenceAfterQualifyRes
[14] ltl_ProjectApproveAfterTeamAndPILicenceApprove1
[15] ltl_RunQueryAfterProjectApprove1
[16] ltl_UpdateConformanceAfterRunQuery
[17] Conformance

ID	**Variables**	**Primary Variables**	**Clauses**	**Time to Generate Variables and Clauses (h:mm:ss.s)**	**Result**	**Solving time (h:mm:ss.s)**	**Total Time to generate and solve (h:mm:ss.s)**
13	1425109	15039	3081002	0:05:02.444	No counterexample found	0:00:00.372	0:05:02.816
14	1415909	15039	3054122	0:02:50.326	No counterexample found	0:00:00.426	0:02:50.752
15	9325096	15039	19532096	6:58:20.917	No counterexample found	0:00:03.634	6:58:24.551
16	1157968	15039	3206000	6:47:47.352	No counterexample found	0:00:00.696	6:47:48.48
17	379078	15063	840237	0:00:06.596	No counterexample found	0:00:00.18	0:00:06.614
Total time to generate variables and clauses				13:54:07.635	Total Solving Time (h:mm:ss.s)		0:00:05.146

table notes show the names of the Alloy assertions.[1] These assertions were executed with a scope of *8 but 15 Rule*, i.e., use a maximum of 8 instances for all the signatures but use 15 for the rules. The names and numbers in the table notes are matched with the table entries, i.e., *ID*'s in the table. The items in the table with:

1. IDs 13-16, are process-aware rules used to verify that the sequences of operations as defined in Section 4.1.2 are never violated; and

2. ID 17 is a data-aware rule that verifies that query results conform to access tickets used to execute the query.

These results show that assertions with IDs 15 and 16 that we have also highlighted in the table have the longest running times, almost 7 hours each.

4.3 RCA using Promela/Spin

4.3.1 Overview of Spin/Promela

The model checking tool Spin, uses the Promela language to specify models [45]. Each Promela model may be verified according to assertions, Linear Temporal Logic (LTL) formula, or *never claims*, i.e., violation of correct behaviour, in the model. If an error is found then the verification steps leading to the error (saved as a trail) may be replayed in simulation mode to show the violation. In theory, Spin can be configured to use as much memory and processors as available on a computer or a number of accessible computers.

For the verification, Spin offers 5 different storage/search modes: 1) *exhaustive*, 2) *exhaustive plus minimised automata* (MA), 3) *exhaustive plus collapse compression* (Collapse), 4) *hash-compact* (HC), and 5) *Bitstate*. Some of the modes may be combined, e.g., *MA+Collapse* and *HC+Collapse*. When an exhaustive analysis can be completed we are assured that if Spin reported that no errors were found, that it is indeed so. *HC* and *bitstate* perform approximate searches but give good results where exhaustive searches are not possible. An exhaustive analysis is usually more space intensive than the other modes for the same computer resource allocation.

[1] The names are descriptive enough to identify which operations are involved.

Spin has been developed as a tool to verify process models, and so, does not include constructs for specifying structural constraints beyond those that may be represented with numerical (integer) data types. Additionally, Spin is not suited for the complex computations in data. The power of Spin as a model checking tool lies in the fact that it can be used to exhaustively analyse all interleaving of process statements in a non-deterministic way. From this interleaving, we know that if there is an error within the bounds of memory assigned to the analysis, it will be found.

Even though each piece of numeric data requires a small amount of memory, the exhaustive combination of process variables cause a state space explosion that can quickly reach the assigned memory bounds. This means that abstraction techniques are required by the modeller. In addition, Spin employs memory minimisation techniques to further reduce a state space explosion. However each application of abstraction or memory minimisation may cause loss of details or precision respectively.

Regardless of these limitations and constraints, our aim is to test the memory limits for RCA using Spin, especially for the operations highlighted in Section 4.1.2 and their associated structural details in Figure 4.1.

4.3.2 NJH Promela Specifications

Since we are aware of the limitations of Spin to handle (low-level abstractions in) data, we decided to develop a Promela spec for the NJH system incrementally to tests its limits. We decided to focus on operations where an access ticket is applied for, approved, or declined. The model in Figure 4.1 captures the *DecisionRule*s used to approve an access ticket (see annotation E and H in the figure). Except for the *QualifierPresent* decision rule, all the other decision rules are used when approving a project's access ticket application. We decided to start with the *NoSupsInPIandDC* decision rule.

The *NoSupsInPIandDC* rule declines a project's access ticket if the project's principal investigator and data collector are in a supervisory relationship. These can be determined using the elements at annotations A, B and C in Figure 4.1. In the Promela model, we use:

1. the *init* process to initialise a random configuration of personnel in the supervisor association and for a project's principal investigator and data collector;

Table 4.2: Computer Specifications for Verification

Name	**Type**	**Processors**	**Memory**	**Operating System**
C_1	HP-Z800-XeonE5645-SAS	12x2.4Gh	96Gb	Linux(Fedora)
C_2	HP-Z440-XeonE5-1650v3	6x3.5Gh	32Gb	Linux(Fedora)
C_3	HP-Z440-XeonE5-1650v3	6x3.5Gh	32Gb	Linux(Fedora)

2. a process to approve a project's access ticket application;
3. a process to decline a project's access ticket application;
4. an LTL formula to verify that the *NoSupsInPIandDC* rule is never violated; and
5. a never claim to ensure that a project's access ticket cannot be approved and declined at the same time.

It is important to add the never claim because 1) Spin's analysis examines all interleaving of the processes, 2) we use different variables to indicate approved or declined access ticket application, and 3) we want to ensure that race conditions will not set both variables. The Promela model is shown in Appendix A.1.

4.3.3 Promela Model Verification Results in Spin

After analysing the Promela model of the NJH system using many different configurations for memory, storage/search modes, and number of processors to the limits of those available for the computers in Table 4.2, we were unable to determine its maximum depth, search space, or number of transitions. Therefore, we decided to explore a (different) smaller Promela model to try to understand why we were not able to achieve full exploration of the NJH model. We describe the smaller model in Section 4.3.3.1 and the best results we have achieved for the NJH Promela model in Section 4.3.3.2.

4.3.3.1 Evaluating Promela/Spin on a Small Model

The smaller model, hereafter called t_1, is shown in Listing 4.1. It defines:

1. a message channel, *sChan*, whose size is determined by the value stored in the variable M (see lines 1, 2, and 5 of the listing);

Listing 4.1: t_i, *a Promela Example: non-deterministic add and remove 3 known values from a channel*

```
#ifndef M
#define M  3
#endif

chan send_chan = [(M*M)-1] of {byte }// a message channel of (M*M)-1 slots

/***************************************************************************
 * LTL
 ***************************************************************************/
/* ensures that we have some nondeterminism in the approve and decline of
 projects
 */
ltl ltl1 {
    /* infinitely executing the statement in init with label end implies
    (ensures) we infinitely execute the statement labeled end_again in get() */
    []<>send@end_send -> []<>get@end_get }

init {assert(17>M);}

active proctype send() {
end_send: do
    // send value 50to the channel
    :: send_chan!50

    // send value 198to the channel
    :: send_chan!198

    // send num to the channel
    :: send_chan!M
    od}

active proctype get() {
    byte num;

end_get:
    send_chan?num;
    goto end_get }
```

2. an initialisation process *init* that ensures that *sChan* cannot have more than the 255 slots that *Spin* allows[2];

3. a process, *send()* that loops forever to non-deterministically to put any of 3 values on the channel;

4. a process *get()* that removes a value from the channel; and

5. an LTL formula, *ltl1* that ensures fairness between the two processes.

Executing the model in verification mode works to :

1. verify that *ltl1* is not violated; and

2. enumerate all the possible ways the three values can be placed in and removed from the channel.

4.3.3.1.1 Simpler verification for of t_1 for $M = 3$ and $M = 4$. Some results for verifying t_1 under all the different storage/search modes for $M = 3$ and $M = 4$ are shown in Tables 4.3 and 4.4 respectively. The verification was conducted on computer C_3 (see more details for this computer in Table 4.2) and each verification used a single processor.

While many of the storage modes explored all of the search space, the least memory requirement is for a storage mode using *MA*, and this row is highlighted in grey. Compared to all the executions where a single mode is used, the *MA* mode requires the most time to complete.

4.3.3.1.2 Verification of t_1 for $M = 5$. For t_1, $M = 5$ makes a channel of size 24 slots. Within an 84GB memory allotment to the verification from computer C_1 listed in Table 4.2, we have not been able to determine the limits for $M = 5$. We show verification results for some of the storage modes for $M = 5$ in Table 4.5.

The question mark in the *Percent of Total States Explored* column indicates that the search space was not completely explored so we cannot say what percentage of the states were explored.

Specifically, in the case of:

1. the *Exhaustive* storage mode, the memory bound was reached without exploring all the state space;

[2]We can pass to the model from the command line a different value of M than its value of 2 defined in the model.

Table 4.3: Verification *ltl1* in t_1 for $M = 3$

Storage Mode	sChan Size	Depth Reached	States Stored	States Matched	Transitions	Percent of Total States Explored	Equivalent Memory Usage for States (MB)	Actual Memory Usage for States (MB)	Memory Compression for States (%)	Hash Table (MB)	Bit Stack (MB)	Proc and Chan Stack (MB)	Actual Memory Used (MB)	Time Taken (s.ss)
Exhaustive	8	15069	19683	59040	78723	100.00%	1.502	1.165	77.56%	1024			1078.48	0.02
MA	8	15069	19683	59040	78723	100.00%	1.502	0.678	45.14%	0			53.994	0.21
Collapse	8	15069	19683	59040	78723	100.00%	1.502	1.36	90.55%	1024			1078.675	0.02
MA + Collapse	8	15069	19683	59040	78723	100.00%	1.502	1.159	77.16%	1024			1078.384	0.3
HC4	8	14983	19611	58824	78435	99.63%	1.496	0.97	64.84%	1024			1078.285	0.02
HC4 + Collapse	8	15069	19683	59040	78723	100.00%	1.502	1.36	90.55%	1024			1078.675	0.02
Bitstate	8	15069	19683	59040	78723	100.00%	1.352			16	7.629		77.719	0.02

Search Space Assigned = 1000000, Memory Assigned = 2048MB, Memory Used for DFS Stack = 53.406MB

Table 4.4: Verification of *ltl1* in t_1 for $M = 4$

Storage Mode	sChan Size	Depth Reached	States Stored	States Matched	Transitions	Percent of Total States Explored	Equivalent Memory Usage for States (MB)	Actual Memory Usage for States (MB)	Memory Compression for States (%)	Hash Table (MB)	Bit Stack (MB)	Proc and Chan Stack (MB)	Actual Memory Used (MB)	Time Taken (s.ss)
Exhaustive	15	28274461	43046721	1.29E+08	1.72E+08	100.00%	3612.625	2300.614	63.68%	1024			4925.978	53.5
MA	15	28274461	43046721	1.29E+08	1.72E+08	100.00%	3612.625	130.105	3.60%	0			1732.155	482
Collapse	15	28274461	43046721	1.29E+08	1.72E+08	100.00%	3612.625	2628.274	72.75%	1024			5253.907	86.9
MA + Collapse	15	28274461	43046721	1.29E+08	1.72E+08	100.00%	3612.625	1267.424	35.08%	1024			3893.167	4.14e+04
HC4	15	20405341	38266377	1.16E+08	1.54E+08	89.48%	3211.442	1459.954	45.46%	1024			4086.036	37.1
HC4 + Collapse	15	28274461	43046721	1.29E+08	1.72E+08	100.00%	3612.625	2628.274	72.75%	1024			5253.907	62
Bitstate	15	27739169	39644347	1.22E+08	1.62E+08	94.05%	3024.624			16	762.939	1058.21	2905.356	61.8

Search Space Assigned = 30000000, Memory Assigned = 9192 MB, Memory Used for DFS Stack = 1602.173 MB

2. *MA* storage mode, the maximum search depth assigned to the verification was reached at $time = 1.15e+04$ seconds (3 hours) when only $5.68e+08$ states and $1.19e+09$ transitions were explored; the values shown in Table 4.5 are for when we interrupted the search after 22 hours; and

3. *Bitstate* storage mode, the search completed without reaching the memory bounds or search depth of either of the *Exhaustive* or the *MA* storage modes.

4.3.3.2 NJH Model

We instantiated the model with 8 projects, 8 personnel having supervisors[3] defined from which we randomly chose both the principal investigator and the data collector. We executed the verification with up to 24GB of memory. Table 4.6 shows results for some of the storage modes. These results show us that the verification was able to:

1. reach a search depth of 7,876,539 as shown for the exhaustive mode;

2. store 8.49e+10 states as shown for the MA mode; and

3. explore 9.34e+11 transitions as shown for the MA mode.

However, we neither know if the search depth reached nor the transitions explored are the complete state space. Of the four rows in the table, the row highlighted in grey, gave the best result; the best result is determined as the verification that explored the most states. For this row, the verification:

1. was assigned all the processors on computer C_2 described in Table 4.2, 24GB of memory, and a MA storage/search mode;

2. reached only 13% of the depth of the exhaustive mode;

3. explored more than 2000 times the transitions of the exhaustive mode;

4. was (manually) terminated after 10 days, 23 hours, 36 minutes, and 40 seconds with the knowledge that the full state space for the model has not been explored; and

[3] The *Supervisor* association is a tree.

Table 4.5: Verification of *ltl1* in t_1 for $M = 5$

Storage Mode	Depth Reached	States Stored	States Matched	Transitions	Percent of Total States Explored	Equivalent Memory Usage for States (MB)	Actual Memory Usage for States (MB)	Memory Compression for States (%)	Hash Table (MB)	Bit Stack (MB)	Proc and Chan Stack (MB)	Actual Memory Used (MB)	Time Taken (s.ss)
Exhaustive	868113565	4.9155098e+08	5.3626596e+08	1.0278169e+09	?	45002.836	30562.438	67.91%	2048			86015.930	906
MA	999999999	4.8229954e+09	1.7561717e+10	2.2384713e+10	?	441558.414	4747.638	1.08%				58152.053	79813.23
Bitstate	128680347	82009798	4.7289205e+08	5.5490184e+08	?	6882.536			16	7629.395	5399.689	66453.891	353

sChan Size = 31, Search Space Assigned = 1000000000, Memory Assigned = 2048MB, Memory Used for DFS Stack = 53406MB

Table 4.6: Verification Details for Spin Model without Analysing Process-Aware Rule or Never-Claim

CPUs	Depth Reached	States Stored	Transitions	Memory Used (MB)	Time (s.ss)	Storage Mode
1	7876539	3.4e+07	4.61e+08	6637.351	656	Exhaustive
6	1000039	8.49e+10	9.34e+11	-	9.49e+05 (approx. 11 days)	MA
6	610204	7.770891e+10	8.4079559e+11	17715.799	9.42e+04 (approx. 26 hrs)	Bitstate
12	627994	3.891082e+10	4.2225942e+11	9979.258	6.81e+04 (approx. 19 hrs)	Bitstate

5. did not use all of the 24GB assigned to it: since we terminated the execution we were unable to determine the amount of memory used.

In comparison to the test program, this model has a larger number of states, and it is possible that while the MA storage mode could explore them, it takes too much time.

4.4 Discussion of Results and Summary

Our first evaluation question asked:

What kinds of rules are best suited for each tool?

and our second evaluation question asked:

What are the space and time measures when using the tools and how can we use these measures to motivate HMCA?

The results in tables 4.3 through 4.6 confirm that Promela/Spin is not designed for data intensive processing. However, even the large abstractions we applied to model and check the *NoSupsInPIandDC* rule contained too many states because our verification was not able to explore all its states. Therefore, while Promela/Spin is suited for checking properties for process interleaving, even using large abstractions of data cause a large explosion of the state space. We note that even with a small configuration of projects and researchers all the possible execution states could not be explored.

Therefore, for the first question from Section 4.1.1, we conclude that Spin, like UPPAAL, is most suited for analysing process-aware rules using very large abstractions. This is supported from the analysis results. The analysis results also answer the second question such that we know that Spin is not suitable for RCA when we need to use the details in a data model to show rule conformance.

On the other hand, the results in Table 4.1 show that the Alloy Analyzer was able to handle the analysis of complex data relationships in data-aware rules. However, it showed much longer execution times when we combined both process-aware rules and data-aware rules in a single model. Since the Alloy Analyzer is able to return results for data-aware rules, we know that applying slicing, as discussed by Sun et. al, will yield better results, i.e., shorter execution times.

Therefore, for the first question from Section 4.1.1, we conclude that the Alloy Analyzer with slicing is best suited for data-aware rules. This shows that the Alloy Analyzer may be useful in HMCA.

5. HMCA AND NJH

HMCA follows three phases in conformance analysis: model construction, model analysis, and providing feedback. The aim in the construction phase is to have formally analysable system models and a representation of the rules that can be checked on the system models. In the analysis phase we show the application of a divide and conquer strategy to construct the transition system. For the final phase we provide feedback to the user especially where conformance rules have not been satisfied. While our proposal for analysing rule conformance is generalisable, it is most easily explained when applied to an example. For this, we will use the NJH research system whose conformance rules come from HIPAA regulations. We will show how to 1) construct models, 2) analyse them for conformance to HIPAA de-identification (a *DeIDed access ticket* in the NJH System); we will hereafter refer to this as the HIPAA de-identified rule, and 3) provide feedback in the case of non-conformance. In addition, for the purposes of explaining conformance to the de-identified rule, we simplified its definition to only cover de-identification of dates, i.e., dates contain only a year value.

The rest of this chapter explains HMCA using the NJH system. We return to a generalisation of HMCA in Chapter 6.

5.1 Phase 1: Model Construction

5.1.1 Construct Activity Model and Class Model

In previous work [23, 39] we constructed a path representation of the system as a UML activity diagram, and a system structure representation as a UML class diagram.

The input for the activity diagram was from the *Map of Integrated Bioinformation and Specimen Centre Research Support* flowchart [44] that shows the process used by researchers to apply for licenses and access tickets and to access data. Flowchart constructs, e.g., sequential flows, choice, and loops, have equivalent representations in UML activity diagrams, so our aim was to transform the flowchart to a more formal activity diagram. However, the flowchart had non-standard flowchart representations, e.g., more than a single flow out of, and into action nodes, so we applied normalisations, e.g., inserting decision and merge points so we could distinguish the flows in and out of action nodes. These normalisations ensured that the flowchart was well-formed. In preparation for transforming the flowchart to an activity diagram, we also distinguished whether the paths out of decision nodes should be concurrent flows or not so we could know where to apply an activity diagram *fork and join* transformation versus an *if-then-else* transformation. In order to have all possible paths represented we ensured that all possible values for a decision were included. The transformation rules we applied to the flowchart to produce the activity diagram were mainly from our experience with flowcharts and our observation of how to represent them, together with the added formalisms of activity diagrams.

The design-level class diagram was constructed through our understanding of the structural elements and relationships required to support the activities in the activity diagram in Sections 5.1.5 and 5.1.6.

5.1.2 Construct Entity Views

Activity diagrams have semantics that make them amenable to state machines and transition systems [9, 10, 31, 35, 37, 42, 73], so we applied transformations to the system activity diagram to create a system-wide state machine in [23, 39]. However, the state machine produced is still quite complex and does not allow us to isolate the parts of it that relate to showing conformance to specific rules. In order to handle complexity in the NJH system, we identify and model different aspects of the NJH system as separate state machine entity views, hereafter referred to as *entity views*.

The entities may be understood as objects in the system that either perform operations that change their own states, or are states of interest to rule conformance, e.g., researchers and patient health information. The focus of constructing entity views is to bring understanding to the individual states of entities and how the composition of these individual entity states influence the complete system state.

5.1.2.1 Individual Entity Views

We construct an entity view by extracting operations performed by the entity on other entities in the system, e.g., researcher queries patient health information. Each constructed entity view is a representation of abstract operations and state pertaining to that entity.

For the *DeIDed* access ticket, the entities of interest are the researcher and the patient's health information that will be accessed by the researcher. We show in Figure 5.1 the researcher's entity view in the NJH system.

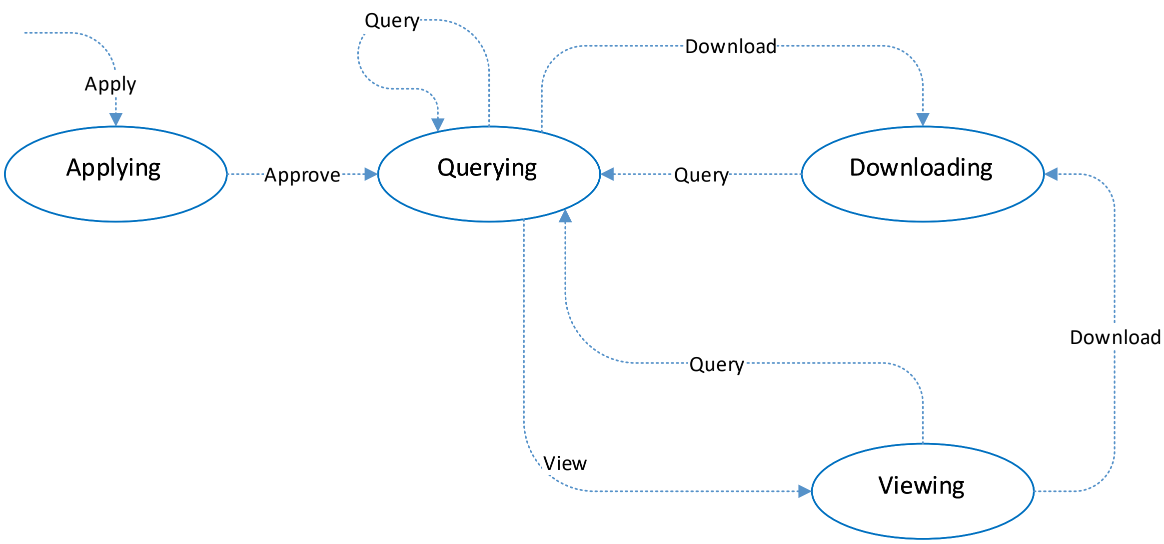


Figure 5.1: Researcher/Project Entity View

The nodes in Figure 5.1 use *atomic propositions* to show what the researcher is doing e.g., *Applying* for an access ticket, and the edges show the operations, e.g., from the *Applying* state an *Approve* operation takes the researcher to the *Querying* state. While the researcher does not carry out the *Approve* operation it is important to include it in the researcher's entity view as it affects the reachability of other states. This entity view contains non-determinism as we have not shown the additional conditions that differentiate the enabling of any of the edges exiting the states. This entity view also reflects the operations and states for a project. One of the aspects of the *DeIDed* rule is whether the *Query* operation result contains any patient's identifying information.

The entity view shown in Figure 5.2 is a view of the patient's health information for de-identified access: we specifically use atomic propositions to model that the health information can be *Identified* or *De-identified* when either of the *View* or *Download* operations are performed; the use of the {} on the self loop from the *Identified* state means that any operation is allowed, e.g. a researcher could be viewing or downloading *Identified* patient health information. This view also contains non-determinism as we have not shown the additional conditions from the system state that differentiate the enabling of either of the edges exiting the *Identified* state.

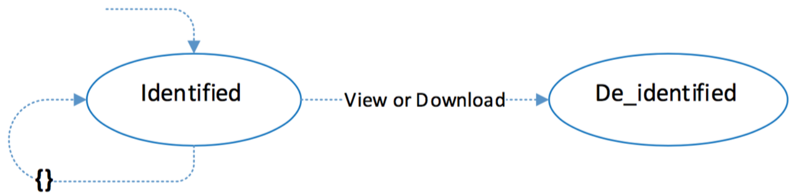


Figure 5.2: Patient Health Information Entity View for De-identified Access

5.1.2.2 *Composing Entity Views*

When a researcher obtains a *DeIDed* access ticket, we are then interested in both the researcher and the patient health information entity views. In order to understand the system in terms of what the researcher is doing and the state of the health information, we compose the views. We show the composition of the entity views in Figure 5.3. The process of composing the views relies on the *handshaking* [14, see section 2.2.3] of operations, such that when identical operations occur on the label of an edge, their next states are combined into one state. The composition of the entity views produces a *rule specific entity view*, that now labels a state with a 2-tuple atomic proposition; the first element identifies the state of the researcher, and the second element is the state of the patient health information.

In any real system implementing querying operations, the results are immediately accessible, i.e., querying and viewing will appear to a researcher to be an atomic operation, so showing that the state changes in the patient health information occurs after the *View* or *Download* operations is an acceptable representation.

Again, this rule specific entity view contains non-determinism, e.g., the *Download* operation is a label on edges from the <*Querying, Identified*> state to both the <*Downloading, Identified*> and the <*Downloading, De-identified*> states. This non-determinism identifies that these possibilities exist in the system at this level of abstraction.

5.1.3 Modelling Conformance Rules

The rule specific entity view in Figure 5.3 contains some states that show non-conformance to the de-identified rule, i.e., the states identified by <*Downloading, Identified*> and <*Viewing, Identified*> are illegal and we must be able to probe a transition system for their occurrence. A transition system produced by a model checker may be viewed as sequences of states, or traces.

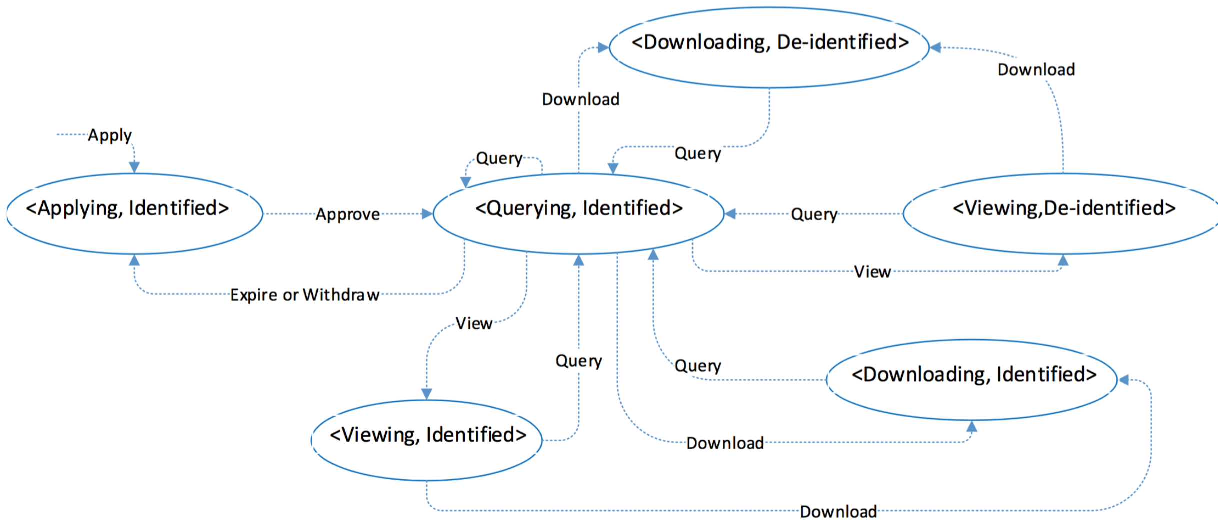


Figure 5.3: De-identified Rule Specific Entity View

An example of a partial trace for Figure 5.3 is

<Applying, Identified><Querying, Identified><Viewing, Identified>...

and the model checker must identify that this transition system shows non-conformance because an illegal state is present in the trace when a *DeIDed* access ticket is used.

In order to find this non-conformance, we specify a property using a graph formalism, called a non-deterministic finite automata, NFA [14, see Chapter 4], that checks the transition system for illegal states. Figure 5.4 shows the formalism for the de-identified rule specified using the atomic propositions in Figure 5.3. It shows that the system is in state *Conforms* when *View* or *Download* is executed with a *DeIDed* access ticket. An NFA processes each item (e.g., *<Applying, Identified>*) in the trace and if the final state, shown by the node with two elipses, can be reached then the system does not satisfy the property.

We add that the system may still be adjudged to be conforming to the HIPAA de-identified rule even if the *Query* operation gives *Identified* results since the non-conformance happens when *Identified* results are viewed or downloaded. The use of the *Not <Viewing, Identified>* or *Not <Downloading, Identified>* label on the self loop into the *Conforms* state ensures that neither *<Viewing, Identified>* nor *<ViewDownloading, Identified>* are true for the system to be adjudged to be in conformance to the rule.

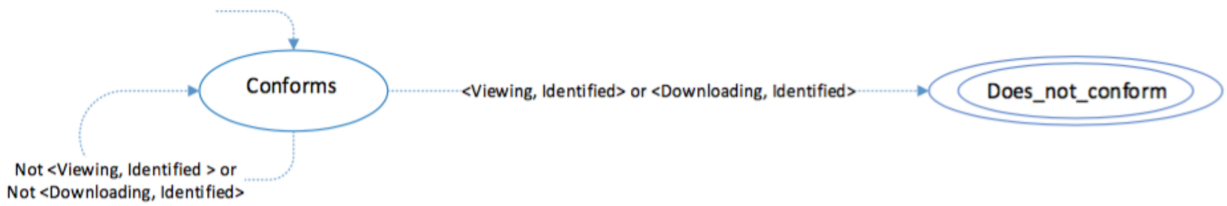


Figure 5.4: Graph Formalism for the HIPAA De-identified Rule

5.1.4 Map Rule Specific Entity Views to System Models

5.1.4.1 Map Operations to Activities in the Activity Model

The operations in a rule specific entity view (e.g. Figure 5.3) are abstractions of actual activities in the activity diagram discussed in Section 5.1.1, and we may map these abstractions to their refinement in the activity diagram. It is important to have such a mapping in order to identify actual system processes that will be examined when analysing for rule conformance. In Figure 5.5, we show a portion of the activity diagram of the NJH system for obtaining a *DeIDed* access ticket and the subsequent querying using the same access ticket.

With reference to the labeled activities in the activity diagram, we know that:

1. A1 (*Decide if research can use de-id'd data*) through A5 (*Apply for DeIDed access ticket*) maps to the *Apply* operation in Figure 5.3;

2. A11 (*Grant DeIDed access ticket*) maps to the *Approve* operation;

3. A13 and A18 together with A25 each maps to the *Query* operation; and

4. A26 maps to the *View* (or *Download*) operation.

5.1.4.2 Map Atomic Propositions to Concrete Class Model Elements

The atomic propositions used to identify states in a rule specific entity view represent abstractions of actual system states and we can map these abstractions to the concrete representation of the states in the class diagram. In order to distinguish patient health information as *Identified* or *De-identified* we will need to provide tests. We will return to how we define these tests in Sections 5.2.3 and 5.2.4 in the analysis phase.

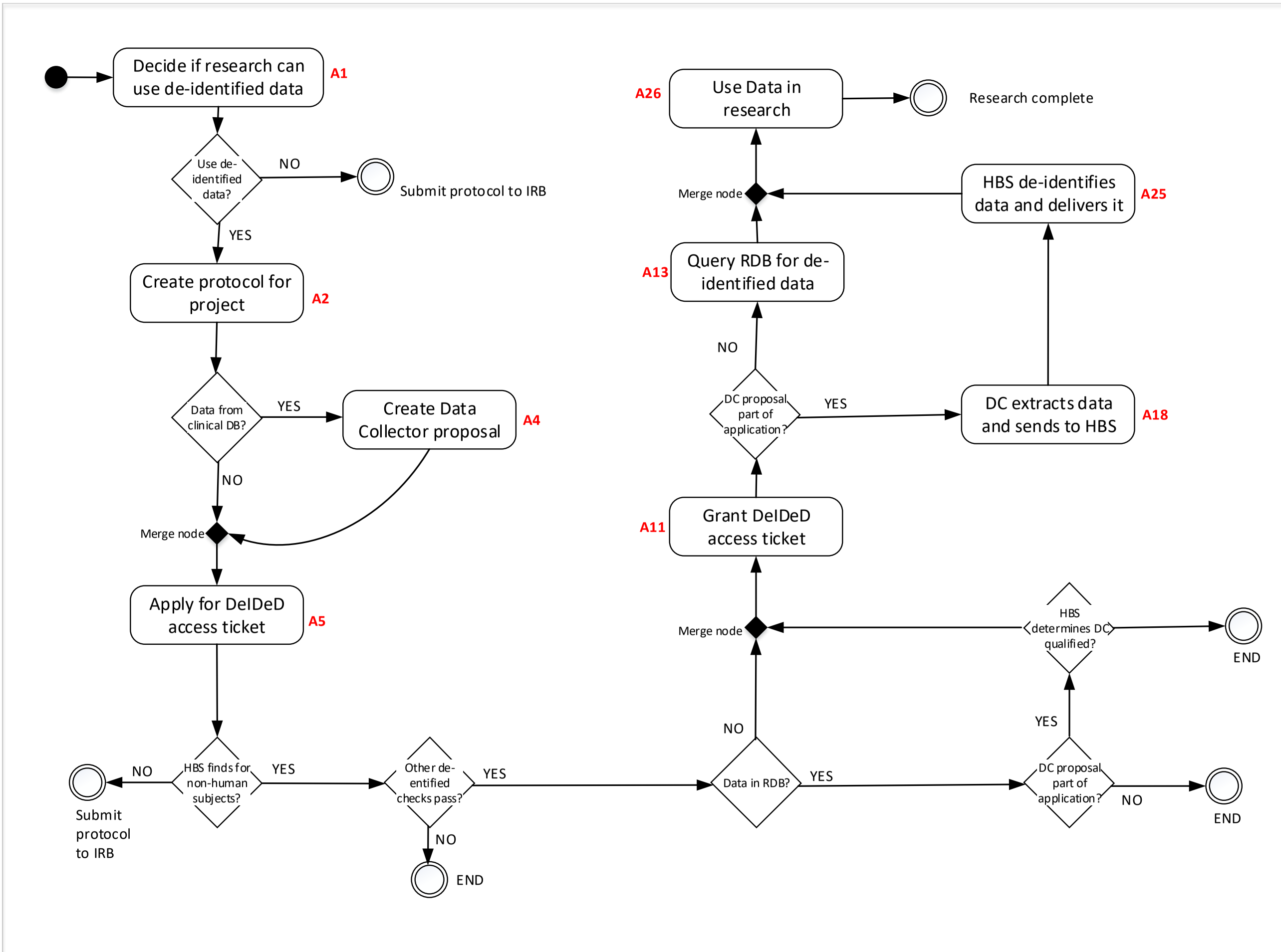


Figure 5.5: AD Segment for De-identified Health Information Access

5.1.5 Annotate Activity Diagram with Details from the Class Model

The details of the system state that allows a researcher to execute a query and view (or download) its result are of interest. We show in Figure 5.6 the class diagram segment of the system state that these operations access when using a *DeIDed* access ticket.[1]

The *runQuery* method in the *Query* class in Figure 5.6 is mapped to the activities in the activity diagram corresponding to querying (e.g., A13 in Figure 5.5) referred to as the *activity diagram query segment* below.

We add annotations to the activity diagram query segment with pre-, and postconditions specified in the class diagram. The activity diagram query segment is annotated with the *runQuery* specification:

- *Input*: Researcher, Query, and associated Project (which allows us to obtain the project's access ticket from *ProjAT*);
- *Precondition*: Researcher requesting access is authorised (as indicated by the access ticket);
- *PostCondition*: output is *Identified* or *De-identified*; and
- *Output*: *QryReturns* Association.

5.1.6 Create Concrete Rule Specific State Machine from Annotated Activity Diagram and Entity Views

A de-identified rule-specific entity view state machine is shown in the composed entity views of the researcher and patient health information in Figure 5.3. We must map details of the annotated activity and the class diagram to this state machine. The mapping tell us which of the system activities and states are of interest to analysing conformance to the HIPAA de-identified rule. For simplicity we do not show the mapped region of this diagram (but these are discussed in more detail in Section 7.1).

This mapped rule-specific entity view is a more concrete representation of the (abstract) rule-specific entity view such that for each:

- operation in the entity views there is traceability to:
  - its corresponding method in the class model,

[1] The class diagram segment represents changes that are improvements to the class diagram initially developed in [23].

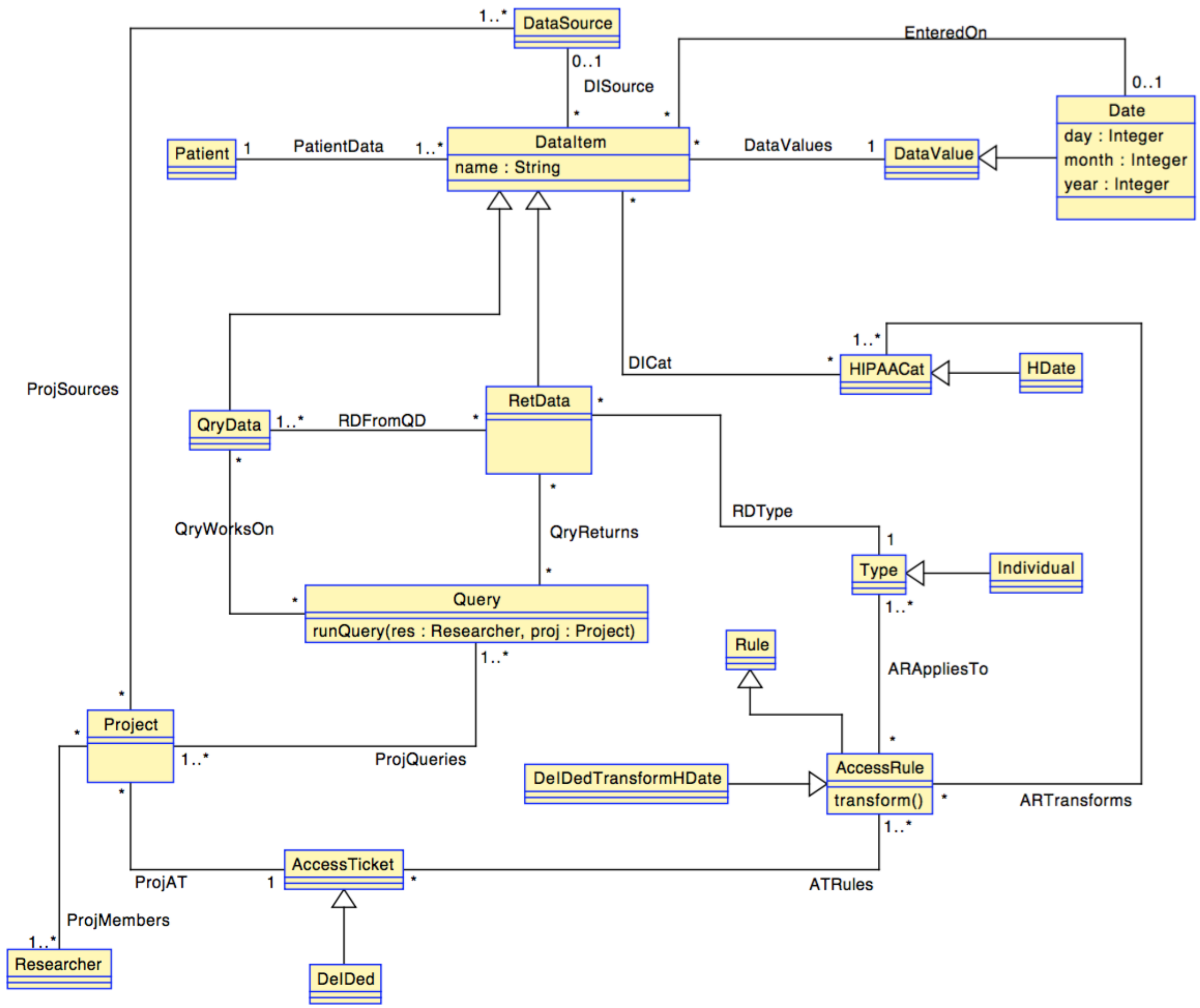


Figure 5.6: System State of Interest to De-identified *Query*, *View*, and *Download* Actions

  - the part of the class model that corresponds to its signature, i.e., input, and
  - its pre- and postconditions based on the class model.

- state, its abstract atomic proposition can be traced to its concrete state as represented in the class model.

We use the mapped rule specific entity view as the program graph that we will use to extract evidence of conformance to our simplified de-identified rule. In the analysis phase we will discuss the specific class diagram methods linked to the operations in the mapped rule specific entity view.

5.2 Phase 2: Model Analysis

The analysis result needed to show conformance to the HIPAA de-identified rule is a transition system that does not contain either the <*Viewing, Identified*> or the <*Downloading, Identified*> states for the *DeIDed* access ticket. In order to achieve tractable analysis results when producing the transition system from the mapped de-identified rule-specific entity view created in Section 5.1.6, we create the transition system semi-automatically by individually analysing the state produced by each operation. The slicing technique extracts (copies) the class model elements required for each method into a smaller class model. The class model slice is transformed into an equivalent Alloy model and we use the Alloy Analyzer to probe the model for its resulting state when the associated operation is performed.

The Alloy language allows us to define predicates and functions whose return values may later be used as other constraints in the model or used to generate instances of the model. In addition, we may use the predicates and functions in assertions to verify that all instances of a model possess certain properties or are in a particular state. The order of operations defined in the de-identified rule-specific entity view state machine tells us how to insert each mapped method's state into the transition system. The final step will be to verify that the transition system does not violate our simplified statement of the HIPAA de-identified rule.

5.2.1 Identifying the Slice of Interest

From our discussion in Section 5.1.2.2, since the *Query* and *View* operations together may be viewed as a single atomic operation, we may assign the job of de-identifying the data to any of their mapped methods. In our model, we assign the job of de-identifying the data to the *runQuery* method and our analysis will be to determine whether its result could cause the *View* (or *Download*) method to produce in an illegal state.[2]

In an Alloy model, classes are represented using signatures. For example, using the *sig* keyword we define the *Individual* and *Type* classes from Figure 5.6 and make the *Individual* class inherit from *Type* class using the code in Listing 5.1.

Listing 5.1: Alloy Signatures

```
abstract sig Type {}
sig Individual extends Type {}
```

Associations between classes are represented using relationships between signatures. For example, Listing 5.2 shows the Alloy representation for the *DataValues* association from Figure 5.6 where each *DataItem* is associated with exactly one *DataValue*.

Listing 5.2: Alloy Relationships

```
DataValues: DataItem -> one DataValue
```

5.2.2 Adding Operation Specification

The Alloy model that contains the equivalent representation for classes and associations must be extended to add specifications for pre- and postconditions of the *runQuery* operation. The precondition includes: 1) the researcher is authorised to execute the query and 2) the results before the query executes are not (yet) known. The postcondition defines the query result. In general, operation specifications declare that when an operation's precondition is satisfied, then the operation's result expressed in the postcondition is also satisfied. We enforce this operation's specification by constraining the model to only change in response to the *runQuery* method executing. For *runQuery*'s full Alloy operation specifications see Appendix B.1.

[2] As a reminder, even if the result the *runQuery* method makes available to the *View* method is *Identified*, we cannot adjudge that the system is in an illegal state until the *View* method terminates.

5.2.3 Probing the Illegal State.

Our specific interest is to determine the state of the query result after any execution of the *runQuery* method. Assertions are used to examine whether all possible configurations of signatures and relationships in our system *always* adhere to our expectation of the system. The *AlwaysDeIDedConformance*[3] assertion in Listing 5.3 models our main expectation of the results of operations.

Listing 5.3: Probing *runQuery* Model for the *Identified* State

```
assert AlwaysDeIDedConformance{
   all njh: NJH, q: njh.queries |
      all qi: q.(njh.QryWorksOn), ri: q.(njh.QryReturns) |
         ConformanceDeIDed[njh, q, qi, ri] }
```

njh is an instance of the system. *AlwaysDeIDedConformance* is an assertion that looks at every possible instantiation of the Alloy model and checks whether all its queries using a *DeIDed* access ticket return de-identified data. We use the predicate *ConformanceDeIDed* to check that each piece of return data in each query's result is de-identified when a de-identified access ticket is used to run the query. We test this assertion using the statement in Listing 5.4 that uses a scope of 3, i.e., generates a maximum of three (3) instances for each signature, and executes this check; further, we expect the system not to find counterexamples, i.e., expect 0.

Listing 5.4: Executing *AlwaysDeIDedConformance*

```
   check AlwaysDeIDedConformance for 3expect 0
```

If no counterexamples are found we know that when the *DeIDed* access ticket is used, each piece of data in the query's result is always de-identified within the scope defined. While we may use larger scopes, the Alloy Analyzer justifies testing models with small scopes because a high proportion of bugs may be uncovered when testing a program for all test inputs using small scopes (see Section 3.1). If *AlwaysDeIDedConformance* returns a counterexample, we know that some query using the *DeIDed* access ticket terminated in an illegal state.

[3]We use a concatenation of words in the names of the assertions, predicates and functions as an easy way to identify their purpose.

5.2.4 Determining Operation States.

In our model, we not only want to determine if an illegal state could be reached, but also that the legal state is possible. While *AlwaysDeIDedConformance* returning counterexamples tells us about the presence of illegal states, if it does not return counterexamples we still require a further sanity check because it may be that the model produces no instances and therefore no counterexamples could be returned. In terms of Figure 5.3 we must also determine if the *<Viewing, De-identified>* state is reachable. The predicate *CanGetConformanceDeIDed* in Listing 5.5 produces an instance of the system where a query's result is *de-identified*. *BasicDeIdentifiedDateConditions* is a predicate that sets up conditions such that a system instance *njh* contains a query *qry* that extracts some data *qi* and has associated return data *ri*. *not IdentifiedDate[ri.(njh.DataValues)]* ensures that *ri* is de-identified according to our simplified definition of de-identified data (i.e. dates are returned as years).

Listing 5.5: Testing *runQuery* Model for the *De-identified* State

```
pred CanGetConformanceDeIDed
     [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
  BasicDeIdentifiedDateConditions[njh, qry, qi, ri]
       and not IdentifiedDate[ri.(njh.DataValues)] }
run CanGetConformanceDeIDed for 3but 1NJH expect 1
```

We expect that *CanGetConformanceDeIDed* will produce an instance. We use as evidence that the *<Viewing, De-identified>* and the *<Viewing, Identified>* states are reachable when *CanGetConformanceDeIDed* gives an instance and *AlwaysDeIDedConformance* gives counterexamples respectively. We note however, that we do not have enough evidence to show that the *<Viewing, Identified>* state is reachable when *CanGetConformanceDeIDed* finds no instance, or that the *<Viewing, De-Identified>* state is reachable when *AlwaysDeIDedConformance* find no counterexamples.

We use the evidence of the reachability of the states when constructing our transition system. While a program graph, such as the one represented in the mapped rule-specific entity view state machine described in Section 5.1.6, represents the possible states and operations in the system, the transition system is a concrete representation of the actual reachable states (or operations) in the execution of the program graph. For example, if *CanGetConformanceDeIDed* returns instances and *AlwaysDeIDedConformance* does not return counterexamples, we expect the that the analysis of the mapped rule specific state machine to produce

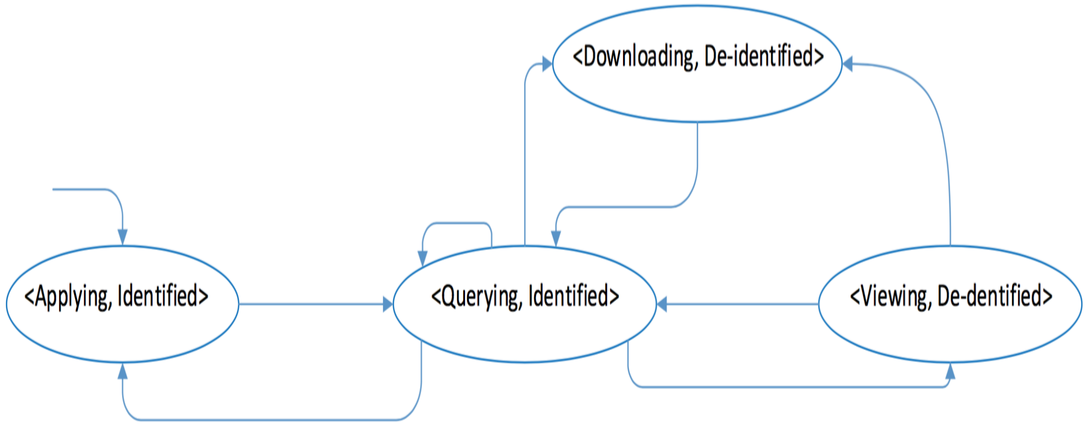


Figure 5.7: Transition System Indicating Conformance to the De-identified Rule

the states in the transition system shown in Figure 5.7. However, if *CanGetConformanceDeIDed* returns instances and *AlwaysDeIDedConformance* returns counterexamples, then we know that the transition system shown in Figure 5.8 containing the illegal states will be constructed from the analysis results. This is because a counterexample from the *AlwaysDeIDedConformance* means that there is non-conformance.

5.3 Phase 3: Results and Feedback

The presence of counterexamples for the *AlwaysDeIDedConformance* assertion represents an illegal state in the system. When an illegal state is encountered, we may use other assertions and predicates to further probe the specifications to find the conditions under which unexpected results were returned. If identified data is returned from a query using a *DeIDed* access ticket, we must be able to generate a detailed system instance that pinpoints the specific project, query, data the query worked on, and the corresponding data returned by the query that produced the illegal state, i.e., object instances of the classes in Figure 5.6.

In our model, all the data that require a de-identifying date transformation are marked using the *DICat* association in Figure 5.6. Therefore, our first check is to ensure that our *Query* operation specification correctly de-identifies marked data. We use the assertion in Listing 5.6 to make this check. If we find counterexamples then we know that our operation specifications are incorrect. *ConformanceDeIDedHDateSet* is a predicate that returns *true* if all the data extracted by a query that is marked as requiring de-identification has been de-identified in the query's result.

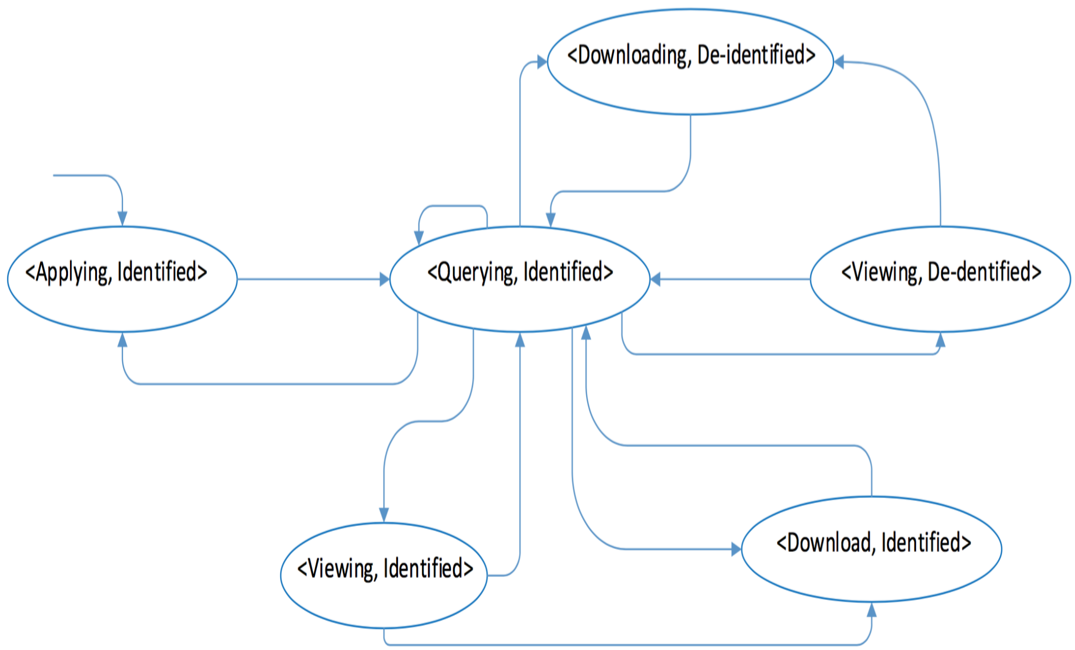


Figure 5.8: Transition System Indicating Non-Conformance to the De-identified Rule

Listing 5.6: Probing for Conformance when Data is Properly Categorised

```
assert AlwaysDeIDedConformanceWhenHDateSet {
   all njh: NJH, q: njh.queries |
      all qi: q.(njh.QryWorksOn), ri: q.(njh.QryReturns) |
          ConformanceDeIDedHDateSet[njh, q, qi, ri] }
check AlwaysDeIDedConformanceWhenHDateSet for 3expect 0
```

However, if this assertion finds no counterexamples, then our probing must continue as the reason for the non-conformance is elsewhere in the model. For the de-identifying transformation to work properly, our model relies on human intervention to link the following:

1. date data items with their appropriate HIPAA category using the *DICat* association in Figure 5.6,

2. transformation rules with the HIPAA categories associated with data they need to transform, using the *ATTransforms* association in Figure 5.6,

3. access rules to the return data types that they should transform using the *ARAppliesTo* association in Figure 5.6, and

4. the access tickets to the appropriate transformation rules, using the *ATRules* association in Figure 5.6.

Our next logical step is to use other assertions to probe the model to verify that these links have been properly created. We have created an example to demonstrate what occurs if data items have not been properly marked for a de-identifying transformation. The predicate in Listing 5.7 may be used to generate an instance where a query *qry* terminates in an illegal state because some data item that the query extracted from the data sources *qi*, and transformed *ri*, were not properly marked as requiring a de-identifying transformation by creating a *DICat* association. *NonConformanceDeIDedFullDateHDateUnSet* is the predicate used to find where this occurs.

Listing 5.7: Probing for a Non-Conformance Instance when a Data Item is Improperly Categorised

```
pred DeIDedNonConformanceFullDateWhenHDateUnSet
     [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
  NonConformanceDeIDedFullDateHDateUnSet [njh, qry, qi, ri]}
```

While the instance produced by the Alloy Analyzer may be viewed graphically, it may not be ideal for giving feedback to the non-technical user. The Alloy Analyser allows the instance to be exported to the Extensible Markup Language (XML) format that we may parse for the query, data items and their associations that resulted in the illegal state. A graphical example of the feedback relevant to this problem is shown in Figure 5.9, where the elements determined to be involved are labeled with the variables used to run the predicate. For example, when giving the feedback, the main variables of interest from the call to the *NonConformanceDeIDedFullDateHDateUnSet* predicate are *qry*, *qi*, and *ri*. These variables are used as additional labels for *Query2*, *QryData1* and *RetData* respectively in Figure 5.9; *QryData1* is the data item extracted by *Query2* and *RetData* is the result that is in an illegal state because *QryData1* was not properly linked to *HDate* to indicate that it should be de-identified.

The counterexample gives us an indication of what needs to be corrected in the model. For this example, we may add corrections to the *runQuery* post condition to ensure that it recognises data values that are dates and de-identifies them, or add a constraint that all *DataValue* classes that are dates are linked to the *HDate* HIPAA category. We applied the correction as a constraint to associate all date *DataValue* classes with the *HDate* HIPAA category in the model. This constraint added in the Alloy model must be propagated to the corresponding class diagram. Through the mapping to the mapped rule-specific entity view described in Section 5.1.6, we know what states in that view are affected. We must also reflect the new constraint in

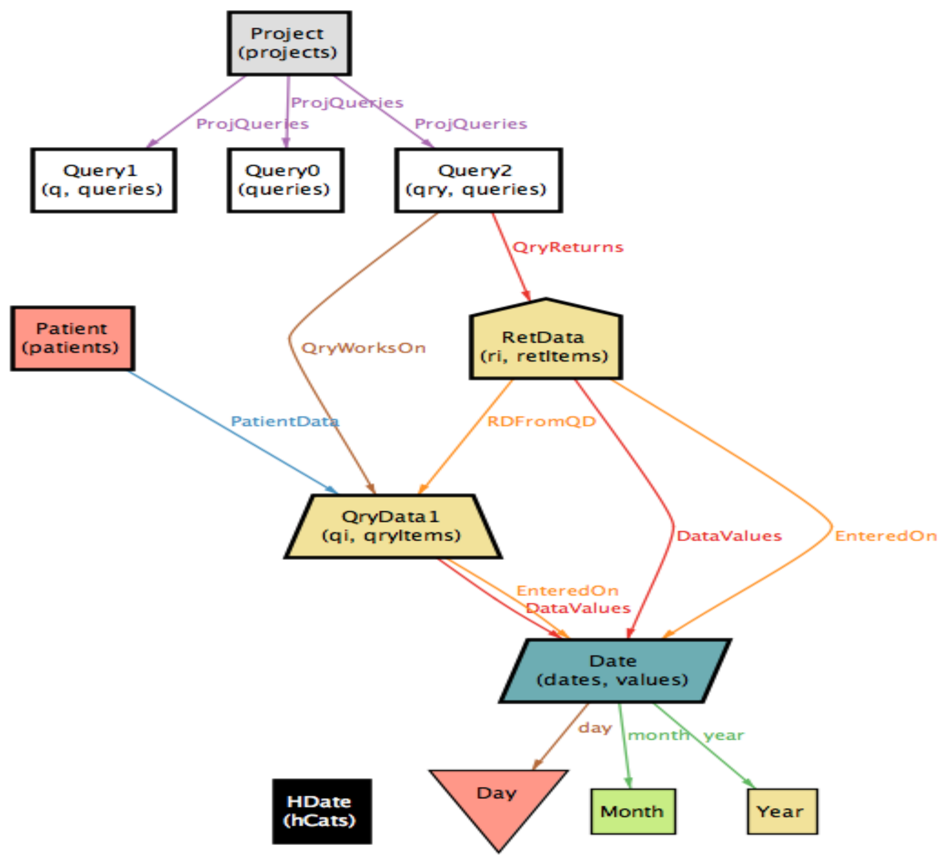


Figure 5.9: Non-Conformance: *Query2* returns *Identified* Data

the part of the annotated activity diagram in Section 5.1.5 dealing with *DataValue* objects, and from there changes may need to be made to the portions of the work flow initially linked to this part of the activity diagram. If, however, the correction was applied in the postcondition, since the *Query* operation in the rule specific entity view has annotations from the class diagram, the post condition change to the class diagram's *runQuery* method can be propagated through this mapping.

Appendices B.1 and B.2 give details on the important predicates and assertions for the *runQuery* operation.

6.1 HMCA Generalisation

The process used and the models created for verifying system conformance to our simplified de-identification HIPAA rule may be generalised for verifying other rules. We use Figure 6.1 to represent the main activities of HMCA, and with each activity show its inputs and outputs as models/artefacts[1]. We will refer to this view of HMCA as an external view because we highlight the major activities and the models/artefacts that are used across these activities. In this external view of HMCA we use the numbers 1 through 6 to highlight the steps.

At step 1 we take as input conformance *Rule Requirements*, e.g. HIPAA regulations, that we use to drive the construction of the models needed for the analysis phase. We highlight that we need more formal:

1. data-oriented system models and for this we construct a UML *Class Diagram* that give us the additional details needed to identify specific places in the system where the rules are not satisfied;

2. process-oriented system models, and for each rule we construct an *Annotated Rule-Specific Entity View* (ARSEV) as a state machine and will use it to test conformance for each rule; and

3. rule representation, and for this we construct a *Rule NFA* for each rule using the atomic propositions labelling the system states in the ARSEV to define conformance rules by defining illegal states e.g., the simplified HIPAA de-identified rule in Figure 5.4.

Each constructed model requires verification from the respective domain experts to verify that they are correct. We also show the links between models/artefacts that are important to maintain by using traceability links. In the construction phase, we link each conformance *Rule Requirement* to its representation as a *Rule NFA*, each *Rule NFA* to its corresponding ARSEV, and since the ARSEV's annotations also come from the *Class Diagram* (see Section 5.1.6) we also provide traceability links between them.

In the analysis phase we produce a *Transition System* from each ARSEV and use it to check whether the corresponding conformance rule has been satisfied. In order to determine system state and to avoid in-

[1] The diagrams in this section are best view in colour to differentiate the purpose of each coloured line.

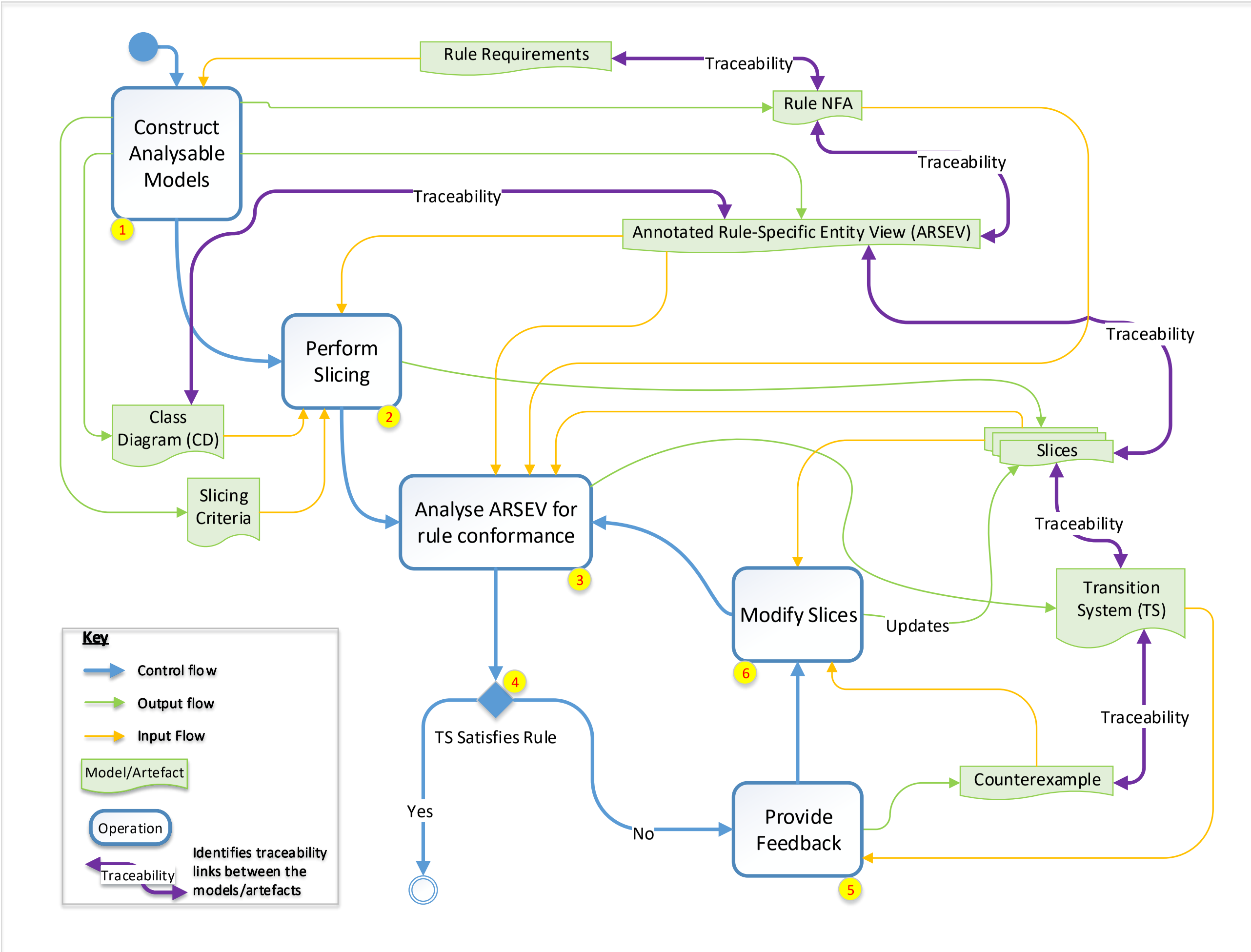


Figure 6.1: Generalised HMCA

tractable analysis results when using a model checker, we construct the transition system semi-automatically by slicing to produce smaller models. Though we do not separate the slicing of the models from the analysis phase, we include it as a separate step in our external view because we later allow the slices to be modified and re-analysed, and slicing the models takes place only once[2]. *Slicing* requires as input the *Class Diagram*, *Slicing Criteria* i.e. each method in the class diagram or each operation in the ARSEV that may correspond to a sequence of several methods in the class diagram, and the ARSEV from the construction phase. The analysis performed at step 3 produces the transition system.

If the conformance rule is satisfied in the transition system, the *Yes* branch at step 4 is taken and our process ends. However, the *No* branch at step 4 becomes important when the conformance rule being checked is not satisfied. This *No* branch at step 4 allows us to provide counterexamples at step 5, to modify the Alloy slices at step 6, and to re-analyse for rule conformance at step 3.

This external view of HMCA hides many internal sub-activities that produce intermediate models/artefacts so we provide decompositions of each phase as HMCA internal views in Subsections 6.2 to 6.4 below.

6.2 Construct

Using the same conventions in the key from Figure 6.1, we show a more detailed view of the construction phase in Figure 6.2 by giving a step-wise decomposition as internal sub-activities and include additional models/artefacts used and produced. We use the numbers 1 through 8 to highlight the steps. Some of the highlights are that:

- we show the specific sub-activities that use and produce models/artefacts, e.g. *Rule Requirements* is used at step 4 and the *Rule NFA* is constructed at step 5;
- we include additional internal models/artefacts, produced at steps 1, 3, 4, and 6;
- we include internal traceability links for the models/artefacts, e.g., the ARSEV now has traceability with the *Annotated Activity Diagram* (AAD) because we link its operations with the actions in the AAD;

[2]However,if changes are made to the operation specifications in the system class model, the slicing must be re-done.

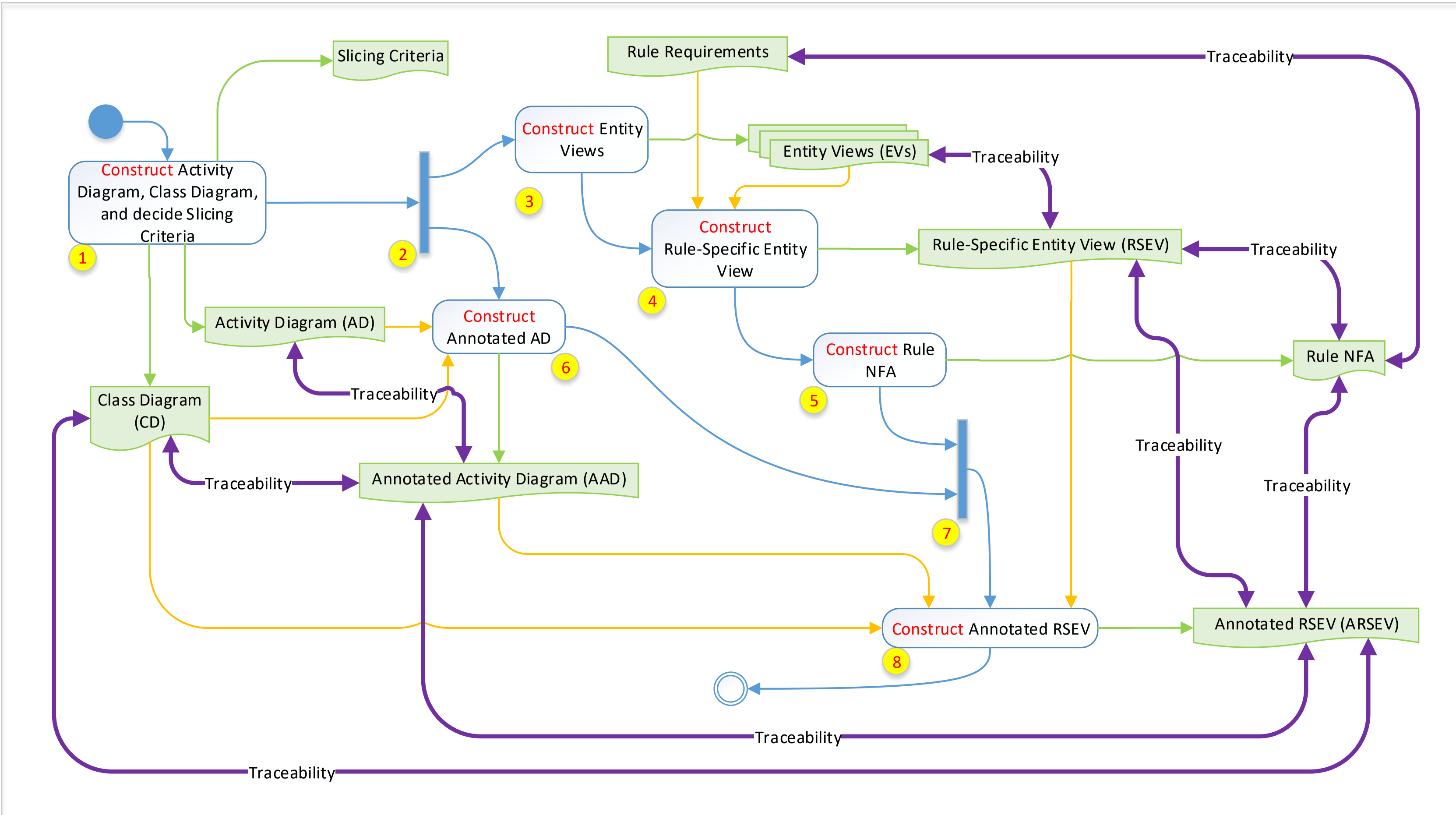


Figure 6.2: Constructing in HMCA

- we have organised some of the internal activities using two parallel paths identified by steps 2-6-7 and 2-3-4-5-7 because the models/artefacts used and produced along these paths do not overlap; and
- at step 8, we require the two parallel paths 2-6-7 and 2-3-4-5-7 to complete in order to construct the ARSEV that depends on the models/artefacts previously produced on the identified paths.

We note that although our explanations in Section 5.1.1 started with a flowchart and its conversion to an activity diagram, HMCA assumes that we will have an activity diagram representation of the system's actions.

6.3 Analyse

Using the same conventions in the key from Figure 6.1, we show a more detailed view of the analysis phase in Figure 6.3 by giving a step-wise decomposition of the internal sub-activities and and include additional models/artefacts used and produced. We use numbers, 1 through 5, to highlight the sub-activities.

The highlights in this phase are that though sub-activities 2 through 4 are not observable externally, they add the models from which we extract the states to use in constructing the *Transition System*. *Slicing* in sub-activity 1 partitions the class diagram using the *Slicing Criteria*; currently we use the operations in the ARSEV as the slicing criteria. We transform each class diagram slice to an equivalent Alloy model. While the class diagram may contain constraints specified using the Object Constraint Language (OCL) [68], these are not automatically transformed in the Alloy model because many of the concepts in OCL are not directly representable in the Alloy language. Thus, adding of the constraints is a manual activity. In addition to the constraints, we add Alloy predicates and assertions to extract state information from the models. The alloy specifications are included in the *Constrained Alloy Slices* (CAS). This equivalent representation of the class diagrams slices as Alloy models help us to undertake detailed analysis to check that operations do not terminate in illegal states, or if they do, to pinpoint where problems in the system specification exist. We then use the states indicated from the execution of the assertions and predicates to construct the transition system in sub-activity 5.

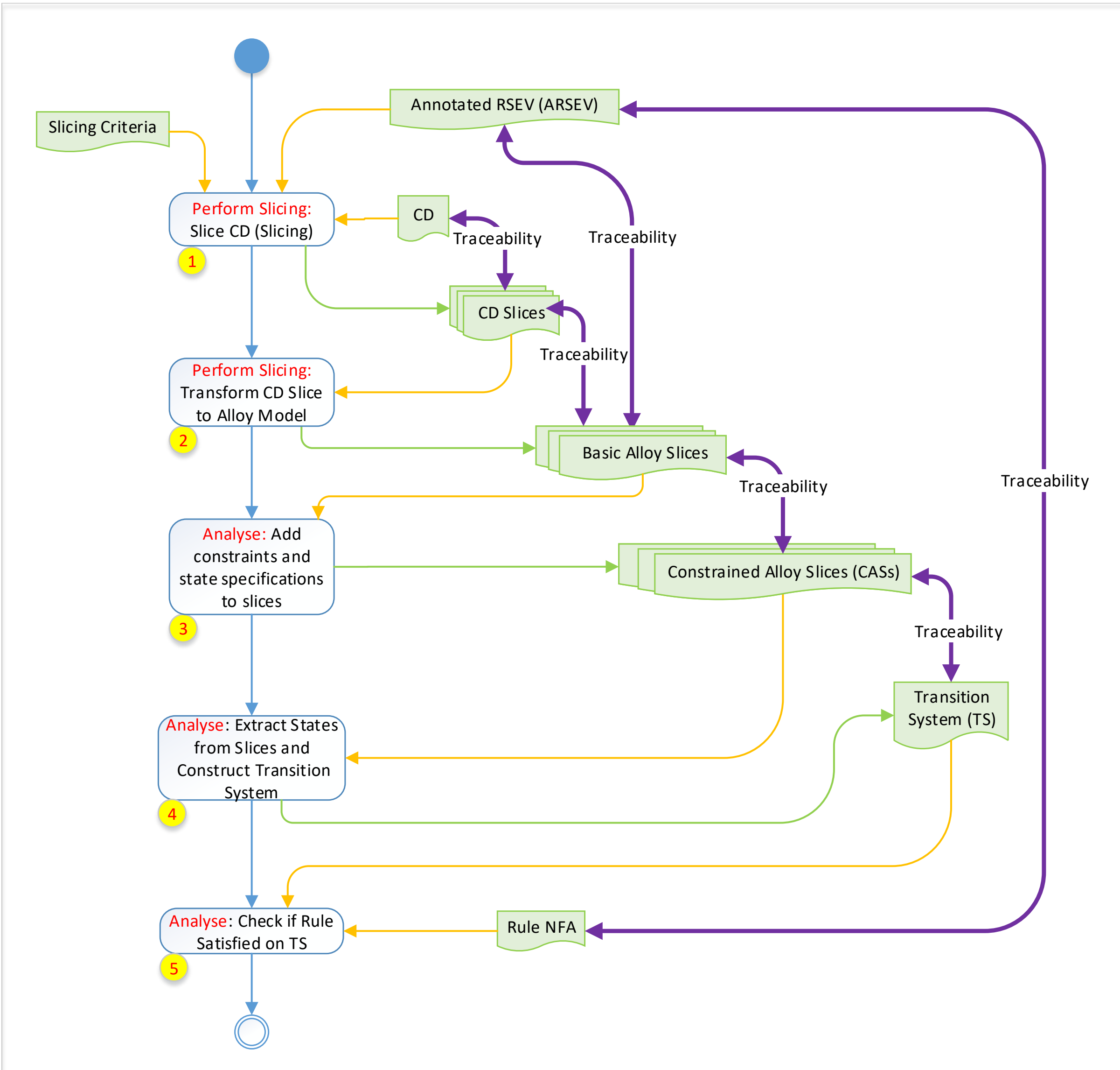


Figure 6.3: Analysing in HMCA

6.4 Provide Feedback

We also show a more detailed view of the feedback phase in Figure 6.4 by giving a step-wise decomposition of the internal sub-activities and include additional models/artefacts used and produced.

We use numbers 1 through 6 to highlight the important steps. We repeat sub-activity 5 from Figure 6.3 as step 1 because its results determines the flow and may be re-used in the feedback phase. Step 2 shows branching flows based on the results in step 1. When a conformance rule is not satisfied, step 3 is taken, otherwise we go to step 5. In step 3, we extract a counterexample from the transition system. The counterexample will indicate structural conditions under which a rule fails and we can use this to modify the constraints in the Alloy model in step 4, and re-analyse the conformance rule in step 1. In step 5, if any of the CASs have been modified, we must reconcile their modifications with the ARSEV, which produces a modified ARSEV. Since each CAS has indirect traceability to the class digram slices, the reconciliation applies to the class diagram as well.

Currently, analysis in HMCA considers each rule individually, so the steps must be followed for each rule. We consider that HMCA is complete on the *No* branch of step 5, or after step 6 completes for all the rules.

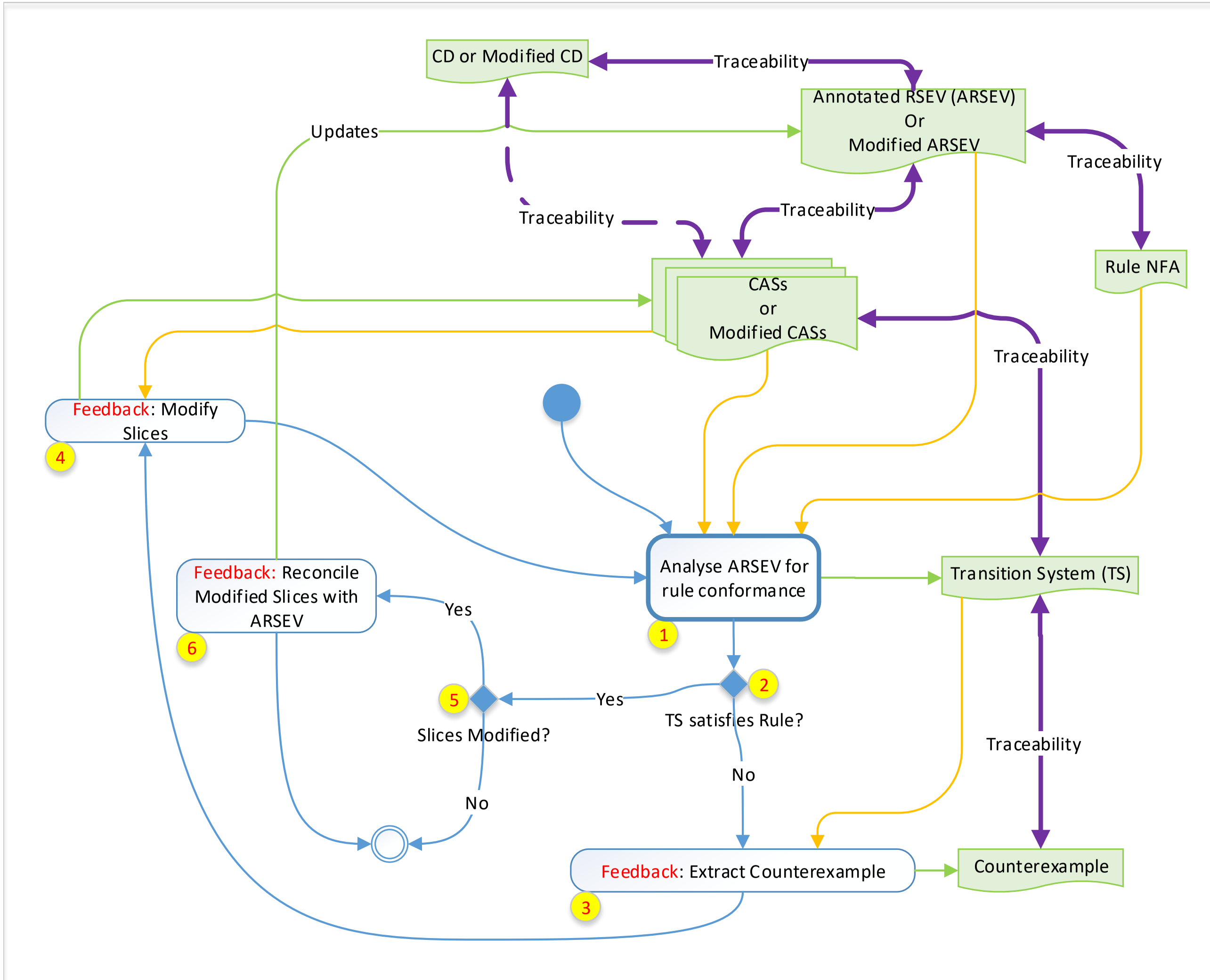


Figure 6.4: Feedback in HMCA

7. NON-CONFORMANCE FEEDBACK

In this chapter we provide additional and updated models for the NJH system to complement the models in Chapter 5, and discuss the feedback phase of HMCA in more details. The models presented will provide the background for sections 7.2 through 9.3. The additional models are in Section 7.1 and the details of the feedback phase are in sections 7.2 through 7.6. We give a summary of the feedback phase in Section 7.7.

7.1 Updating NJH Models

We discuss updates to and include new 1) *entity views*, 2) HIPAA conformance rules as NFAs for the *DeIDed* and the *Identified* access tickets, 3) class model, 4) annotated activity model, and 5) transition systems and non-conforming states in analysis. These were previously discussed in sections 5.1 and 5.2.

7.1.1 Entity Views

Recall that an entity may be understood as an object in the system that either perform operations that change its own states, or is a state of interest to rule conformance. Therefore, entity views are needed to bring understanding to the individual states of entities and how the composition of these individual entity states influence the complete system state. We discussed the entity views and rule specific entity views for the *DeIDed* access ticket in section 5.1.2. However, since we only considered cases where the data used start out in an *Identified* state, our models must be updated to include where the data can start out in a *De-identified state*, i.e., since a project may use data from different sources and some of them may have data that is in an *Identified* or[1] *De-identified* state.

7.1.1.1 Individual Entity Views

Both individual entity views, i.e., the *Patient Health Information Entity View* and the *Researcher Entity View* require updating as we now include new operations and states for the former and new transitions for the latter. We show in Figure 7.1 the updated *Researcher Entity View*. We compare this with Figure 5.1 where we now have a new state for when a researcher is being qualified.

[1] This is to be interpreted as the *inclusive-OR*

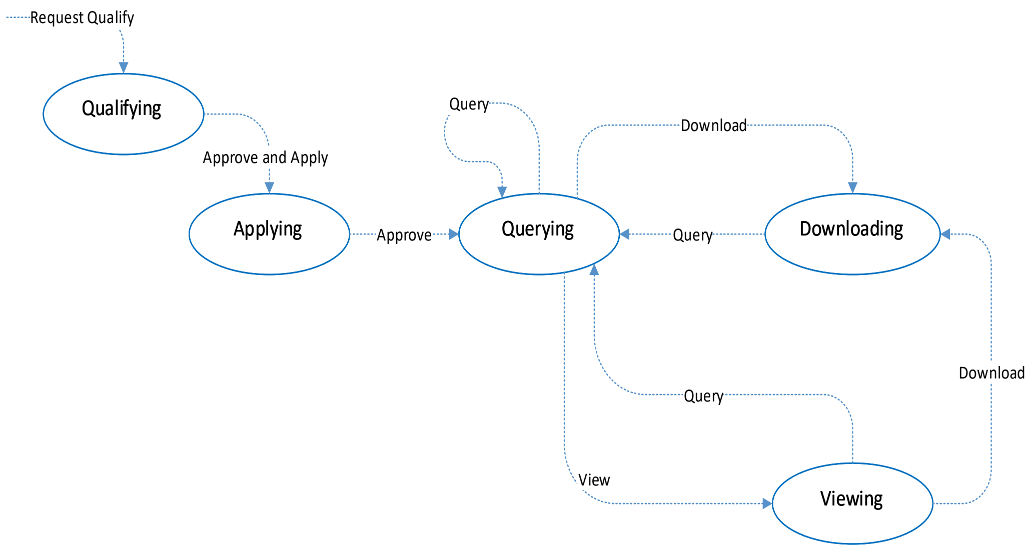


Figure 7.1: Researcher Entity View (Updated from Figure 5.1)

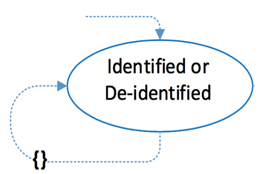


Figure 7.2: Patient Health Information Entity View (Updated from Figure 5.2)

We show in Figure 7.2 the updated *Patient Health Information Entity View*. We compare this to Figure 5.2 where we no longer have a separate *De-identified* state as this is included in the state labelled *Identified or De-identified*. Since the *or* is the *inclusive-OR* the data may be in three distinct states: *only Identified*, *only De-identified* or *both Identified and De-identified*. As with the previous data entity view in Figure 5.2, this view also contains non-determinism as we have not shown the additional conditions from the system state that differentiate the enabling of either of the edges exiting the *Identified or De-identified* state. Though this entity view updates the model for the *DeIDed* access ticket, we note that it also applies to the *Identified* access ticket.

7.1.1.2 Rule Specific Entity View

The changes in the *individual entity views* must be propagated to the *rule specific entity views*. Recall that the latter is constructed based on the *handshaking* [14, see section 2.2.3] of operations in the former,

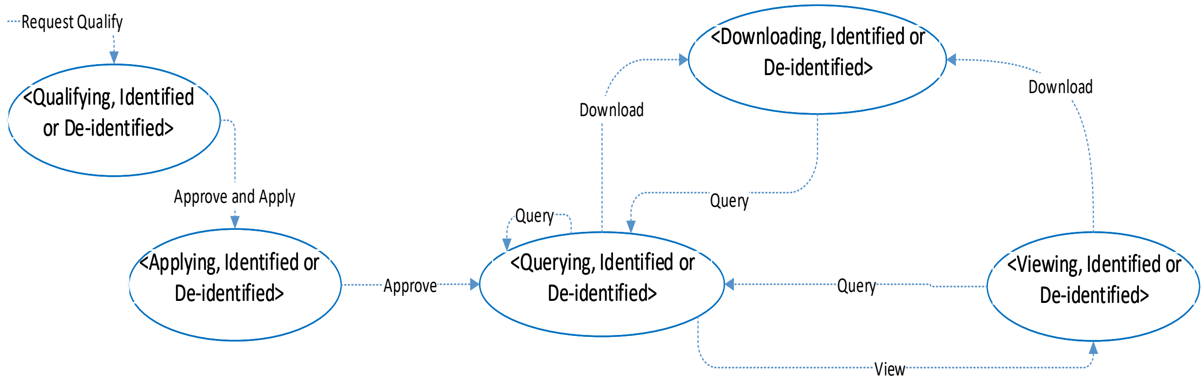


Figure 7.3: *Identified* and *DeIDed* Rules Specific Entity View

such that when identical operations occur on the label of an edge, their next states are combined into one state. We show in Figure 7.3 the composition of the views in figures 7.1 and 7.2. Since both the *Identified* and the *De-identified* states may occur together in the *rule specific entity view*, the composition gives an entity view for both the *Identified* and the *DeIDed* access tickets.

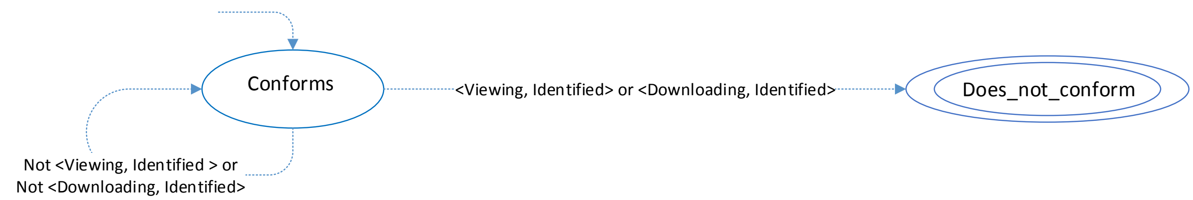
Again, this rule specific entity view contains non-determinism. For example, though there is a single edge from the <*Querying, Identified or De-identified*> state to the <*Downloading, Identified or De-Identified*> state, this (edge) is an abstraction for three edges because of the three different ways the *Identified or De-identified* clause in the states may be assessed to be true. This non-determinism identifies that these possibilities exist in the system at this level of abstraction.

7.1.2 HIPAA Conformance Rules

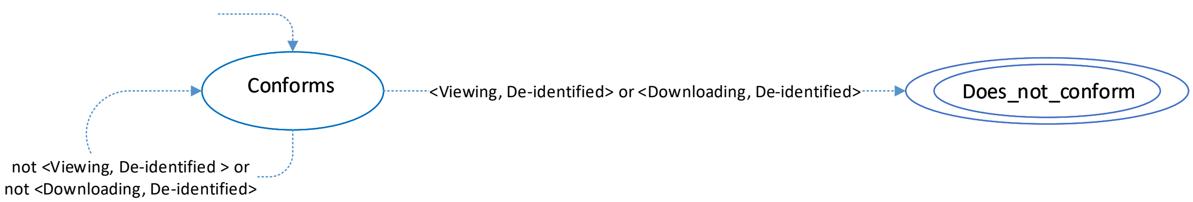
HIPAA conformance rules specify how the system will be adjudged to be conforming to HIPAA regulations. We previously discussed these in Section 5.1.3 and we now return to updating and adding new ones based on the new *rule specific entity view* in Figure 7.3.

7.1.2.1 De-identified *Conformance Rule*

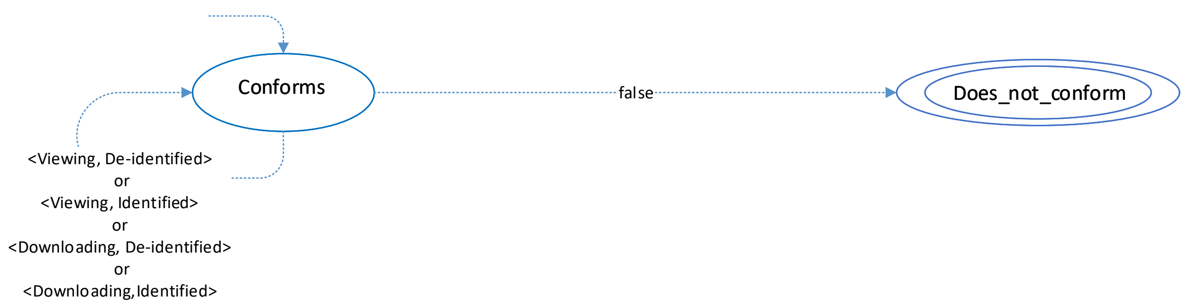
Figure 7.4a is the same as Figure 5.4. It shows the conformance rule for the *DeIDed* access ticket and is specified using the atomic propositions in Figure 7.3. We repeat it here because it will be useful in identifying



(a) Rule NFA for the *DeIDed* access ticket (Same as Figure 5.4)



(b) Rule NFA for the *Identified* access ticket with a *TotallyIDeD* data transformation



(c) Rule NFA for the *Identified* access ticket with an *AllowDeIDed* data transformation

Figure 7.4: Conformance Rules as NFA for the *Identified* and *DeIDed* access tickets

non-confining states for the models in the analysis phase as discussed later in Section 7.1.4. It shows that the system is in state *Conforms* when *View* or *Download* is used to access *De-identified* health information.

7.1.2.2 Identified *Conformance Rules*

Figure 7.4b shows the conformance rule for an *Identified* access ticket requiring a *TotallyIDed* data transformation. It is specified using the atomic propositions in Figure 7.3. It shows that the system is in state *Conforms* when *View* or *Download* is used to access identified health information.

Figure 7.4c shows the conformance rule for an *Identified* access ticket requiring an *AllowDeIDed* data transformation. It is also specified using the atomic propositions in Figure 7.3. It shows that the system is in state *Conforms* when *View* or *Download* is used to access either identified or de-identified health information. We note that, since the *AllowDeIDed* data transformation permits that both the *Identified* and the *De-identified* data states specified in our system to show conformance, there is no case where there can be non-conformance, i.e., the label on the edge into the *Does_not_conforms* state is *false*. This means that all the modelled states of health information will conform to this rule. We also note that when other data transformations are included, e.g., for those allowed by the *Coded* access ticket (see Section 3.4), all the rule formalisms must be updated or else the system will be underspecified due to data states being excluded from the rules and this may result in non-conformance.

7.1.3 Class Models and Activity Model Annotations

7.1.3.1 Class Model

The unsliced class model for the NJH system is shown in Figure 7.5. It includes all the model elements as discussed up to and including Chapter 9.

7.1.3.2 Activity Model Annotations

As we did in Section 5.1.4 to *Map Rule Specific Entity Views to System Models*, where we showed the annotations for the query operation, we now update and add the annotations for all the operations. In particular, we show the annotations for operations that allow the advancing to the different states in Figure 7.3.

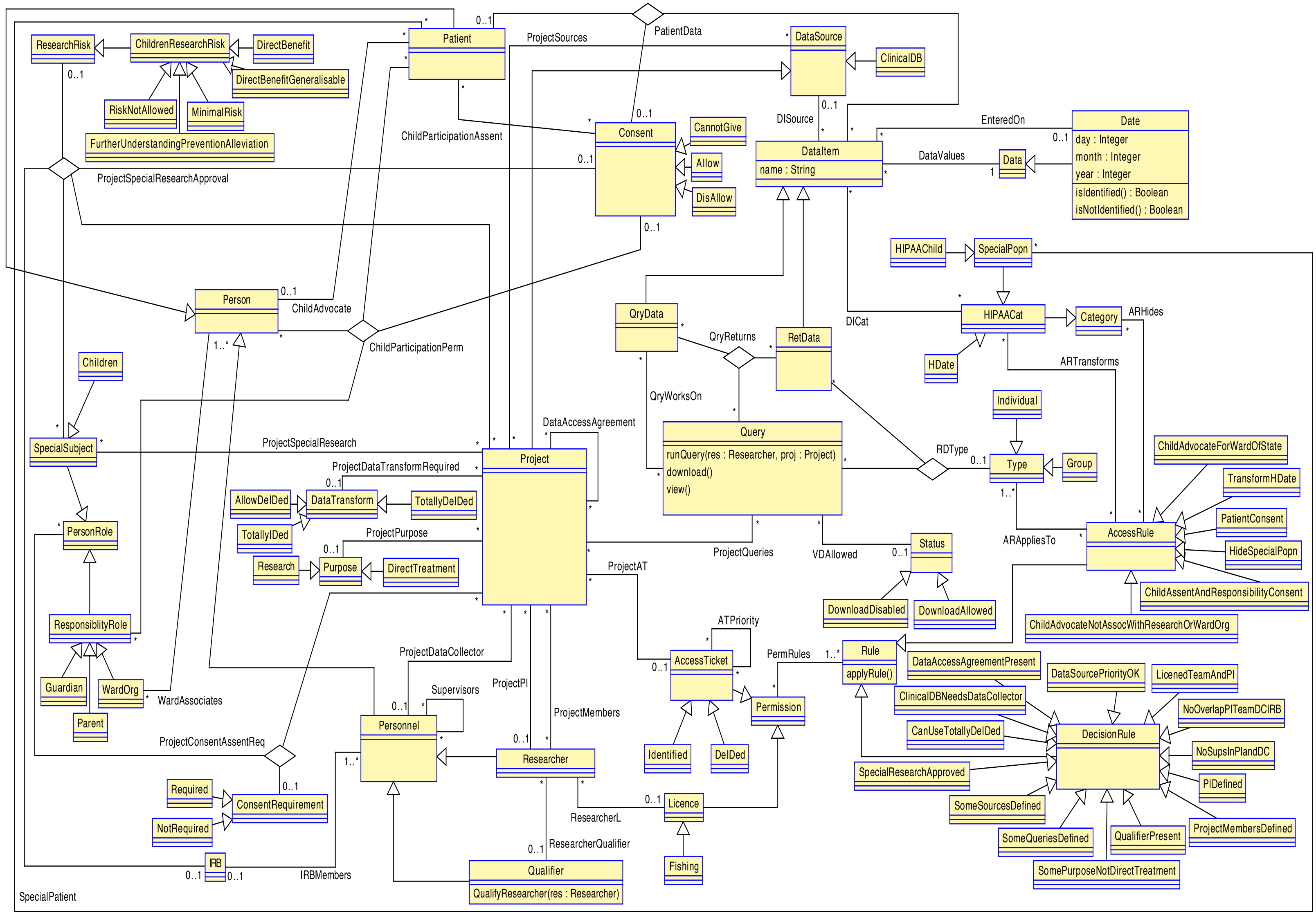


Figure 7.5: NJH Unsliced Class Model: Includes all *AccessRules* and *DecisionRules* and Children as Protected Population

7.1.3.2.1 Approve RequestQualify for Researcher.

- *Input*: Researcher r, Qualifier q;
- *Precondition*:
  1. Personnel that is passed in as Qualifier is Authorised to perform this function; and
  2. there is no link in the *ResearcherQualifier* association between r and q
- *PostCondition*: Link in the *ResearcherQualifier* association between r and q; and
- *Output*: Success.

7.1.3.2.2 Approve a Researcher's Licence Application.

- *Input*: Researcher r, and Licence f;
- *Precondition*:
  1. r qualified; and
  2. there is no link in the *ResearcherL* association between r and f
- *PostCondition*: Link in the *ResearcherL* association between r and f; and
- *Output*: Success.

7.1.3.2.3 Approve a Project's Application for an access ticket.

- *Input*: Project p, AccessTicket at;
- *Precondition*:
  1. All *DecisionRule*s for at return true; and
  2. there is no link in the *ProjectAT* association between p and at (see footrnote[2])

[2]This second condition is sufficient because any p can only have a single access ticket

- *PostCondition*:

  1. Link in *ProjectAT* association between p and at; and
  2. for each *SpecialSubject* linked to p in the *ProjectSpecialResearch* association, there is a link in the *ProjectConsentAssentReq* association

- *Output*: Success.

7.1.3.2.4 Execute a Query.

- *Input*: Query qry, Researcher r;
- *Precondition*: r is authorised to execute qry;
- *PostCondition*: each

  1. *QryData* linked to qry in the *QryWorksOn* association, all applicable *AccessRules* for qry's access ticket returns true; and
  2. *RetData*, rd, in the *QryReturns* association is:

     (a) is transformed according to the *DataTransform* linked to qry in the *ProjectDataTransform-Required* association through its associated project;

     (b) linked to some *QryData*, qd, in *QryWorksOn* for qry;

     (c) linked to some *Type* in *RDType* such that

     if rd is linked to 1 qd in *QryWorksOn* then

     $Type = Individual$

     else

     $Type = Group$

     (d) is *Identified* or *De-identified*.

- *Output*: Success

7.1.3.2.5 *Check Conformance.*

- *Input*: Query *qry*;
- *Precondition*: *qry* has *RetData* in the *QryReturns* association;
- *PostCondition*: for the applicable conformance rule, each *RetData* linked to *qry* through the *QryReturns* association does not return the *Does_not_conform* state[3]; and
- *Output*: conformance rule state.

7.1.3.2.6 *View (or Download) query's results.*

- *Input*: Researcher *r*, Query *qry*;
- *Precondition*:
  1. *qry* has *RetData* in the *QryReturns* association; and
  2. *r* is authorised to view (or download as applicable) *qry*'s *RetData* in *QryReturns*.
- *PostCondition*: true; and
- *Output*: Success.

7.1.4 Analysis

This section completes the models for the *Analysis* phase as previously discussed in Section 5.1.6.

7.1.4.1 *Slicing*

Recall that we use *Slicing* to partition the class model in Figure 7.5 in order for HMCA to produce tractable analysis. We use the operations discussed in Section 7.1.3.2 as the *slicing criteria* to produce 5 slices as follows to:

1. *qualify a researcher* in slice 1 that is produced using the annotations in Section 7.1.3.2.1. This slice is shown later in Figure 7.10.

[3] We are able to determine the applicable conformance rule (as specified in Figure 7.4) indicated by the access ticket and *DataTransform* linked to *qry* through its associated project in the *ProjectAT* and *ProjectDataTransformRequired* associations respectively.

2. *approve a researcher's application for fishing licence* in slice 2 that is produced using the annotations in Section 7.1.3.2.2. This slice is shown later in Figure 7.11.

3. *approve a project's access ticket* in slice 3 that is produced using the annotations in Section 7.1.3.2.3. This slice is shown later in Figure 9.1.

4. *execute a query* in slice 4 that is produced using the annotations in Section 7.1.3.2.4. This slice is shown later in Figure 9.7. With reference to Figure 7.1, the *View* and *Download* operations also occur in slice 4.

5. *check conformance to the HIPAA regulations* in slice 5 that is produced using the annotations in Section 7.1.3.2.5. This slice is shown later in Figure 8.2.

7.1.4.2 Transition Systems

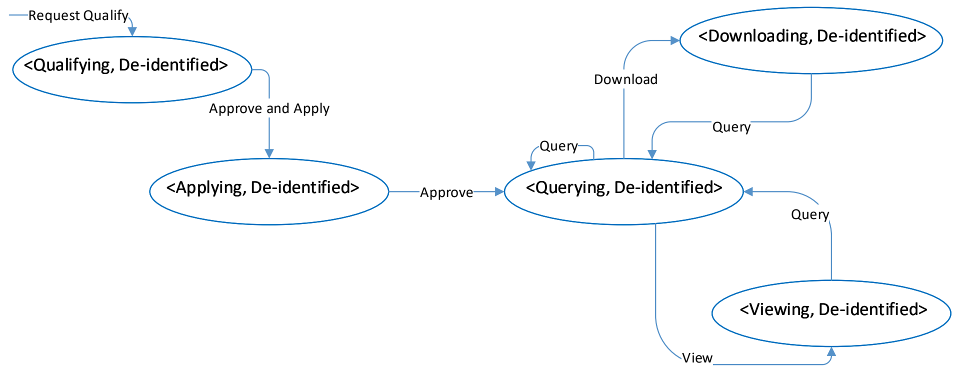
Recall that the *rule specific entity view* in Figure 7.3 is a *program graph* that represents the possible states and operations in the system. Recall also that a transition system (TS) is a concrete representation of the actual reachable states or operations in the execution of the *program graph*.

Through a process of *unfolding* a TS is constructed from Figure 7.3 to produce Figure 7.6. Note that we show the TS as 3 separate subfigures to represent the different starting concrete values in the data states. Figure 7.6a shows the TS where the data starts in a *De-identified* state, Figure 7.6b shows the TS where the data starts in a *Identified* state, and Figure 7.6c shows the TS where the data starts both in the *Identified* and the *De-identified* state.

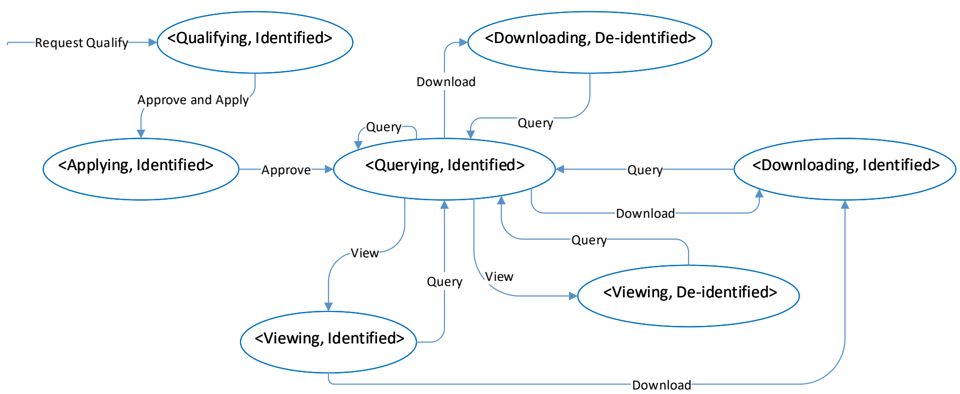
7.1.4.3 Understanding Non-Conformance

Using the rules in Figure 7.4 we determine which states in each of the TSs in Figure 7.6 indicate non-conformance to the rules.

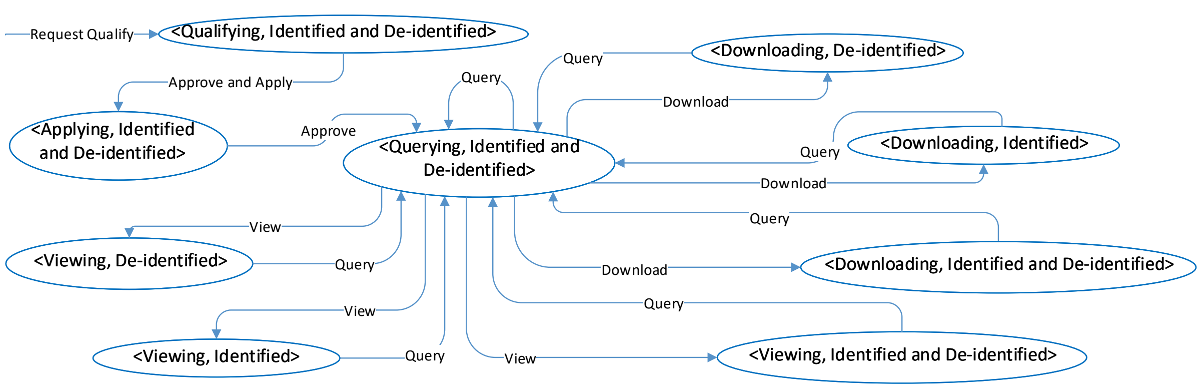
7.1.4.3.1 DeIDed *access ticket.* For the *DeIDed* access ticket with a *TotallyDeIDed* data transformation, the states highlighted in red in the subfigures of Figure 7.7 will cause the de-identified conformance rule in Figure 7.4a to enter the *Does_not_conform* state. For example, Figure 7.7a shows that if the data starts out



(a) TS where data begins in the *De-identified* state

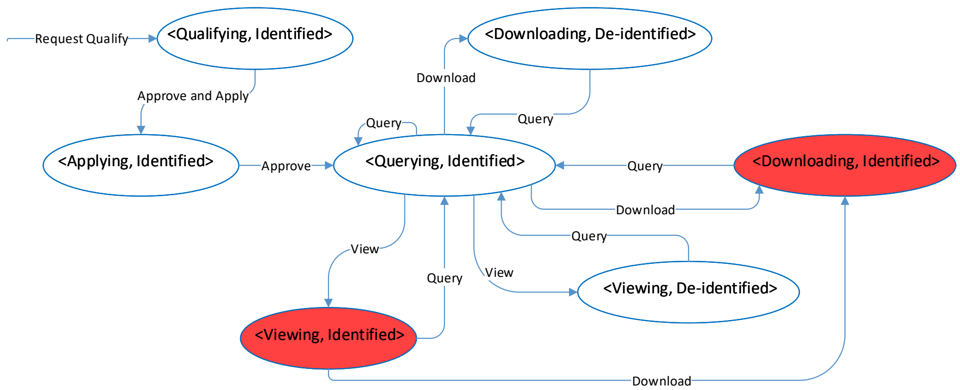


(b) TS where data begins in the *Identified* state

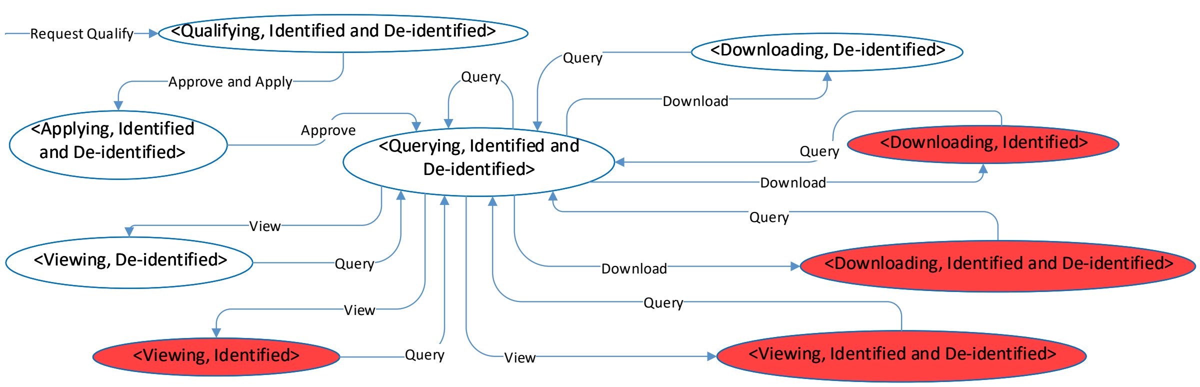


(c) TS where data begins in the both the *Identified* and *De-identified* states

Figure 7.6: Conformance Rules as Graph Formalisms for the *Identified* and *DeIDed* access tickets



(a) Illegal states for the *DeIDed* access ticket with a *TotallyDeIDed* data transformation for the TS where data begins in the *Identified* state



(b) Illegal states for the *DeIDed* access ticket with a *TotallyDeIDed* data transformation for theTS where data begins in the both the *Identified* and *De-identified* states

Figure 7.7: Illegal states for the *DeIDed* access ticket

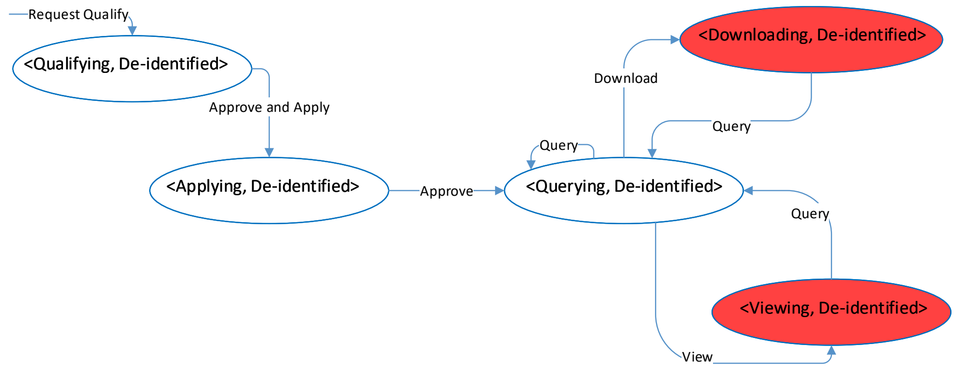
in the *Identified* state we know that *Viewing* or *Downloading* a query's result that is still in the *Identified* state is non-conformance. An example of finding this non-conformance is shown in Figure 7.15 where we showed non-conformance using a counterexample generated using the Alloy Analyzer and an equivalent representation in Figure 7.17 using a UML object model (in Chapter 7). We note that the TS in Figure 7.6a has no illegal states for the *DeIDed* access ticket since all its states are *De-identified.*

7.1.4.3.2 Identified *access ticket with a* TotallyIDed *data transform.* For the *Identified* access ticket with a *TotallyIDed* data transformation, the states highlighted in red in the subfigures of Figure 7.8 will cause the rule in Figure 7.4b to enter the *Does_not_conform* state. For example, Figure 7.8a shows that if the data starts out in the *De-identified* state, we know that *Viewing* or *Downloading* a query's result will show non-conformance because it is impossible to re-identify *Deidentified* data. In addition, Figure 7.8b shows that if the data starts out in the *Identified* state, we know that *Viewing* or *Downloading* in a *De-identified* state is evidence of non-conformance. An example of finding this non-conformance is shown in Figure 8.3 (in Chapter 8).

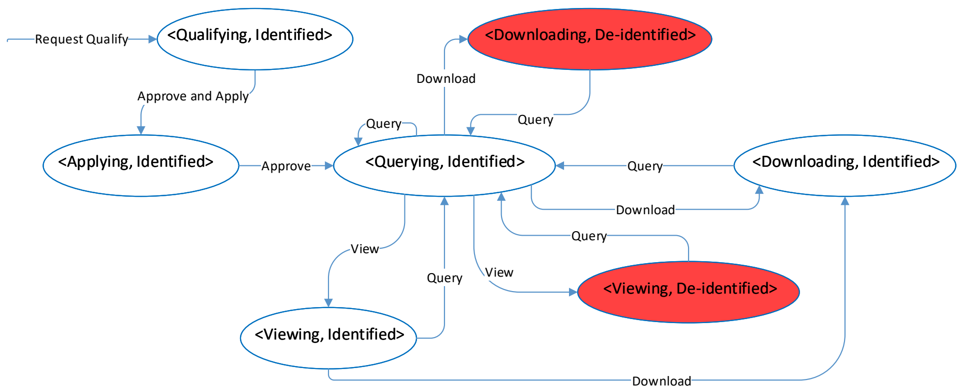
7.1.4.3.3 Identified *access ticket with a* AllowDeIDed *data transform.* For the *Identified* access ticket with a *AllowDeIDed* data transformation, none of the data states in the TSs as shown in Figure 7.6 will indicate non-conformance. This is because this access ticket an its accompanying data transformation permits both the *Identified* and *De-identified* data states.

7.2 Feedback Context and Overview

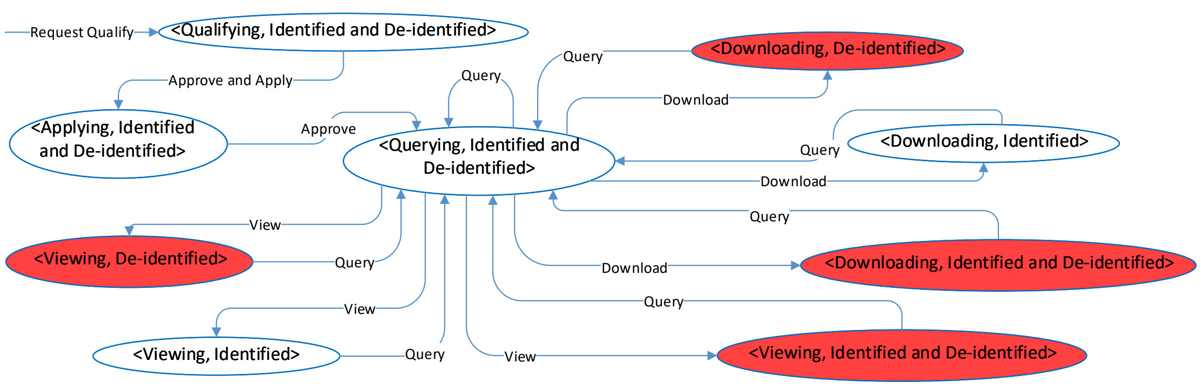
In HMCA we use the Alloy Analyzer to generate a counterexample when a rule is not satisfied. We wish to show the feedback in a format that is easier to understand so we will convert the Alloy counterexample to an equivalent UML object model. However the object model created from the Alloy counterexample may not have enough information in it to understand why non-conformance occurs because the counterexample is an instance of the slice in which the checking of the rule occurred. For example, we show the current UML class model to support operations of interest for the NJH system in Figure 7.9 and in Figures 7.10 through 7.14 the objects and associations from Figure 7.9 that are in each slice.



(a) Illegal states for the *Identified* access ticket with a *TotallyIDed* data transformation for the TS where data begins in the *De-identified* state



(b) Illegal states for the *Identified* access ticket with a *TotallyIDed* data transformation for the TS where data begins in the *Identified* state



(c) Illegal states for the *Identified* access ticket with a *TotallyIDed* data transformation for theTS where data begins in the both the *Identified* and *De-identified* states

Figure 7.8: Illegal states for the *Identified* access ticket with a *TotallyIDed* data transformation

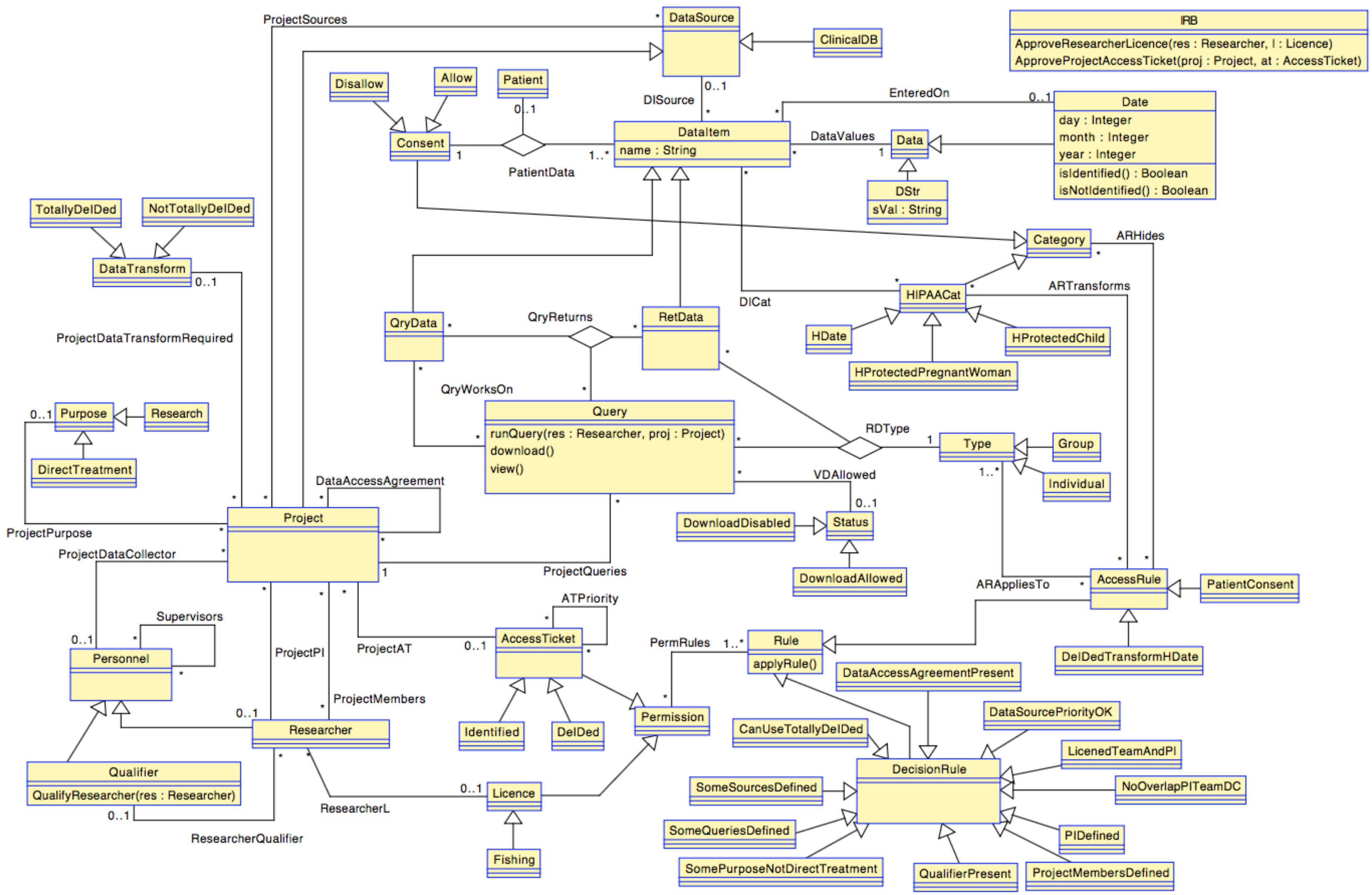


Figure 7.9: NJH Class Model: Capturing Model Elements for Qualifier Researcher to Checking Access Ticket Conformance on Query Results

If the following sequence of operations occurred:

personnel per_1 **qualifies researcher** r (slice 1, S_1, in Figure 7.10)

→ **approve researcher** r **for fishing licence** f (slice 2, S_2, in Figure 7.11)

→ **approve project** p_1 **to use** d**,** q_1 **is a part of** p_1**'s queries, researcher** r **is a project member in** p_1 (slice 3, S_3, in Figure 7.12)

→ **researcher** r **runs query** q_1 **using** d (slice 4, S_4, in Figure 7.13)

→ **check conformance to de-identified access ticket** d**, for the results from query,** q_1 (slice 5, S_5, in Figure 7.14)

and, if conformance failed in slice 5, the counterexample only contains instances of elements in that slice. However, the user may need an object model of the full system model to determine the reason for non-conformance.

In order to give the user enough information to determine the reason for non-conformance, we will show the feedback as a UML object model. To do this, we augment the object model generated from the counterexample with additional objects and links in such a way that is consistent with the constraints of the system class model. The USE tool provides capabilities to create object models and check that they satisfy the constraints in the associated class model. In addition, we can supply the tool with a partial object model and use its generation capabilities to add objects and links to have a valid instance of the associated class model. In order to accomplish this with the USE tool, we must include all the Alloy model constraints that have been created to run the analysis as OCL constraints.

In order to reduce the cognitive overload of showing the object model of the full system all at once, we will sequence the feedback as instances of the slices in figures 7.10 to 7.14. The general procedure is to construct the feedback as an on-demand (user-driven) sequence of object models, starting in the slice that the non-conformance is observed and generating the object model for the previous slices as needed. In each subsequent object model, we will highlight the overlapping objects and links with each previous object model. For instance, in our running example where conformance failed, the first object model in the sequence is the counterexample from slice 5, the second an object model from slice 4, etc. The user will be shown the first object model in the sequence and can request the second, and so on.

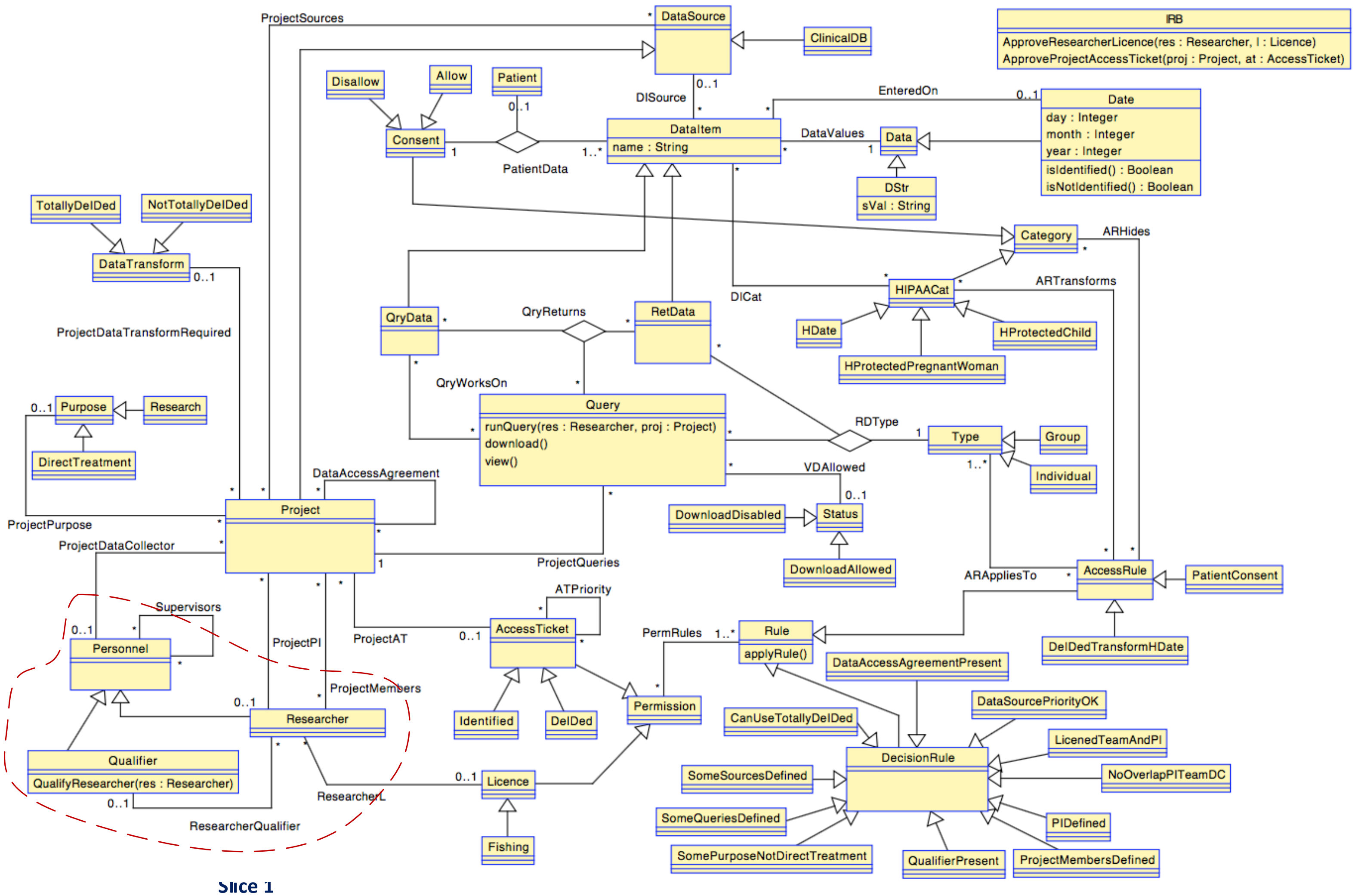


Figure 7.10: Slice 1 (S_1) - Qualifier Researcher Slice

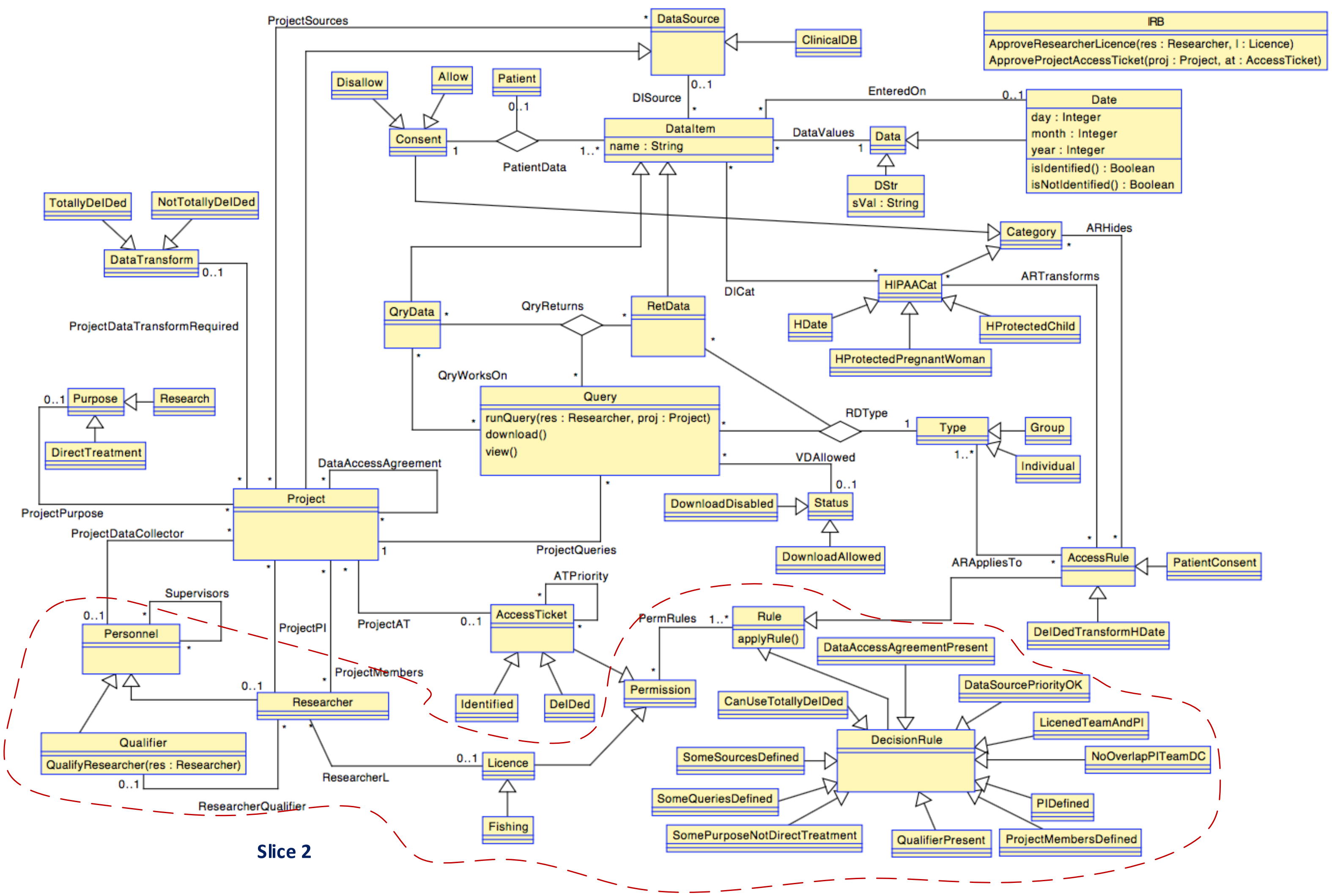


Figure 7.11: Slice 2 (S_2) - Approve Researcher Licence Slice

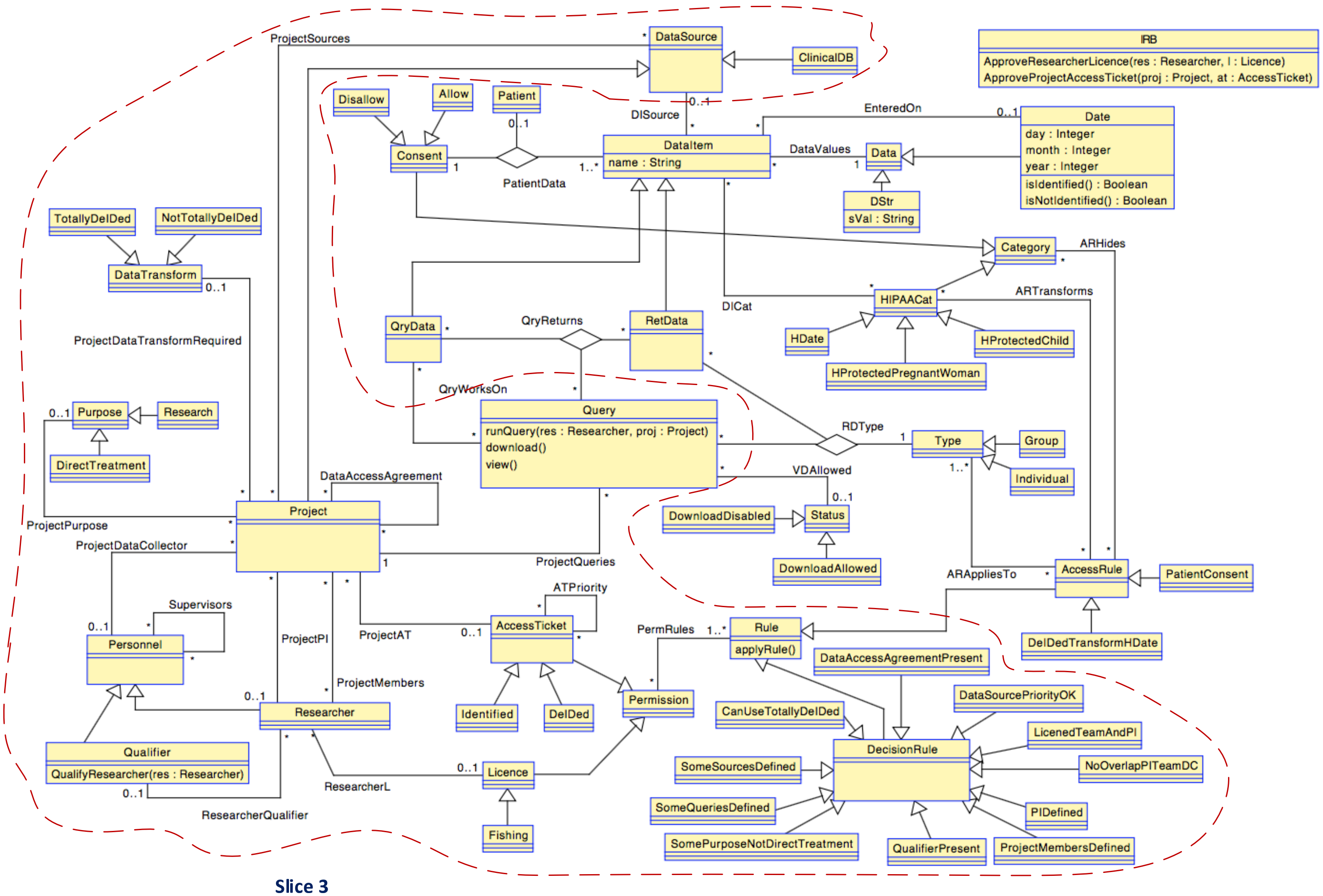


Figure 7.12: Slice 3 (S_3) - Approve Project Access Ticket Slice

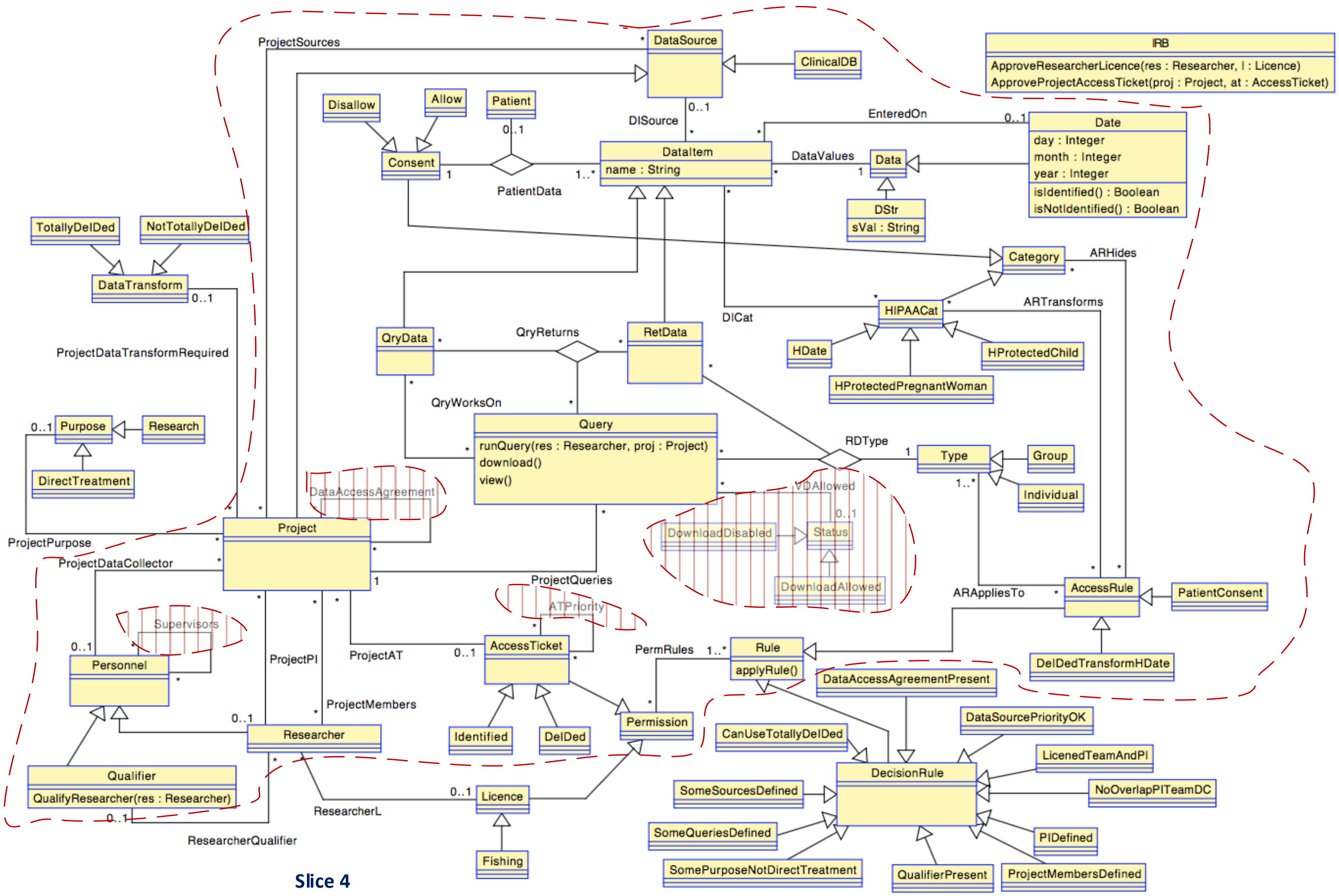


Figure 7.13: Slice 4 (S_4) (excludes shaded areas) - Data Collector, PI, or Researcher Runs Query Slice

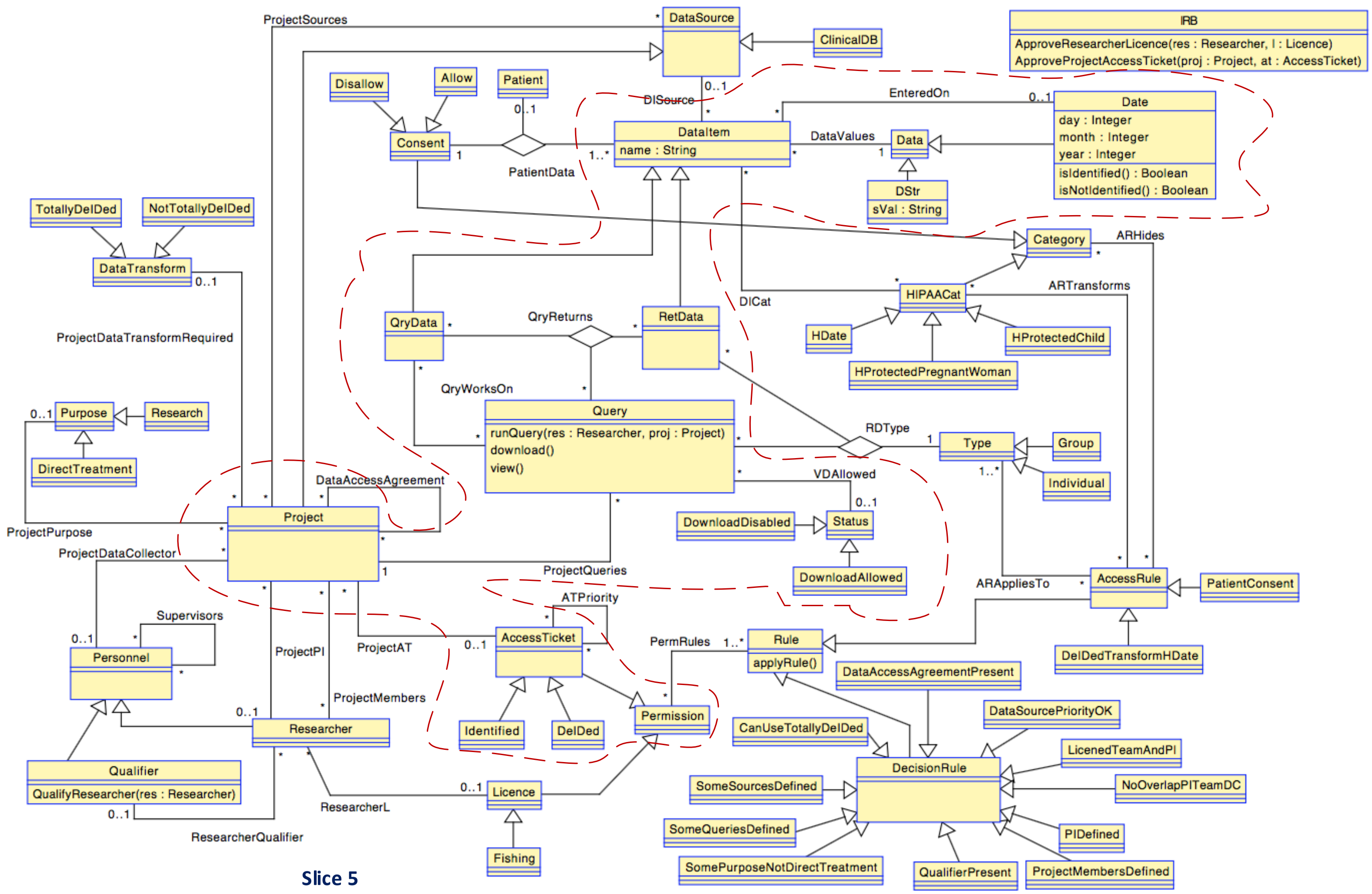


Figure 7.14: Slice 5 (S_5) - Check Conformance Slice

Depending on the size and complexity of the class model and constraints, constructing the feedback in this way may save computations. In the next section we discuss the specific commands that the USE tool provides for generating object models. Section 7.4 discusses the USE specifications and we return to a detailed examination of generating the feedback in Section 7.5. We show in Section 7.6 how the generated object models may be used to analyse and understand why conformance failed. Section 7.7 ends this chapter with some conclusions and future directions.

7.3 USE Tool Object Model Generator

The *USE* tool can generate object models that conform to a class model with OCL constraints. To accomplish this, it employs A Snapshot Sequence Language (ASSL) [41]. ASSL provides additional commands to the OCL and Simple OCL-based Imperative Language (SOIL) languages already included in USE.

SOIL provides commands to *create*, delete, and *insert* objects and links among objects, but does not ensure that the objects and links satisfy constraints in the corresponding class model; so using these commands may produce an ill-formed object model. Using the SOIL language to produce object models produces deterministic models, i.e., the same object model each time the commands are executed.

ASSL commands include equivalent commands provided by SOIL and additional ones that can perform guided searches in the space of objects and insert links among them that satisfy the constraints in the class model. The commands will only report success, i.e., objects and links created will persist, if the object model created satisfy all the constraints, otherwise a rollback occurs and the object model is returned to the state it was before the commands were executed. In this way, we are assured that the object model returned is well-formed. While the SOIL commands may be issued directly in the USE tool, the ASSL commands must be packaged in a procedure and the procedure executed using other special USE commands.

The guided searches of some of the ASSL commands mean that we do not have a deterministic object model, in the same way as using SOIL commands, even if the same ASSL commands are re-executed. In order to produce deterministic object models from ASSL, we can take advantage of how the USE tools logs when an ASSL procedure reports success and generates equivalent SOIL commands to recreate the exact object model that is returned. In addition, the searching for valid states means that executing ASSL procedures may be computationally intensive. Both having the SOIL commands available and not having to re-execute a computationally intensive procedure are important when generating the sequence of object models with overlapping objects and links. For example, when generating the object model for slice 4 the same instances of the overlapping objects and links from slice 5 must be used.

Having the SOIL commands used to create the object model for slice 5 presents an opportunity for reuse because we can extract the commands for the overlapping object and use them as the starting point for generating slice 4.

7.3.1 Object Model Generation Commands

ASSL commands include those to:

1. *Create* objects, e.g.,

   (a) $Create(Personnel)$ to create and return a single $Personnel$ object; and

   (b) $CreateN(Personnel, 5)$ to create and return 5 $Personnel$ objects as a sequence of objects.

   Create gives the objects created arbitrary identifiers.

2. *Delete* objects and associations, e.g.,

   (a) $Delete(Personnel_1)$ to delete the object identified by $Personnel_1$; and

   (b) $Delete(Personnel- > allInstances()- > asSequence())$ to delete all objects of type $Personnel$.

3. *Insert* links between objects to form associations; e.g., $Insert(ResearcherQualifier, p_1, r)$ to add a link between p_1 and r in the $ResearcherQualifier$ association;

4. Randomly generate objects, values, or associations links:

   (a) *Any(seq : Sequence(T))*, to make and return a random selection from a sequence objects or values of type T and use or assign it to a variable of the same type

   (b) $Try(seq : Sequence(T))$ also works like the *Any()* command;

(c) $Try(a : Association,$

$$seq_1 : Sequence(T_1), seq_2 : Sequence(T_2)[, ..., seq_n : Sequence(T_n)]*)$$

to generate random association links among objects from the sequences given.

While we noted that both the *Any*(seq : *Sequence*(T)) and the *Try*(seq : *Sequence*(T)) commands produce the same results, they are semantically different because the latter also checks whether the assignments satisfy the constraints in the class model before returning the object/association. As discussed before, if any of the commands in an ASSL procedure causes the object model to be in an inconsistent state, the procedure will not succeed.

7.4 USE Specifications

The slicing of the class model in the construction phase of HMCA described in Section 6.2 allows us to not only produce the Alloy slices, but to also produce equivalent class model slices. Since we are using the USE tool, the class model slices must be represented in the USE language. This representation may be achieved by employing an algorithm similar to Algorithm 2 that transforms the Alloy counterexample into a USE object model. The constraints that ensured well-formed slices were included in the Alloy specifications. In order for our generation program to work correctly and produce well-formed object models, we must now add the equivalent Alloy constraints to the sliced USE class models using OCL constraints.

Alloy and OCL have many similarities as specification languages and in their associated tools, i.e., the Alloy Analyzer and USE. However one of their main difference is in their support for sets and collections. In OCL sets and other collections are one-dimensional, but in Alloy everything is a set [15]. For this and other differences, it is not always possible to automatically transform Alloy to OCL because several Alloy expressions do not have a one-to-one equivalent in UML or OCL [28]. Since overcoming these challenges are not the focus of this research, the reader may examine the papers for translating Alloy to UML annotated with OCL in [6, 7] and the examination of translation back to Alloy in [28].

We transformed the constraints in the Alloy specifications to OCL manually. Refer to Appendix C for the detailed UML and OCL constraints for each slice. Our manual transformations provided many insights that may be useful not only for the automatic translation of Alloy to OCL, but also for insights on how the difference in their support for sets and collections may produce slightly different associations/relationships and constraints among classes/signatures. We will return to discussing this in Chapter 11 where we give insights into the details of applying HMCA.

7.5 Detailed Algorithms: How to Construct the Object Model for the Feedback

Algorithm 1 outlines the high-level steps we will take to generate and request on-demand object models. It makes reference to Algorithm 2 to convert an Alloy instance to a USE object model, Algorithm 3 to extract overlapping objects from object models, and Algorithm 4 to complete an object model so that it satisfies the constraints in the class model. The first is outlined in Section 7.5.1, the second and third in Section 7.5.2. The ASSL procedures and USE commands that implement the algorithms for the NJH system are listed in Appendix C.

Algorithm 1 Generate On-Demand Feedback Object Model Sequence Construction

procedure OnDemandFeedback($cm_{use} : USEClassModel,$
  $cm_{seq} : Sequence < USEClassModel >, inst_{aa} : AlloyInstance$)
  $current \leftarrow cm_{seq}.first()$
  $om_{use} \leftarrow ConvertAlloyInstanceToOM(inst_{aa})$ ▷ See Algorithm 2 in Section 7.5.1
  $Show(om_{use})$ ▷ displays object model
  $getNext \leftarrow UserRequestsNext()$ ▷ $UserRequestsNext()$ is a Boolean value
  while $getNext \wedge cm_{seq}.hasNext()$ **do**
    $current \leftarrow current \cup cm_{seq}.getNext()$
    $om_{use} \leftarrow ExtractOverlappingObjects(current, om_{use})$ ▷ See Algorithm 3 in Section 7.5.2
    $om_{use} \leftarrow CompleteFeedback(current, om_{use})$ ▷ See Algorithm 4 in Section 7.5.2
    $Show(om_{use})$
    $getNext \leftarrow UserRequestsNext()$

7.5.1 Represent Alloy Slice as a UML USE Object Model

Algorithm 2 outlines the steps to convert an Alloy instance to an object model.

7.5.2 Generate Feedback as a Complete Object Model

Algorithm 3 to Algorithm 7 gives the steps to generate a complete an object model with the objects and associations to satisfy a given class model.

Algorithm 2 Convert Alloy Instance to USE UML Object Model

function ConvertAlloyInstanceToOM($aa : AlloyInstance$)
    $init(om)$ ▷ om initialised to type $USEObjectModel$
    for $sigs \in aa.getSignatureInstances()$ **do**
        $om \leftarrow om \cup !new(s.getSigType(), s.getSigName())$
        ▷ $!new()$ translates to the Soil command: $!new\ Class(object\ identifier)$
    for $rels \in aa.getRelations()$ **do**
        $inst_{sigs} \leftarrow rel.getRelationSignatureInstances()$ ▷ returns ordered signature instances
        if $inst_{sigs} \not\subseteq sigs$ **then**
            error
        $om \leftarrow om \cup Insert(rel.getName(), inst_{sigs}[1], inst_{sigs}[2][, ..., inst_{sigs}[n]]*)$
        ▷ See Section 7.3 for notes on $Insert()$
    return om

Algorithm 3 Extract Overlapping Objects

function ExtractOverlappingObjects($cm_{use} : USEClassModel, om_{use} : USEObjectModel$)
    init($om_{partial}$)
    $assocs \leftarrow \{a : Association \mid a \in cm_{use}.getAssociations()\}$
    for $a \in assocs$ **do**
        $om_{partial} \leftarrow om_{partial} \cup om_{use}.getMappings(a)$
    return $om_{partial}$

Algorithm 4 Complete Feedback

function CompleteFeedback($cm_{use} : USEClassModel, om_{use} : USEObjectModel$)
    if $cm_{use}.unconstrained() \not\models om_{use}$ **then**
        ▷ ensures that all objects and associations in om_{use} have corresponding definitions in cm_{use}
        error
    $a_{diff} \leftarrow \{a : Association \mid a \in cm_{use}.getAssociations() \wedge instance(a) \notin om_{use}\}$
    $obj_p \leftarrow CreatePotentialObjects(om_{use}, a_{diff})$ ▷ See Algorithm 5
    $om_{use} \leftarrow om_{use} \cup obj_p$
    $om_c \leftarrow om_{use}$
    repeat
        for $a \in a_{diff}$ **do**
            $om_c \leftarrow CreatePotentialAssociations(om_c, a)\}$ ▷ See Algorithm 6
    until $cm_{use}.constrained() \models om_c$
    $om_c \leftarrow Cleanup(om_c, obj_p)$ ▷ See Algorithm 7
    AcceptObjectModel(om_c) ▷ makes objects and associations added permanent
    return om_c

Algorithm 5 Create Potential Objects

function CreatePotentialObjects($assocs : Set < Association >$)
  init(c_{diff}) ▷ $c_{diff}$is initialised to $Map < Class, Value < Integer, Integer >>$
  for c: assocs.getAssociationEnds().getClasses() **do**
    $c_{diff}.put(c, 0, 0)$
  for $a : assocs$ **do**
    ▷ iterates through the multiplicities of the association ends to compute the min and max instances required
    for $< ae : a.getAssociationEnds() >$ **do**
      $c \leftarrow cmm.get(ae.getClass())$
      $c.value.first \mathrel{+}= ae.minMultiplicity()$
      $c.value.second \mathrel{+}= +ae.maxMultiplicity()$ ▷ if multiplicity is * then 0 is returned
  $obj_p \leftarrow \{\}$
  for $< entry : c_diff >$ **do**
    ▷ the following if statements updates first and second values to ensure that we create at least 1 of each missing object
    if $entry.value.first = 0$ **then** $entry.value.first \leftarrow 1$
    if $entry.value.second = 0$ **then** $entry.value.second \leftarrow entry.value.first$
    $obj_p \leftarrow obj_p \cup Create(entry.key, Any([Sequence\{c.value.first()..c.value.second()\}]))$
    ▷ See Section 7.3 for notes on $Create()$ and $Any()$
  return obj_p

Algorithm 6 Create Potential Associations

function CreatePotentialAssociations(om: ObjectModel, $assoc : Association$)
  init(seq) ▷ seq is initialised to $Sequence < Sequence < Object >>$
  $i \leftarrow 1$
  for $c : Class \in assoc.getClasses()$ **do**
    $seq[i] \leftarrow om.getObjects(c).asSequence()$
    $i \mathrel{+}= 1$
  $Try(assoc, seq[1], .., seq[n])$ ▷ 1's based indexing assumed
  return om

Algorithm 7 Cleanup Object Model - Delete Unused Potential Objects

function CleanUp($om : USEObjectModel, o_p : Set < Object >$)
  $om \leftarrow Delete(o_p - om.getAssociations().getAssociationEnds.getObjects()))$
  ▷ See Section 7.3 for notes on $Delete()$
  return om

7.6 Examining Object Models

Suppose, in our analysis of the Alloy model, conformance fails and gives us the counterexample in Figure 7.15. We see that while the *Query$0* was executed with a *DeIDed* access ticket, we are barred from downloading its result, i.e., the *VDAllowed* relation links *Query$0* with *DownloadDisabled$0*. Further examination of the counterexample shows that:

1. downloading the query's results is disabled because *DataItem$3*, whose *DataValue* is *Date$1*, has not been (properly) de-identified, this is highlighted using the blus dashed line;

2. *DataItem$3* was derived from *DataItem$5*, i.e., the edge from *Query$0* to *DataItem$5* shows that the *qryReturns* relations links these instances with *DataItem$3*) ; and

3. other return data (*DataItem$0, DataItem$1, and DataItem$2*) have been derived from *DataItem$4*, i.e., shown on the edges from *Query$0* to *DataItem$4*, but these have been properly de-identified.

While the user executing the query may be disappointed/inconvenienced that the results of the query are not available, the system owners/administrators will be relieved that conformance according to the *DeIDed* access ticket has been demonstrated (verified). However, the system administrator will be concerned that this scenario occurred and should investigate. HMCA's next step will allow the administrator to examine object models along the path to the non-conformance to try to determine the reason that *DataItem_3*'s *DataValue* is returned identified.

Recall that we identified an equivalent class model for the counterexample as slice 5 (S_5) in Figure 7.14; we now show this slice as a separate class model in Figure 7.16. While only the object model is shown to the user, we include the class model as a reference and note that this too may be included in the on-demand feedback to give a further context for each object model. Following Algorithm 2, *Convert Alloy Instance to USE UML Object Model*, we construct its equivalent object model in Figure 7.17.

This object model contains all the instances of the signatures and relations in the Alloy counterexample. The failure is circled by a blue dashed line. Beyond showing that conformance was violated, this object model is not helpful in identifying why conformance fails. Therefore, we ask the system to give us the previous slice in which the query was executed. The slice in which the query was executed was identified as slice 4 (S_4) in

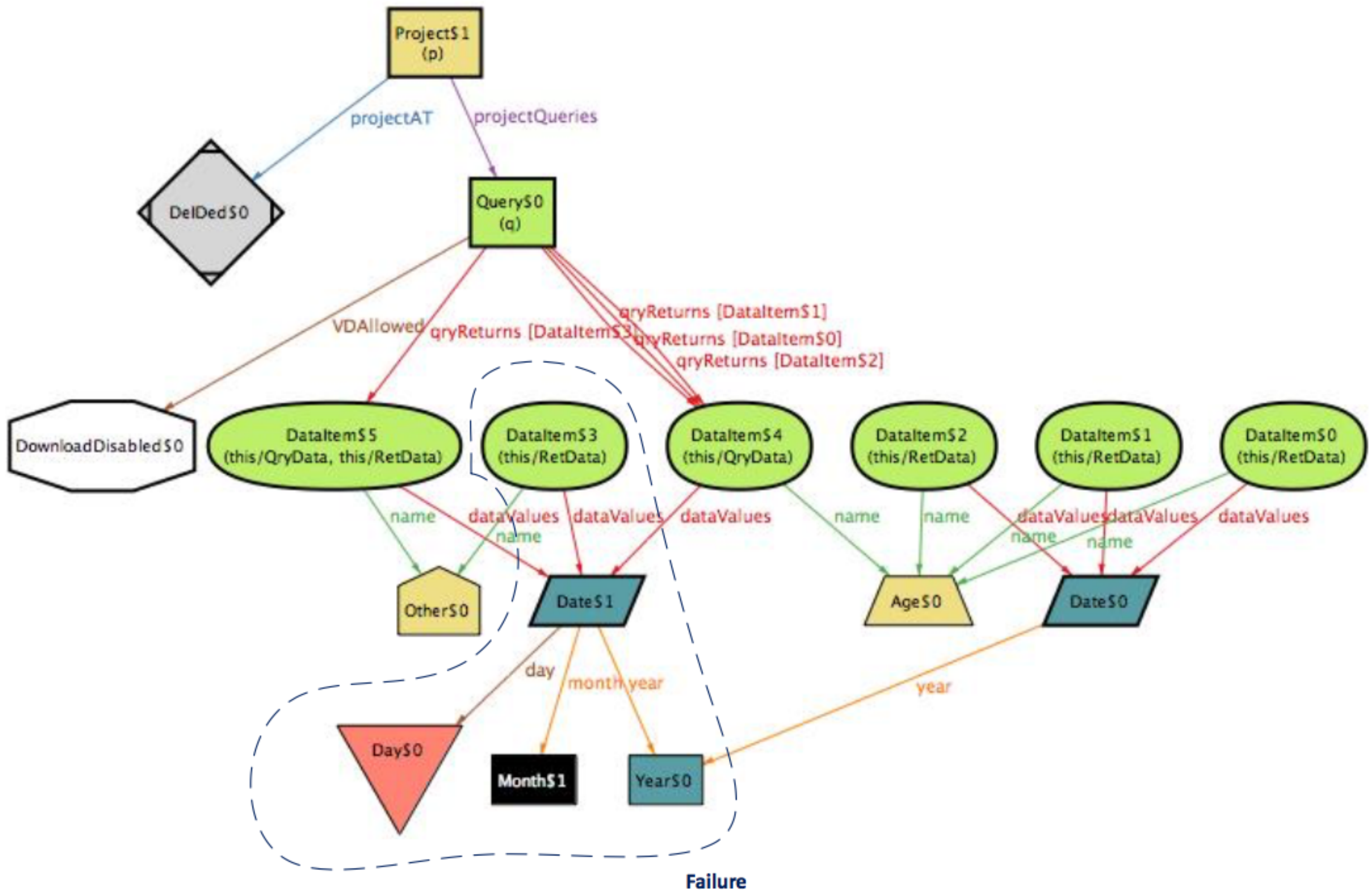


Figure 7.15: Alloy Analyzer Conformance Counterexample in Slice 5

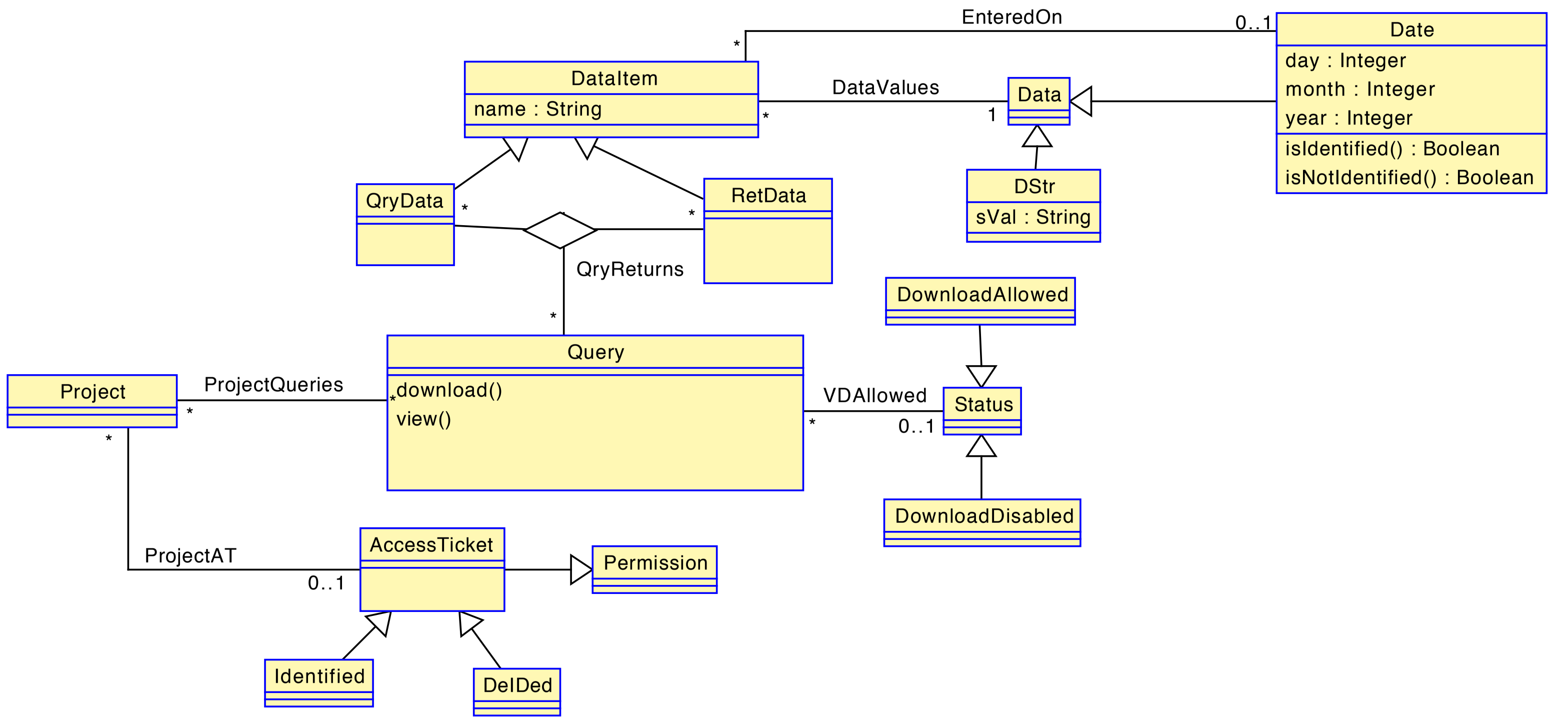


Figure 7.16: Class Model for Slice 5

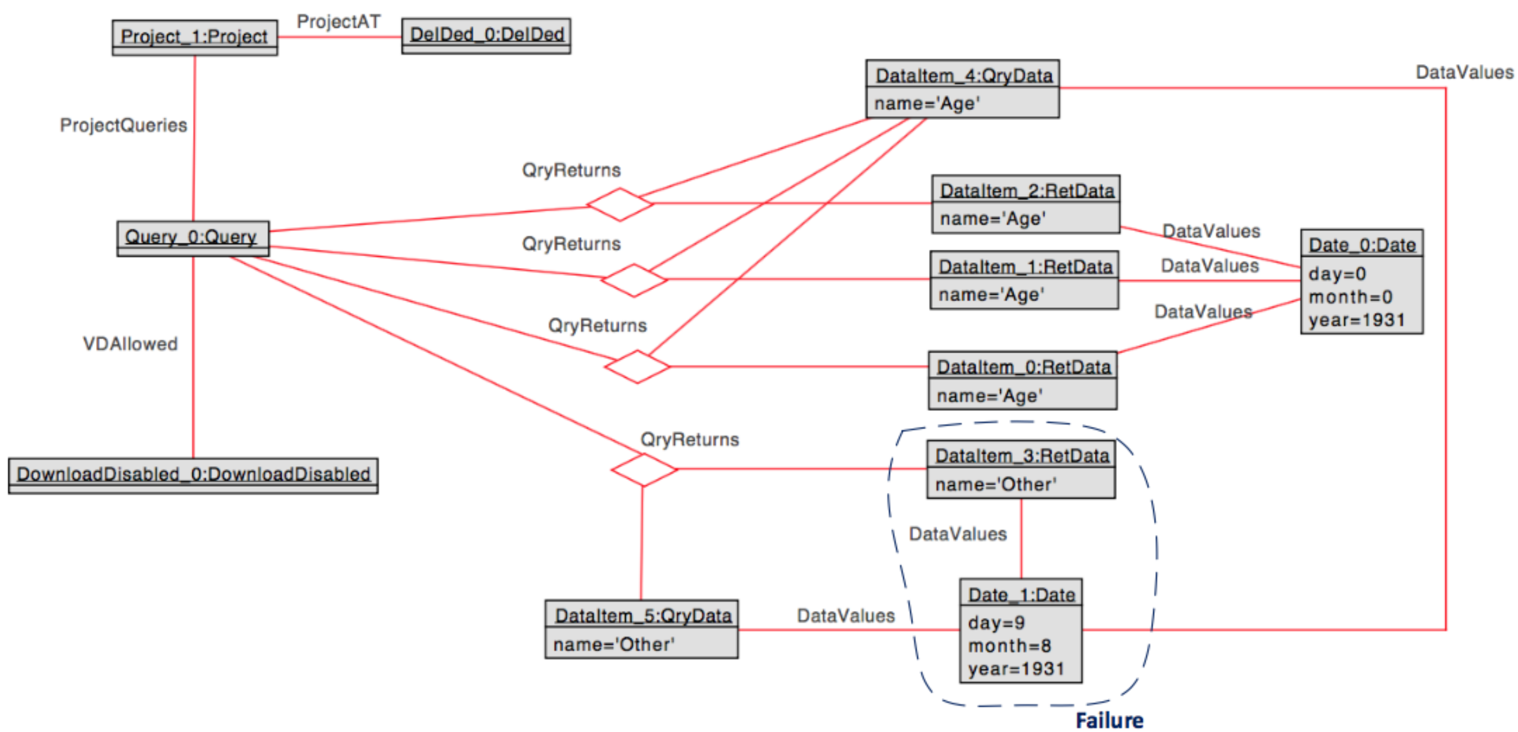


Figure 7.17: Non-Conformance Object Model for Slice 5

Figure 7.13; we show it as a separate class model in Figure 7.18 and outline the class model elements that overlap with the class model for slice 5 (in Figure 7.16). We use Algorithm 3, *Extract Overlapping Objects*, to extract the overlapping objects and links from Figure 7.17, i.e., the objects and links that are instances of the overlap of the slices highlighted in Figure 7.18. We then pass the class model in Figure 7.18 and the object model returned from Algorithm 3 to Algorithm 4, *Complete Feedback*, to generate an object model satisfying Figure 7.18. Note that for representing the Alloy counterexample as an object model we made a change to how dates are presented.

In Alloy a de-identified date is one that has a value for *year*, but does not have a value for neither *day* nor *month*. In OCL we modelled a de-identified date as having a non-zero *year*, $day = 0$ and $month = 0$. We then add the required instances of the other model elements to satisfy the constraints of slice 4. We show this object model in Figure 7.20 and use a grey shading to highlight the objects and links that overlap with the objects and links in the object model for slice 5. We also outline and label the failure using a blue dashed line/ font and that show the data that have been correctly de-identified using a red dashed line/font.

We identified in the previous slice (slice 5) that the return data derived from *DataItem_4* were properly de-identified. We can therefore use this as the starting point to try to account for why this de-identification was successful. We see that the setup of links ensures that *DataItem_4* will be transformed by the *DeIDed_0* access ticket, i.e., from *DataItem_4* we navigate:

1. the *DICat* link to the *HDate1* category that shows that *DataItem_4* is correctly categorised;
2. the *ARTransforms* link from *HDate1* to the *TransformsHDate1* access rule that shows that the correct transformation rule is linked;
3. the *ARAppliesTo* link from *TransformsHDate1* to the *Individual1* type that shows that individual *HDate* instances, i.e. *HDate1*, are designated to be transformed; and
4. the *PermRules* link from *TransformsHDate1* to the *DeIDed_0* access ticket to ensure that the project's access ticket applies the *TransformsHDate1* access rule.

Since the links we have seen are consistent with what we expect for de-identification, the user will (now) check if these corresponding links also exist for *DataItem_5* (as a way to possibly understand why data

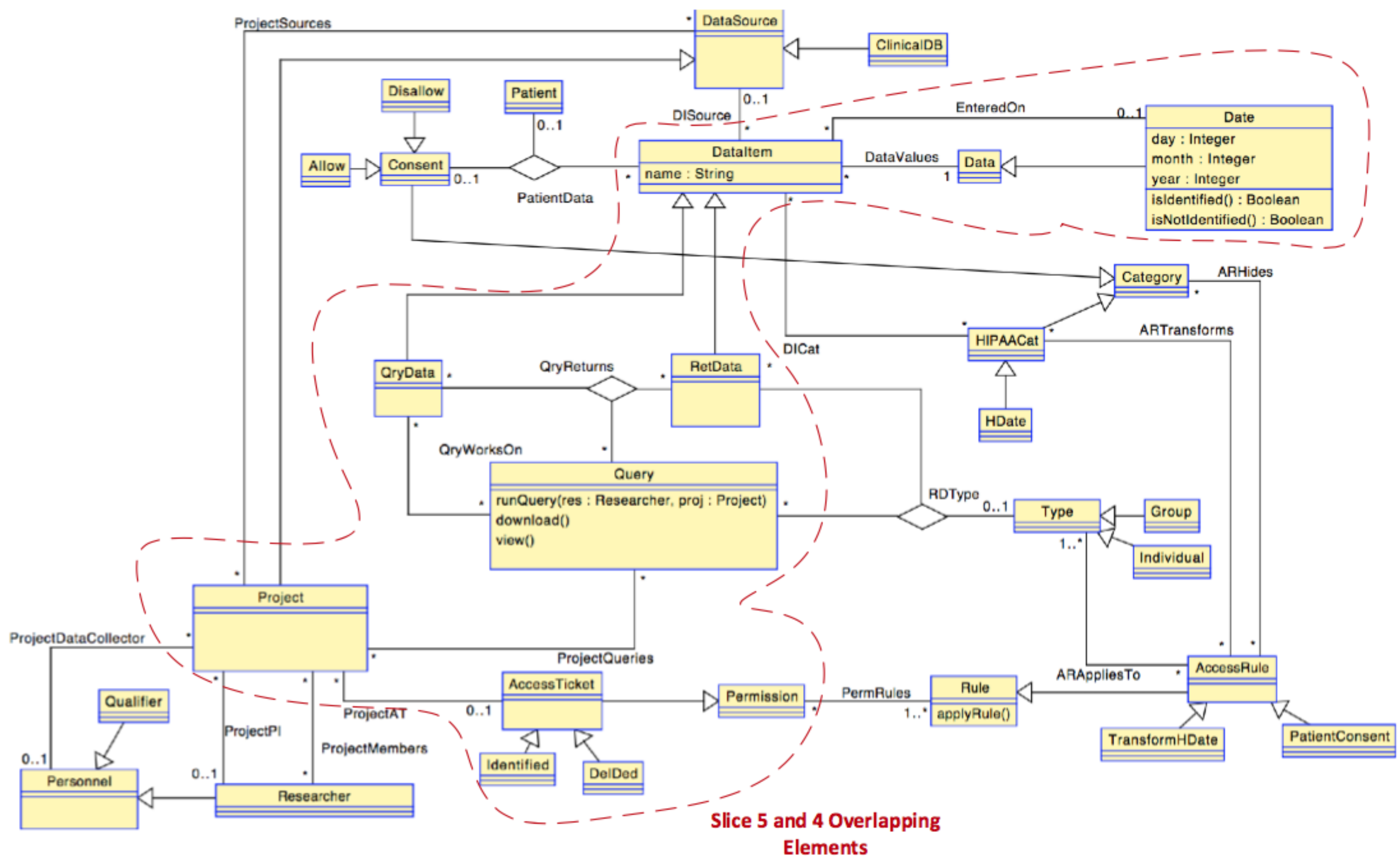


Figure 7.18: Class Model for Slice 4 Outlining Overlapping Model Elements in Slices 5 and 4

derived from it were not properly de-identified). Our object model shows that it has not be categorised as an *HDate* and observe that all the other data items whose data values are dates have been correctly categorised. This is definitely an explanation for the non-conformance. The missing link that shows the fault is drawn into Figure 7.20 using a green dashed line and labeled with the same colour font.

At this point we may request the system to show us the previous slice so we may investigate other reasons for the non-conformance. An object model for slice 3, where the *DeIDed_0* access ticket was approved for *Project_1*, is shown next. It is constructed in a similar way as was described for constructing the object model for slice 4. We show it in Figure 7.21 also highlighting in grey the overlapping object model elements with slice 4 (the extracted class model is shown in Figure 7.19).

We do not identify any problems with the objects and links in this object model that could cause the non-conformance shown in Figure 7.17. However, yet another step may be that the user requests to see an object model with all the slices merged. We show this in Figure 7.22. In it there is further confirmation that there is nothing in the overlaps of slice 3, 4, and 5 that could cause the non-conformance. Therefore, we return to the previous object model for slice 4 to devise our next steps. These steps include examining the OCL constraints to identify why *DataItem_5* was not also categorised as an *HDate*.

Our specification shows that no constraint enforces that *every DataItem* that is a *Date* to be categorised as an *HDate*, i.e., this system model leaves such categorisation to the discretion of the system administrator even though HIPAA mandates it. To ensure that we can always pass the conformance checks, we add a constraint to the OCL system model specification to ensure that all dates are categorised as *HDate*. The constraints providing the fix must be added to both the USE and the Alloy specifications. Re-executing the conformance check in the Alloy Specifications should now show no counterexamples. However, if we have a counterexample, the previous investigation we performed on the object models gives us assurance that the problem may be in the actual de-identification of the data and not in the system configuration represented by the class model and constraints.

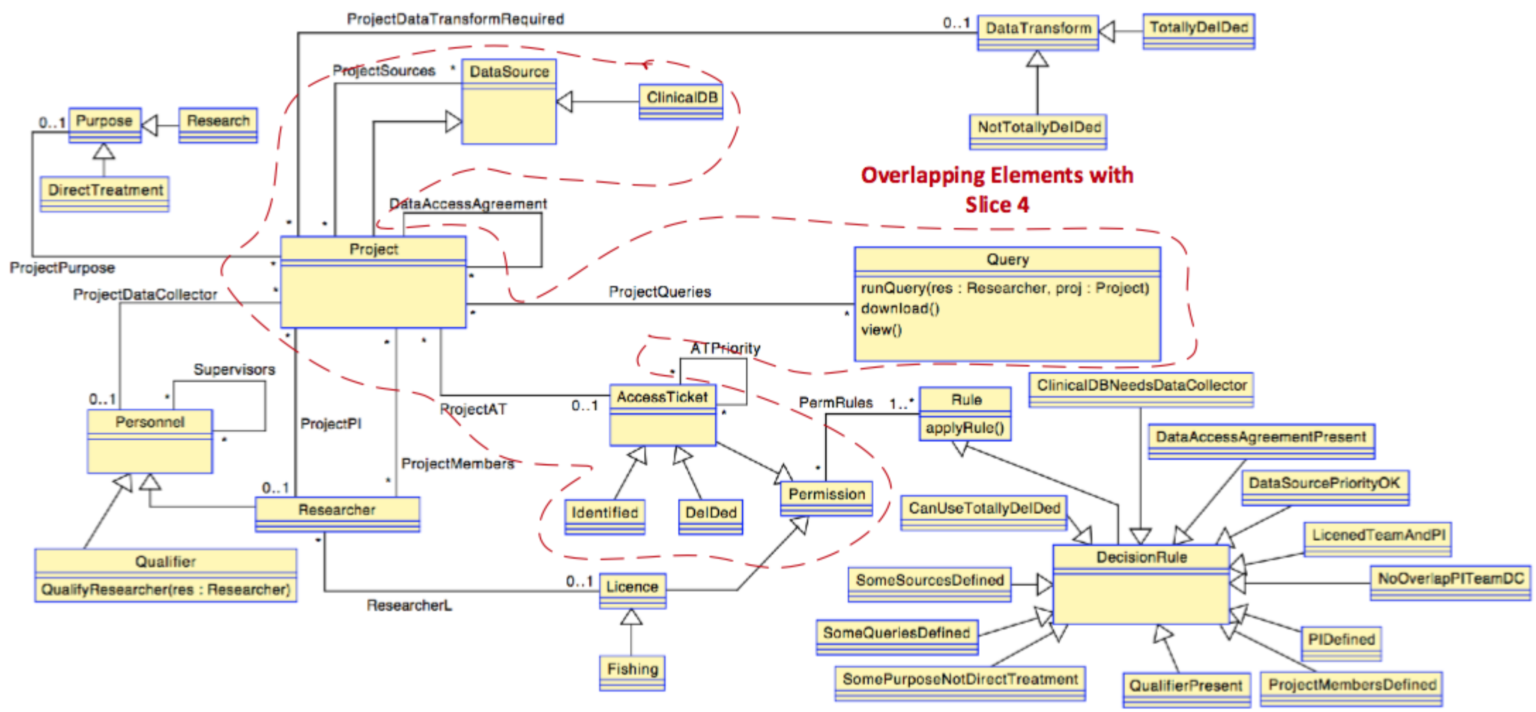


Figure 7.19: Class Model for Slice 3 Outlining Overlapping Model Elements in Slices 4 and 3

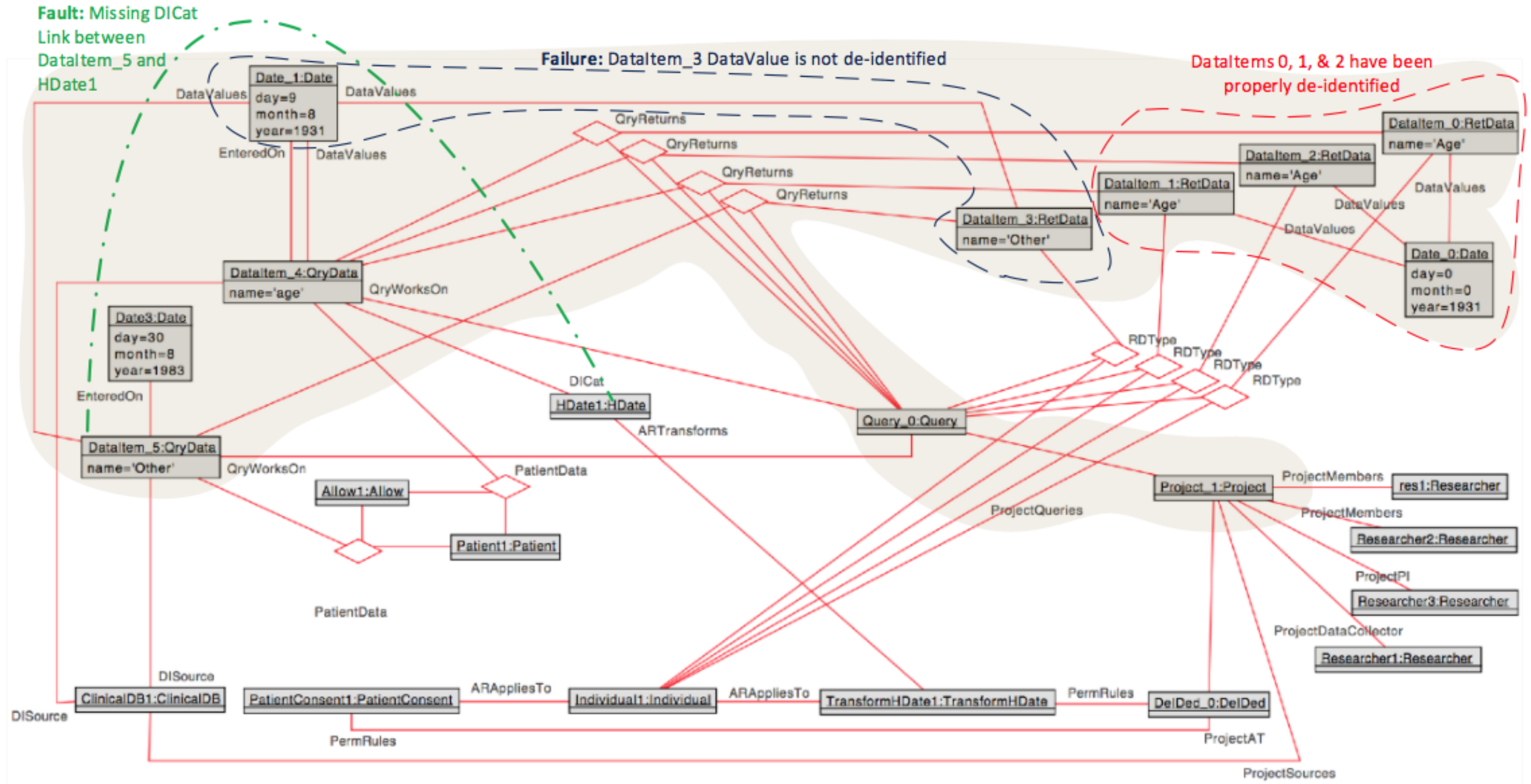


Figure 7.20: Non-Conformance Object Model for Slice 4 Identifying Failure and the Fault. (overlapping objects with Slice 5 are highlighted)

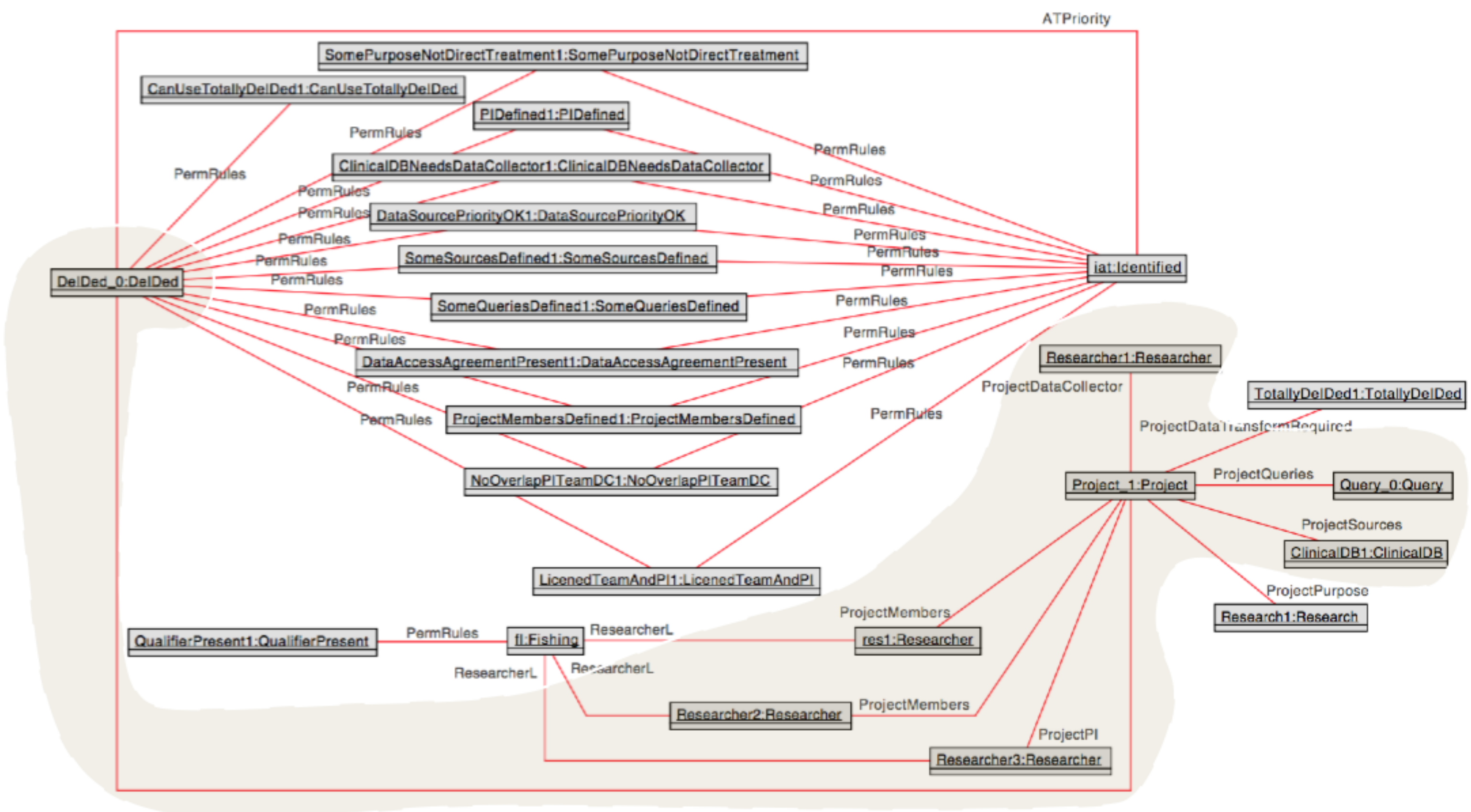


Figure 7.21: Object Model for Slice 3 (overlapping objects with Slice 4 are highlighted)

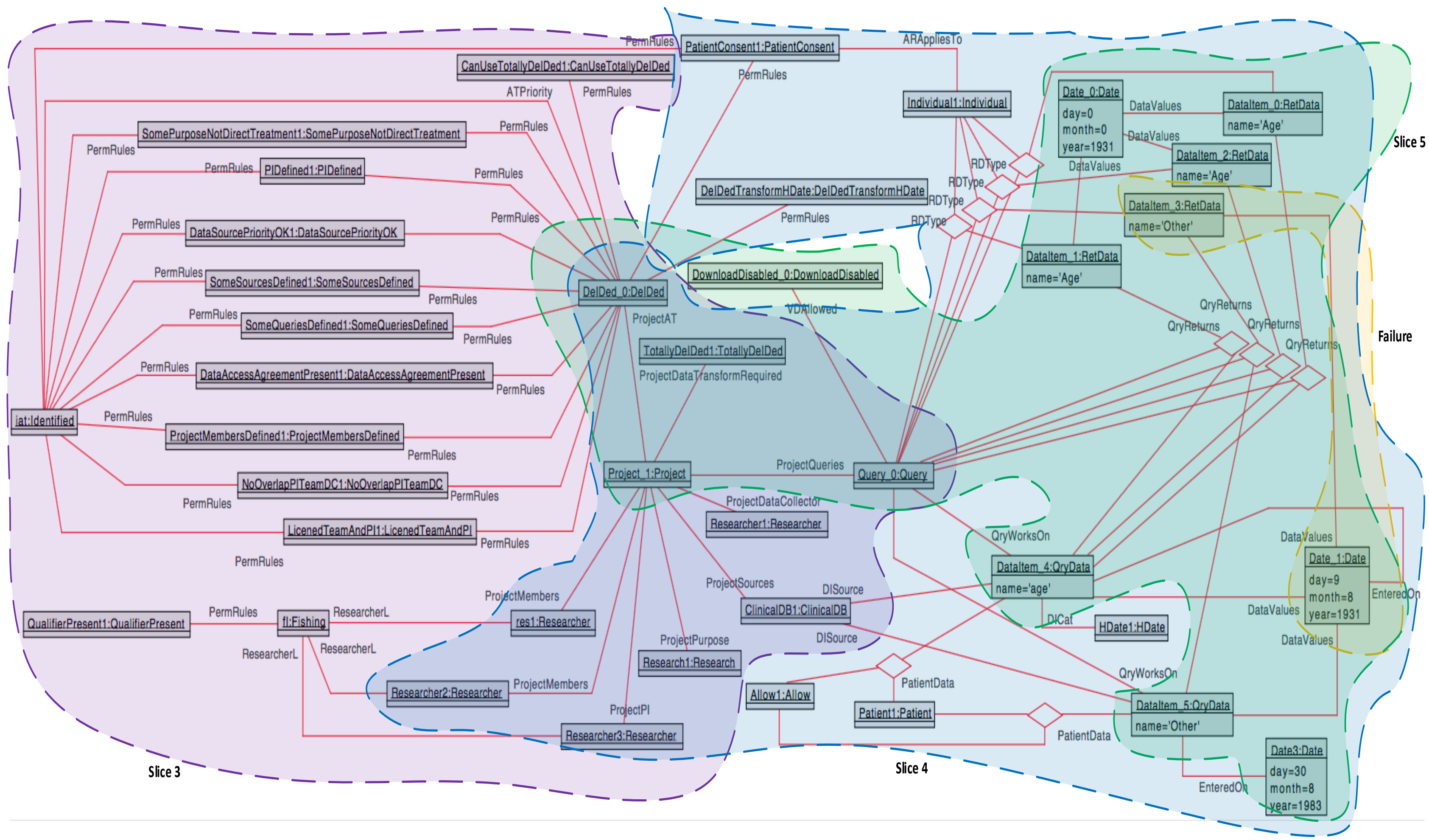


Figure 7.22: Merged Object Model for Slices 3, 4, and 5. Slice 3 is outlined by the purple dashed line, Slice 4 is outlined by the blue dashed line, Slice 5 is outlined by the green dashed line, and the Failure is outlined by the yellow dashed line.

7.7 Summary

Non-conformance represents the failure of the system in the verification of rules and the validation of user and external agency expectations. We have demonstrated that when non-conformance occurs, the object models can be useful to a domain expert as a starting point into their investigation of the error state that led to the failure. We have previously discussed in Chapter 1 how enforcing rules requires us to examine the details of our system. Thus, the modelling and analysis at the granularity of the class and object models on data fields is crucial.

While the object models are useful, in system like the NJH system it is not unrealistic for a query to examine 10 million fields and to return results from 10 thousand of them. Further still, we know from a human computer interface point of view, it is not feasible to show an object model with all 10 thousand fields! Therefore, future research may include examining the scale of such object models and identifying some semantics for what the feedback shown to the user should contain to make it usable, i.e., slicing the feedback. For example, while we examined date fields to demonstrate non-conformance on individual fields, there are other rule-parts regarding de-identification as discussed in Section 3.4.

One way to slice the feedback may be to first identify which parts of the rule were not satisfied leading to the non-conformance and then to show only those objects and links relating to those rule parts. We may further slice the object models by each (non-satisfying) rule part, and if the object model it still too large, return a sample of the fields exhibiting the non-conformance. This proposed slicing of the object model can be used to reduce the cognitive overload to the user and make the feedback more usable. In addition to slicing, any request for previous slices must also use the rule part of predecessor slice so that the object and links generated have the appropriate context and overlap. We will discuss additional verification and validation of HMCA in Chapter 8.

8. VALIDATING HMCA

8.1 Introduction

In general, HMCA is designed to encode and analyse rules to tell us when non-conformance occurs. One way to apply HMCA is to follow a step-wise process, i.e., for each rule 1) construct models of the system and the rule, 2) analyse rule, 3) examine the feedback where non-conformance occurs, and 4) fix the system. So far, we have used this step-wise process to analyse conformance of our example system, NJH, to the HIPAA de-identified access rule, i.e., when a *DeIDed* access ticket is used the results of a query are de-identified. In this chapter, we demonstrate:

1. additional validation through error seeding - first through a logic error in a rule and second through incomplete analysis of indirect relationships; and

2. that these seeded errors correspond to real-world problems - the logic error causes non-conformance to the previously verified HIPAA de-identified access rule and the second causes conflicts of interest.

For seeding the errors, we analyse two new scenarios not yet explained in our discourse. First we add querying using the *Identified* access ticket, and show that even though we have not changed our specifications for the *DeIDed* access ticket, non-conformance is detected. Second, we revisit conflicts of interest by adding new information on how data collectors may conflict with researchers and show that non-conformance is also detected due to underspecification in our system.

We discuss the identified access ticket to the HIPAA conformance rule in Section 8.2, the conflicts of interest as both a decision rule for all access tickets and as a NJH conformance rule in Section 8.3, and end this chapter with a summary in Section 8.4.

8.2 Adding a New Parts to HIPAA Conformance Rule: Exposing Faulty Logic

8.2.1 Updating Conformance Rule for the *Identified* Access Ticket

One of the decision rules used for granting a *DeIDed* access ticket is that the researchers indicate that only totally de-identified data can be used. In this case we say that the access rule implies that the data requires a *TotallyDeIDed* data transform. For an *Identified* access ticket, the researchers are required to indicate whether they:

1. must have all of their data identified, which requires a *TotallyIDed* data transform; or

2. can use de-identified data, which allows the data to be either identified or de-identified. Here we say that a *AllowDeIDed* data transform is required.

In the case of the *AllowDeIDed* the project's data source, e.g., a previous project, may already or only contain de-identified data, and rather than exclude it in the query result, the researchers are willing to use it.

With the inclusion of the *Identified* access ticket, showing conformance to the HIPAA regulations now has three required parts based on the access ticket type and the required data transformation such that:

1. $(DeIDed \wedge TotallyDeIDed) \rightarrow$ *no date returned is identified*[1];

2. $(IDed \wedge TotallyIDed) \rightarrow$ *no date returned is de-identified*; and

3. $(IDed \wedge AllowDeIDed) \rightarrow$ *any date returned is identified or de-identified.*

To show conformance for the *DeIDed* access ticket we did not require using *TotallyDeIDed* as a part of the rule, because a well formed model meant that only the *DeIDed* access ticket had this condition. Therefore it was sufficient to use

$DeIDed \rightarrow$ *no date returned is identified*

in the conformance rule. This meant that the *projectDataTransformRequired* association outlined by the red dashed line in Figure 8.1 was not required in slice 5 (See Figure 7.16) to show conformance for the *DeIDed* access ticket. (Note that the subtypes of *DataTransform* have been updated from the subtypes shown in figures 7.9 through 7.14 and Figure 7.19 where we replace *NotTotallyDeIDed* with *AllowDeIDed* and add *TotallyIDed* to have the meanings as discussed above.) However, because the *Identified* access ticket has two alternatives for the data transform, showing conformance requires that we now include the *projectDataTransformRequired* association in slice 5. We show an updated slice 5 in Figure 8.2 to include the *projectDataTransformRequired* association outlined by the red dashed line.

[1] Recall that an identified date means that in addition to a value for the year, the date has a value for the day or month and de-identified means that it only has a value for the year.

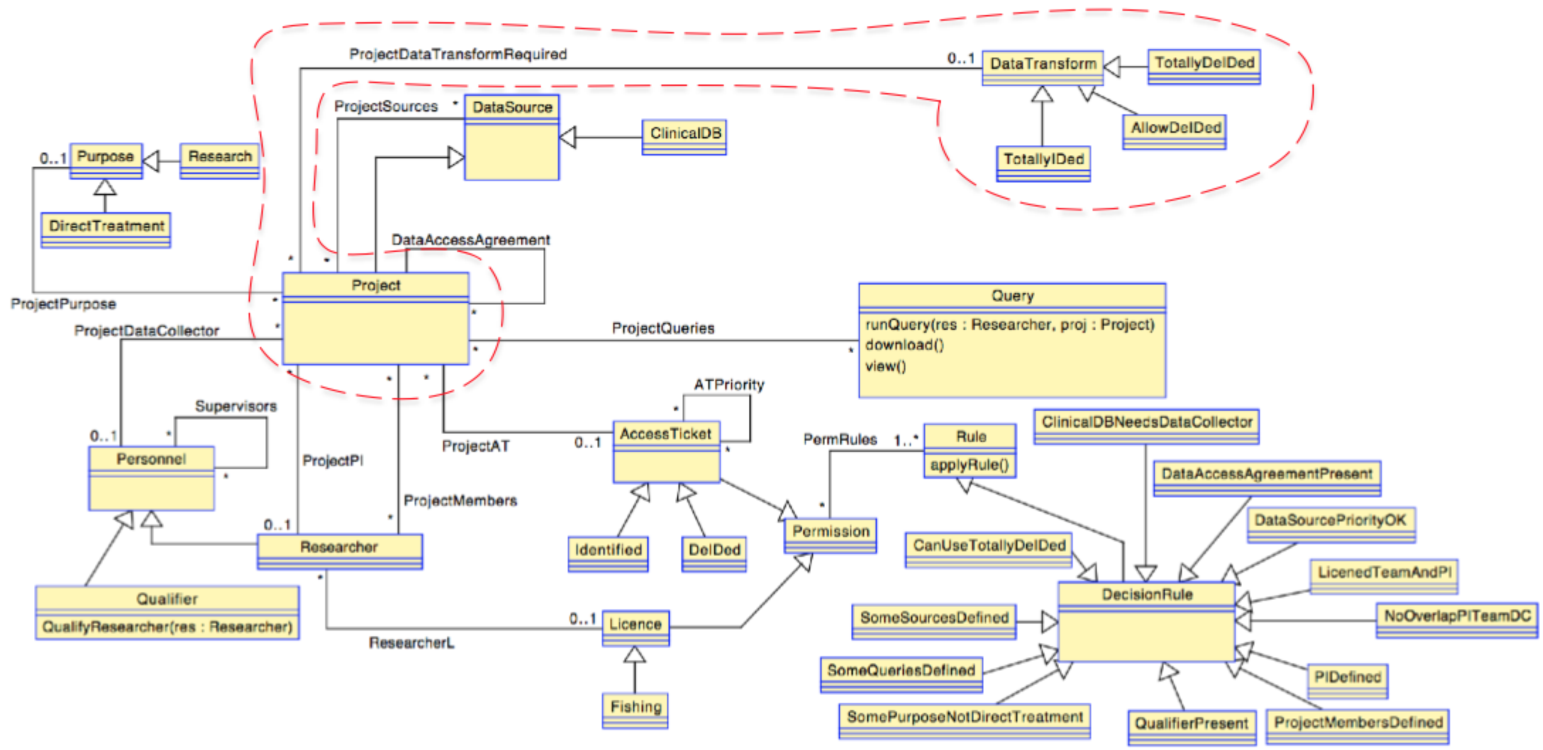


Figure 8.1: Updated Class Model for Slice 3 Outlining *ProjectDataTransformRequired* Association Now Required in Slice 5

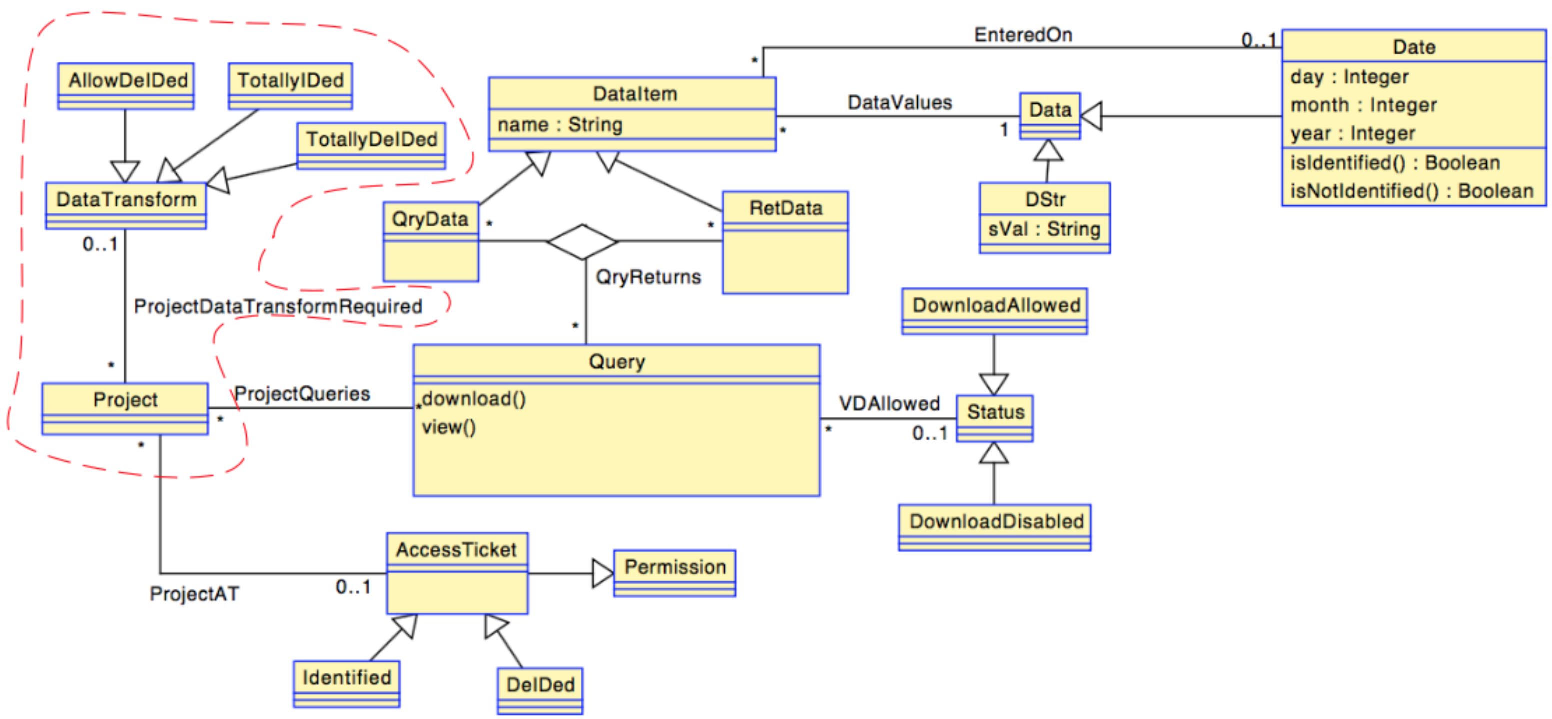


Figure 8.2: Updated Class Model for Slice 5 with the Now Required *ProjectDataTransformRequired* Association Required to Check Conformance

Listing 8.1: HIPAA Conformance Specifications: *VDAllowed* is set

```
all
  njh: NJH, q: njh.queries |
let
  p = njh.projectQueries.q,
  pdtr = p.(njh.projectDataTransformRequired),
  a = some pdtr & TotallyIDed implies totallyIDedTransform[njh, q],
  b = some pdtr & TotallyIDed implies not totallyIDedTransform[njh, q],
  c = some pdtr & AllowDeIDed iff allowDeIDedTransform[njh, q],
  d = some pdtr & TotallyDeIDed implies totallyDeIDedTransform[njh, q] ,
  e = some pdtr & TotallyDeIDed implies not totallyDeIDedTransform[njh, q] | {

/** Query results are downloadable */
some q->DownloadAllowed & njh.VDAllowed implies
  ((a and not b) or (d and not e) or c)

/** Query results are not downloadable */
some q->DownloadDisabled & njh.VDAllowed implies
  ((not a and b) or (not d and e)
  ) }
```

8.2.2 Alloy Specifications

Suppose[2] we use the Alloy predicate in Listing 8.1 to update the conformance status of a query, i.e., the query status in *VDAllowed*[3]. We ensure that the query status is correctly set to *DownloadAllowed* using

```
some q->DownloadAllowed & njh.VDAllowed implies
  ((a and not b)or (d and not e)or c)
```

to mean that a query has a *DownloadAllowed* status in *VDAllowed* if it is true that:

1. its associated project requires a *TotallyIDed* data transform and all the dates returned are identified, i.e., `a and not b`; or

2. its associated project requires a *TotallyDeIDed* data transform and all the dates returned are de-identified, i.e., `d and not e`; or

3. its associated project requires a *AllowDeIDed* data transform and the dates returned are either identified or de-identified, i.e., `c`.

[2]By “suppose” we mean a fault is seeded here.
[3]The ”VD” in *VDAllowed* is for *Viewing* or *Download* of query results.

Listing 8.2: Helper Predicates used to Check Conformance

```
private fun applicableDates(njh: NJH, q: Query): set Date {
   { Date &
      dom[q.(njh.qryReturns)].(njh.dataValues) +
         dom[q.(njh.qryReturns)].(njh.enteredOn) }}

private pred totallyIDedTransform (njh: NJH, q: Query) {
   all d: applicableDates[njh, q] | identifiedDate[d]}

private pred totallyDeIDedTransform (njh: NJH, q: Query) {
   all d: applicableDates[njh, q] | not identifiedDate[d]}

private pred allowDeIDedTransform (njh: NJH, q:Query) {
   all d: applicableDates[njh, q] | identifiedDate[d] or not identifiedDate[d]}
```

We also ensure that the query status is correctly set to *DownloadDisabled* using:

```
some q->DownloadDisabled & njh.VDAllowed implies

   ((not a and b)or (not d and e))
```

that sets up an *XOR* situation for a query status. This formulation means that a query has a *DownloadDisabled* status in *VDAllowed* if it is true that:

1. its associated project requires a *TotallyIDed* data transform and some date is returned that is de-identified, i.e., `not a and b`; or

2. its associated project requires a *TotallyDeIDed* data transform and some date is returned that is identified, i.e., `not d and e`.

Listing 8.1 makes reference to other predicates, i.e., `totallyIDedTransform[njh, q]`, `allowDeIDedTransform[njh, q]` and `totallyDeIDedTransform[njh, q]`, and we include them in Listing 8.2.

In order to check that we have not over constrained the model we (use predicates to) generate instances of the model for all 5 conditions, i.e., *a* to *e*, in Listing 8.1 where we ensure that the query has the expected status in *VDAllowed*. For example when both clauses of *a* are true the query has a *DownloadAllowed* status and when both clauses of *b* are true the query has a *DownloadDisabled* status. We generate instances and this gives us assurance that we have done it right.

The next step is to check conformance. For example, to ensure that a query that should not have a *DownloadAllowed* status, indeed cannot, we use the Alloy snippet below:

```
some p.(njh.projectDataTransformRequired)& TotallyIDed and
   some q->DownloadAllowed & njh.VDAllowed and (
      some p.(njh.projectDataTransformRequired)& TotallyIDed implies
            all r: applicableDates[njh, q] | identifiedDate[r])
```

in an assertion to check that a query q whose associated project p requires a *TotallyIDed* data transform does not have de-identified dates in its result. HMCA detects non-conformance because the assertion finds a counterexample.

8.2.3 Examining Feedback Object Models

We request feedback and we are shown the object model in Figure 8.3 where we see that *DataItem_0*, *DataItem_1* and *DataItem_2* show a conformance failure for the *Identified* access ticket requiring a *TotallyIDed* data transform because their associated dates are de-identified. When a similar assertion is executed for the *DeIDed* access ticket, it also returns a counterexample. The feedback from this is shown in Figure 8.4 where we see that *DataItem_3*, shows a conformance failure for the *DeIDed* access ticket requiring a *TotallyDeIDed* data transform because its associated date is identified.

While not necessarily a part of feedback because there is no conformance failure, we include Figure 8.5, when there is an *Identified* access ticket and the data transform required is *AllowDeIDed*. We note that the figures 8.3, 8.4 and 8.5 use the same set of *DataItems* yet it is the access ticket and the data transforms that tells us whether conformance rules have been violated or not.

Since the *DeIDed* access ticket also shows non-conformance and we know that in Section 7.6 we verified that the status in *VDAllowed* was being set correctly for the *DeIDed* access ticket, it must be that there is a fault in the way we set the status for each query in Listing 8.1.

8.2.4 Understanding Why Non-Conformance Occurs

Inspection of the predicate reveals that the statement

```
c = some pdtr & AllowDeIDed implies allowDeIDedTransform[njh, q]
```

in Listing 8.1 is causing the conformance failures. The fault is now obvious, i.e., the use of *implies* in the statement is the faulty connector.

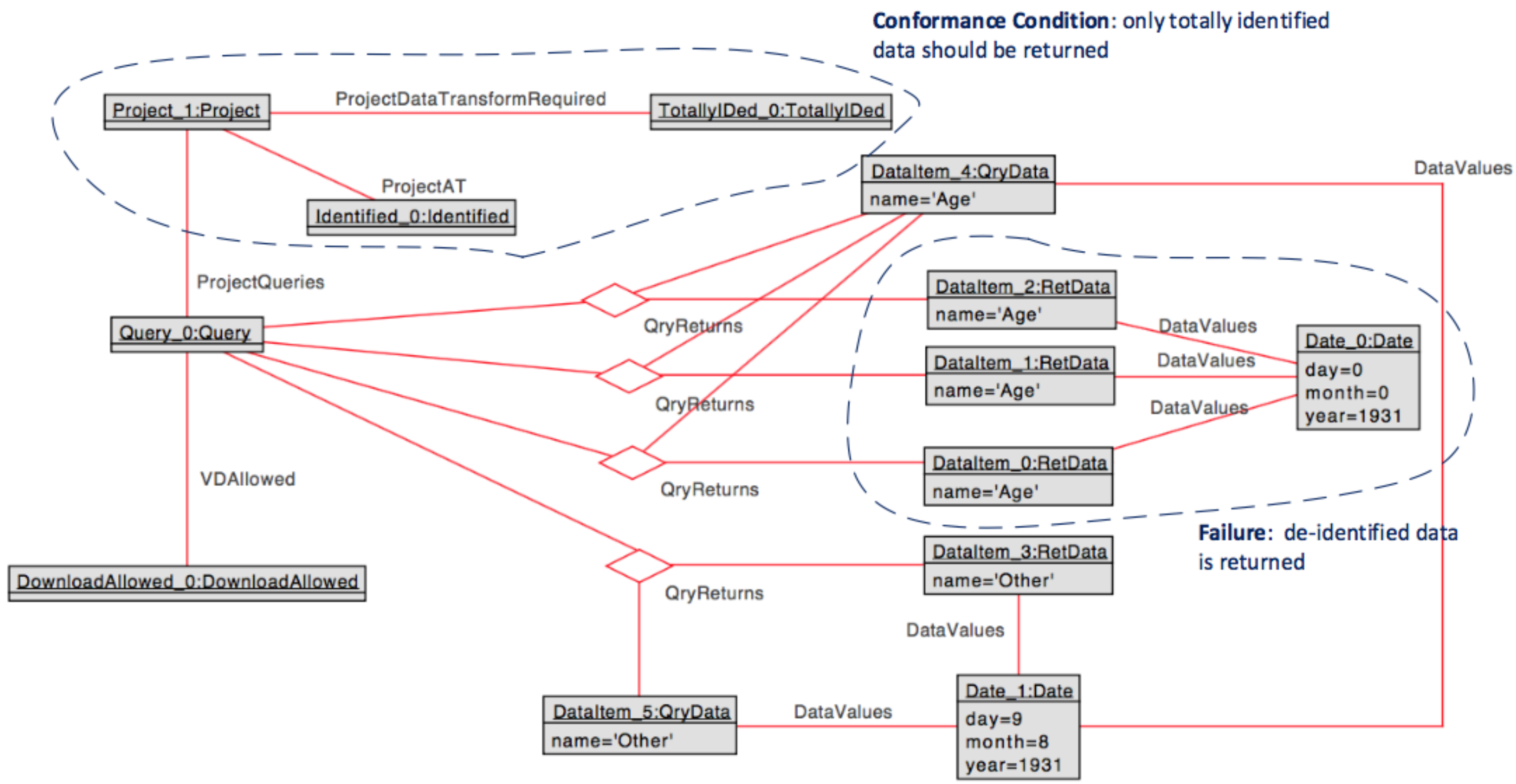


Figure 8.3: Non-Conformance in Slice 5 when an *Identified* Access Ticket is used and a *TotallyIDed* Data Transform is Required

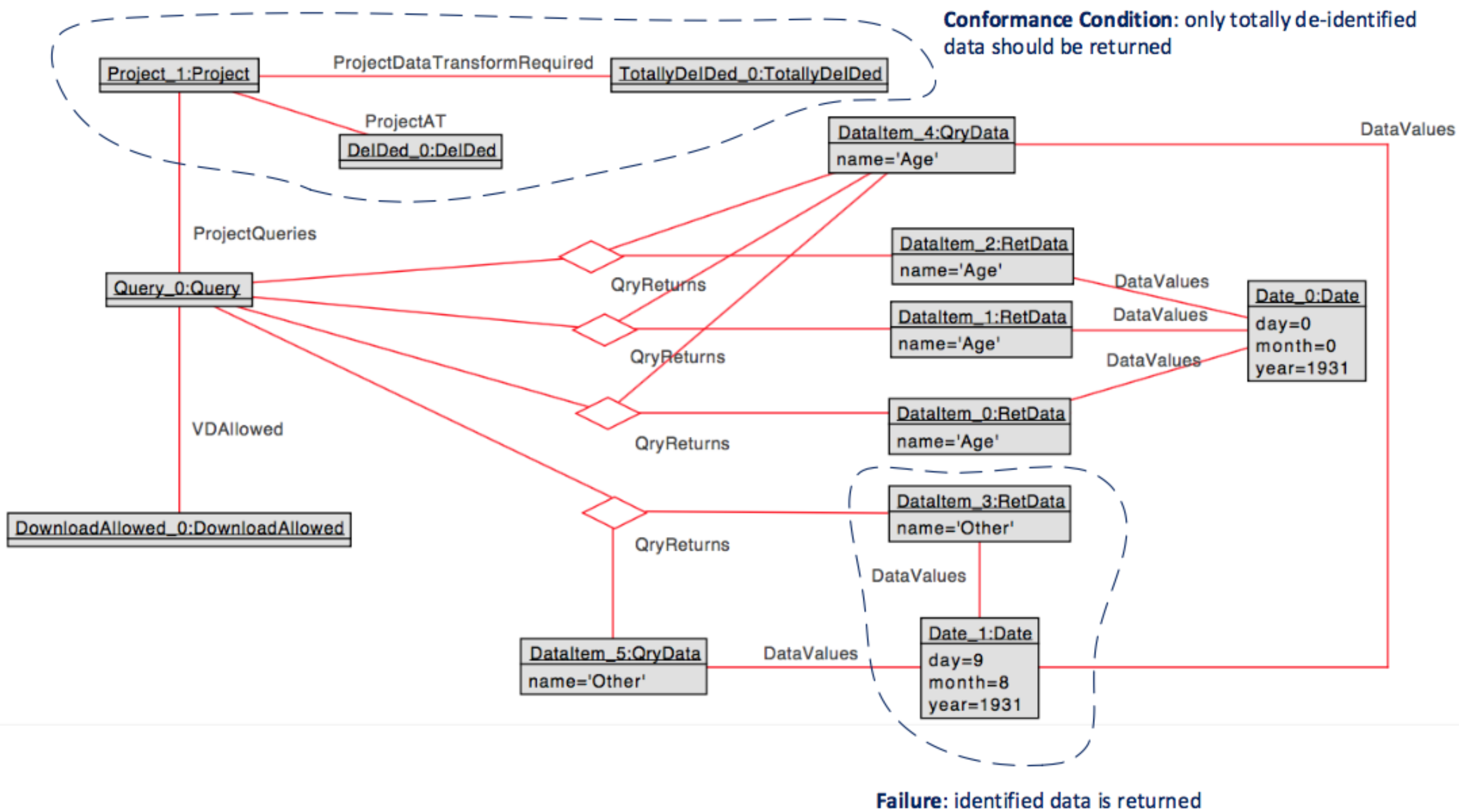


Figure 8.4: Conformance in Slice 5 when an *DeIDed* Access Ticket is used and a *TotallyIDeDed* Data Transform is Required

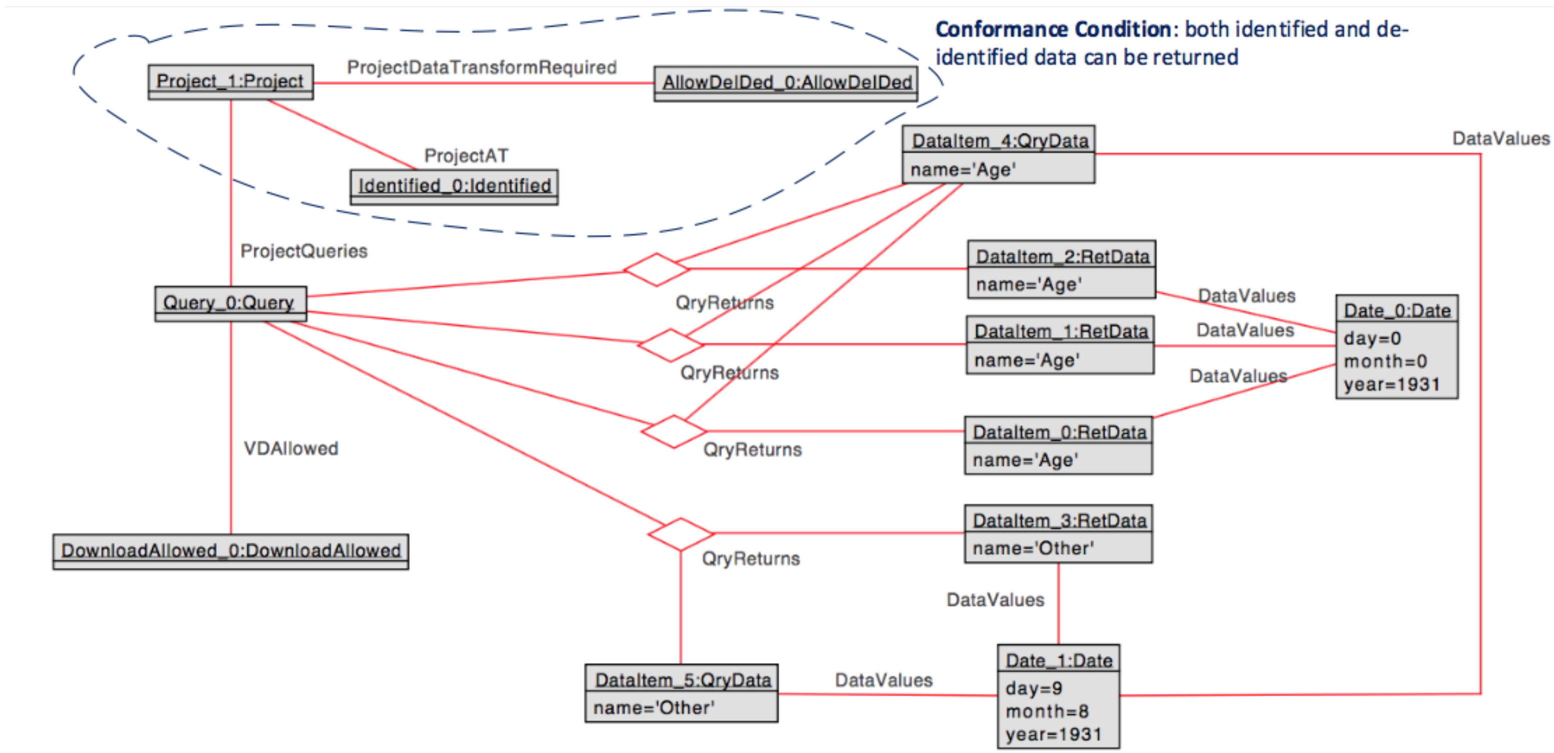


Figure 8.5: Conformance in Slice 5 when an *Identified* Access Ticket is used and an *AllowIDed* Data Transform is Required

The use of *implies* is appropriate for both *DeIDed* with a *TotallyDeIDed* data transform and *Identified* with a *TotallyIDed* data transform, i.e.,

```
a = some pdtr & TotallyIDed implies totallyIDedTransform[njh, q]
```

and

```
d = some pdtr & TotallyDeIDed implies totallyDeIDedTransform[njh, q]
```

respectively, because these were not the only access tickets that allowed de-identified or identified dates.

We note that we could also use *iff* as the connector for the clauses in a and d, i.e., using

```
a = some pdtr & TotallyIDed iff totallyIDedTransform[njh, q]
```

and

```
d = some pdtr & TotallyDeIDed iff totallyDeIDedTransform[njh, q]
```

yet this neither cause changes in the instances we expected for the *TotallyIDed* and the *TotallyDeIDed* data transformations nor HMCA finding non-conformance when their associated access tickets are used.

However, further analysis shows that it is indeed correct to use *implies* because using *iff* excludes the *AllowDeIDed* transform from having dates that only contain all identified dates or all de-identified dates. Therefore the *AllowDeIDed* transform would only contain a mixture of identified and de-identified dates to get a *DownloadAllowed* status because the *iff* mandates that only the *TotallyIDed* data transform to contain identified dates and the *TotallyDeIDed* data transform to contain de-identified dates.

In the case of *Identified* with an *AllowDeIDed* data transform, this was the only access ticket that allowed both de-identified or identified dates to co-exist in the data it returns and still show conformance. Also, using *implies* as the connector means that we have no specification about (the converse of) what status a query should have if it has both identified and de-identified dates.

Therefore, for the clauses in c, *iff* is the required connector. We show the correct formulation below:

```
c = some pdtr & AllowDeIDed iff allowDeIDedTransform[njh, q]
```

This correction still allows us to generate instances for a to e in Listing 8.1 and yet produce no counterexamples for the conformance checks. The complete Alloy specification, including the correction of the fault, is in Appendix D.1.2.

8.3 Adding a New NJH Conformance Rule: Identifying Conflict of Interest Situations

For our discussion in this section, we will make reference to these specific instances of the classes from Figure 8.6:

1. *DC*, the person collecting the data from a *ClinicalDB* to be returned in a project query, is the *Personnel* we reach by navigating the *ProjectDataCollector* association from the *Project* class;

2. *PI*, the principal investigator for a project, is the *Researcher* we reach by navigating the *ProjectPI* association from the *Project* class;

3. *PMs*, the researchers for a project, are the *Researcher*s we reach by navigating the *ProjectMembers* association from the *Project* class;

4. *Sup*, the supervisor of another person, is the *Personnel* we reach by navigating the *Supervisors* association from the *Personnel* class; and

5. *Sources* are the *DataSource*s we reach by navigating the *ProjectSources* association from the *Project* class. *Sources* can be the *ClinicalDB* (the NJH's DB) or other projects. In the case of the latter, we assume that the project has made queries of its own and augmented the NJH with additional data, so both the original data and the additional data are considered as the "*sources*".

When a project requires data from a *ClinicalDB*, a *DC* must be assigned to the project to extract the data from the database on behalf of the project. Since the *DC*, *PI* and *PM*'s for a project are all drawn from the same pool of *Personnel* and to prevent conflict of interest situations, there are some basic conditions that must be true to get a project's application for an access ticket approved. For a project (with respect to *Personnel*) there should be:

1. no overlap in PI and PM; and

2. no overlap in DC and $(PI + PM)$.

These conditions have already been incorporated into slice 3 (see Figure 7.19) as the *NoOverlapPITeamDC DecisionRule* for approving access tickets and to ensure that there are no violations. However, an examination

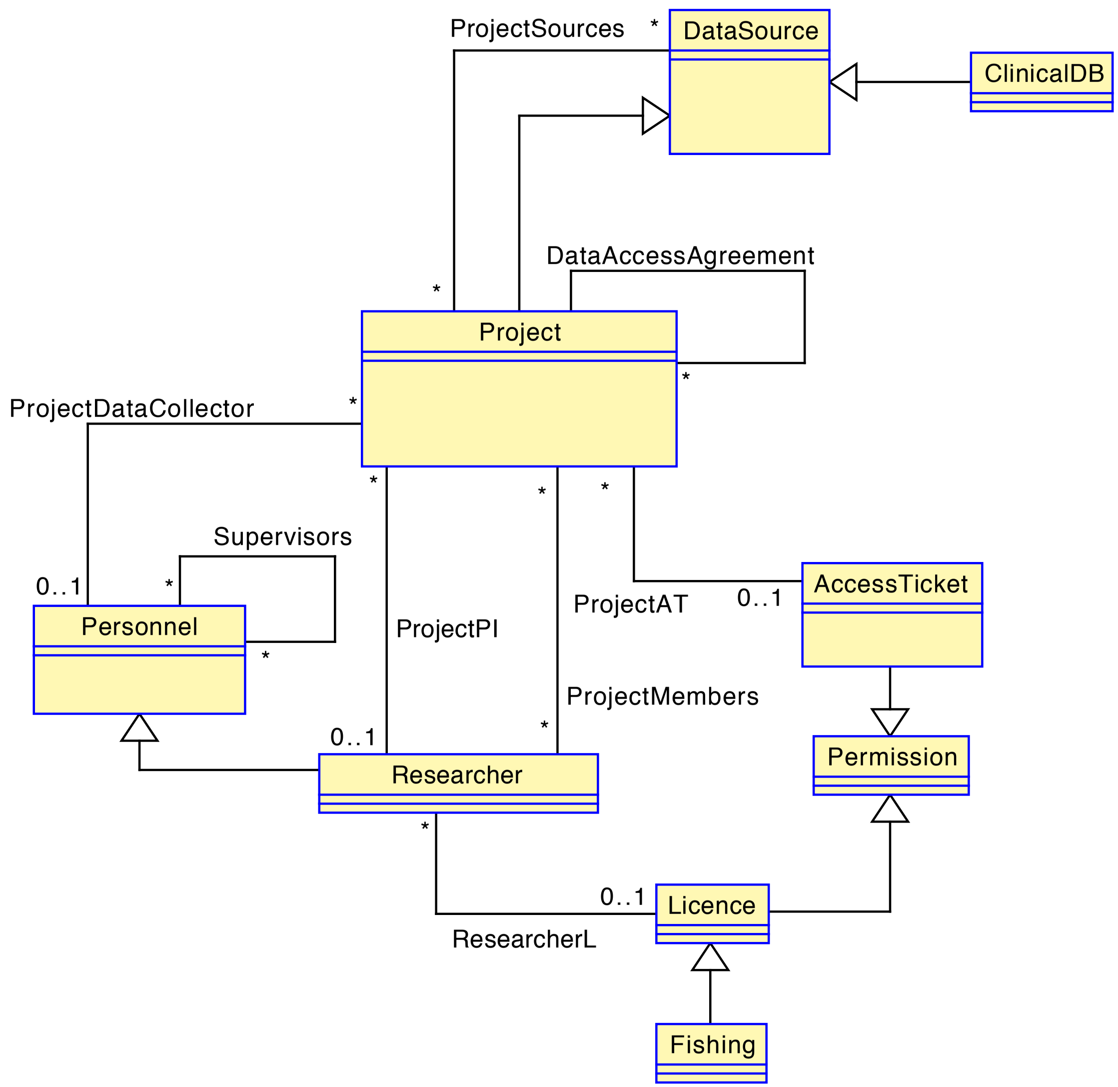


Figure 8.6: Partial Class Diagram Slice extracted from Slice 3 Showing Personnel Relationships influencing Access Ticket Approval

of the instances where an access ticket has been approved shows some other kinds of conflict of interest situations with respect to the *DC*, *PI*, and *PM*s for a project. We use HMCA to detect these situations by including a conformance rule that they should not exist.

Instead of using object diagrams to show instances of the situations, we will use the instances given by the Alloy Analyzer because they show the direction of the relationships where the former does not. For example, both the *DataAccessAgreement* and the *ProjectSources* in Figure 8.6 involve self relationships on the *Project* class. In an object diagram, unless we show the role names at each association end, we cannot know how to understand the links between objects. Showing the role names in addition to association names on the object diagrams causes too much clutter for the size of the object diagrams required. Instead, the Alloy Analyzer provides a better visualisation by showing the domain and range of a relationship[4] (association) by using directed edges. We will see these instances in figures 8.7 to 8.15. Note that the:

1. Alloy Analyzer instances shown will be partial instances of Slice 3 as depicted in Figure 8.6 where we remove the elements not applicable to checking and showing the conflicts of interest; and

2. black dashed lines with labeled annotations in these instances were not generated by the Alloy Analyzer, but were added manually to aid the reader in finding the example being described, e.g., the line labeled "*Project*" on the upper right of Figure 8.7

In the following subsections we discuss 4 conflict of interest situations. The first involve supervisory relationships and the others arise because a project can use another project as one of its *Sources*.

8.3.1 *DC* Conflict of Interest Case 1

The first situation is where the project's *PI* is the *Sup* for the project's *DC*. Using the Alloy partial instance in Figure 8.7 as an example, we see that *Project2* has an approved access ticket, shown by the *projectAT: Identified* label inside the project's ellipse, yet the *PI*, *Personnel5*, supervises its *DC*, *Personnel0*. We note that this supervisory relationship does not have to be a direct one, i.e., the supervisory relationship between the *PI* and the *DC* may be deeply nested.

[4]Since we already use *Sources* to describe the data source for a project, we wish to avoid confusion by saying *source and destination of a relationship (association)*, so we use the language of relations/functions, i.e., substitute *domain* for *source* and *range* for *destination*.

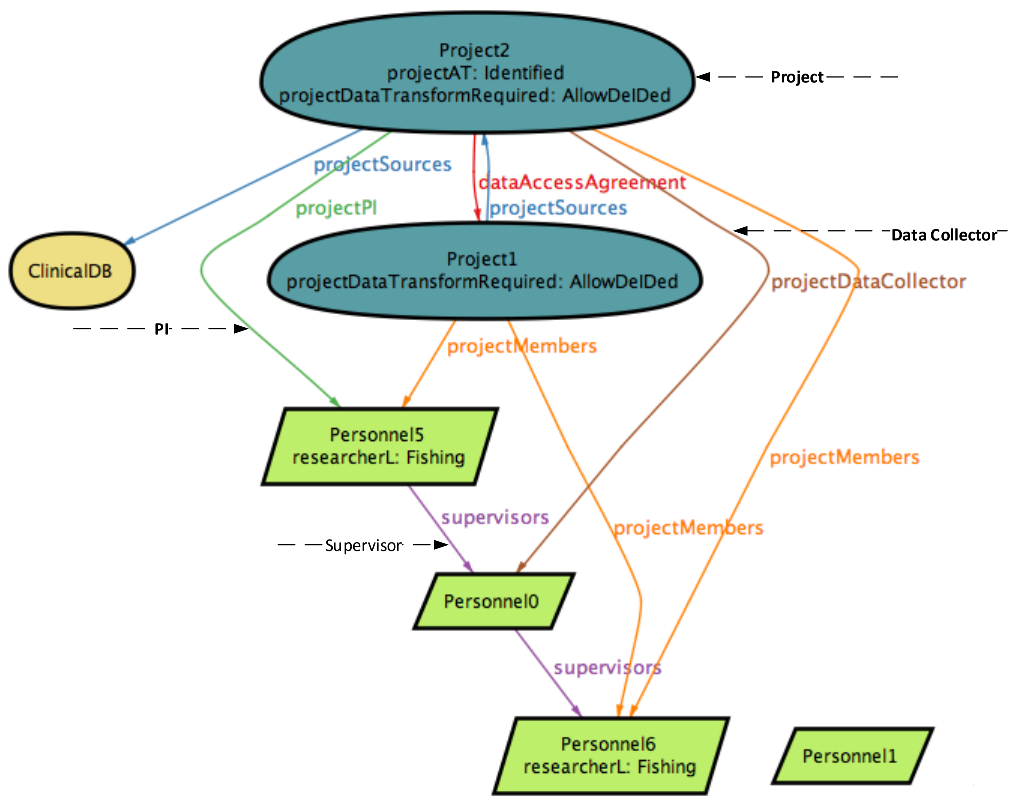


Figure 8.7: *DC* Conflict of Interest Project's *PI* supervises Project's *DC* : *Project2*'s *PI* *Personnel5* directly supervises its *DC* *Personnel0*.

In general, detecting these deeply nested relationships requires the use of closure operations. An example of the indirect supervisory relationship is shown in Figure 8.8, where *Project1*'s *PI Personnel1* supervises its *DC Personnel0*.

8.3.2 *DC* Conflict of Interest Case 2

The second situation is where the project's *PI* is the *Sup* for the *DC*s on any of the project *Sources*. Using the Alloy Analyzer partial instance in Figure 8.9 as an example, we see that *Project2* has an approved access ticket, yet its *PI*, *Personnel1*, supervises the *DC*, *Personnel3* for *Project0*, a *Source* for *Project2*. The conflict of interest still exists if the supervisory relationship is indirect or if the *Source* is indirect. We show examples of these indirect cases with the Alloy Analyzer partial instances in Figure 8.10 and Figure 8.11. In the former figure we see that *Project3* has an approved access ticket, yet its *PI*, *Personnel3*, supervises *Personnel0*, the *DC* for *Project1*, an indirect *Source*, through *Project0*, for *Project3*. In the latter figure we see that *Project3* has an approved access ticket, yet its *PI*, *Personnel1*, indirectly supervises *Personnel2*, the *DC* for *Project1*, an indirect *Source*, also through *Project0*, for *Project3*.

8.3.3 *DC* Conflict of Interest Case 3

The third conflict of interest situation arises when a project's *PI* is the *DC* for any of the project *Sources*. Using the Alloy partial instance in Figure 8.12 as an example, we see that *Project2* has an approved access ticket, yet its *PI Personnel2* is the same as the *DC* for *Project0*, a *Source* for *Project2*. The conflict of interest still exists if the supervisory relationship is indirect or if the *Source* was indirect. We show an example of this with the Alloy Analyzer partial instance in Figure 8.13 where we see that *Project3* has an approved access ticket, yet its *PI*, *Personnel1* is the same as the *DC* for *Project1*, an indirect *Source* for *Project3*.

8.3.4 *DC* Conflict of Interest Case 4

The final conflict of interest situation arises because the project's *PM*s overlap with the *DC* for one of the project's *Sources*. Using the Alloy partial instance in Figure 8.14 as an example, we see that *Project2* has an approved access ticket, yet one of it its *PM*'s *Personnel2* is the same as the *DC* for *Project0*, a *Source*

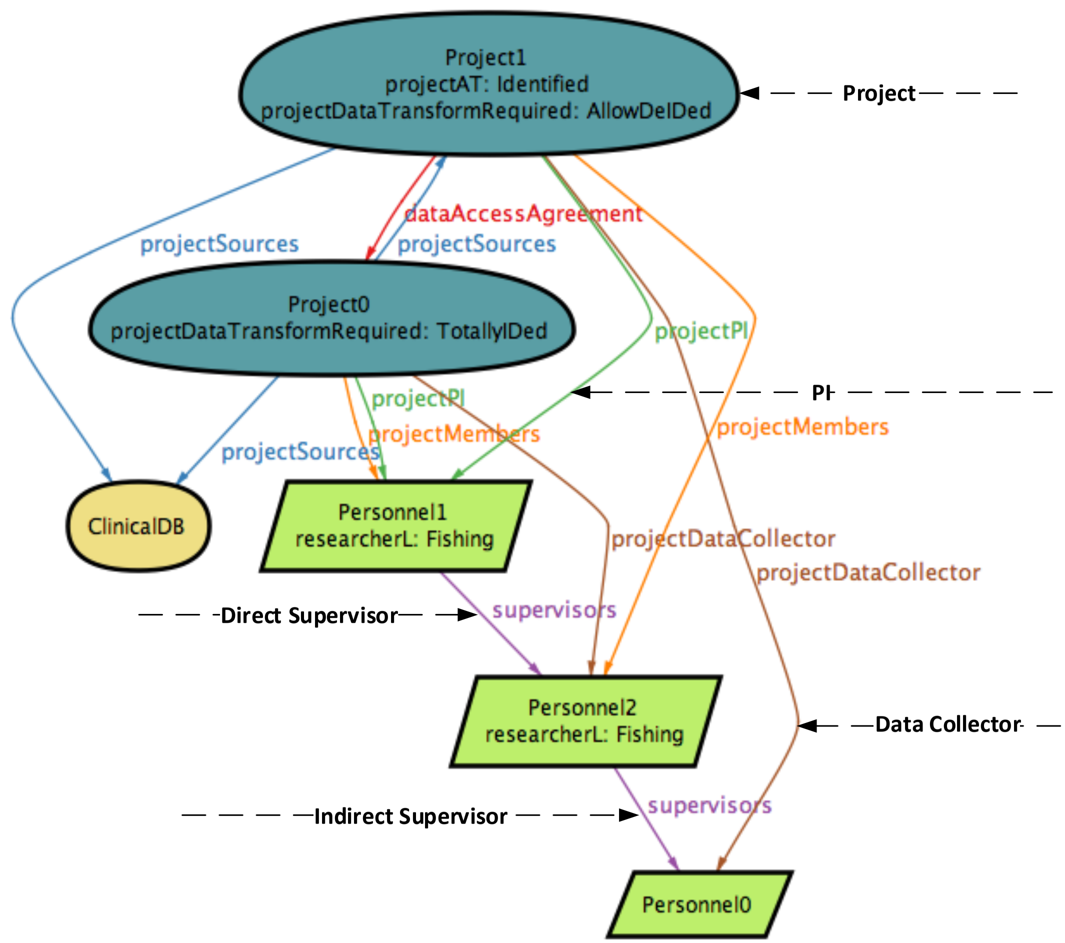


Figure 8.8: *DC* Conflict of Interest Project's *PI* indirectly supervises Project's *DC* : *Project1*'s *PI Personnel1* indirectly supervises its *DC Personnel0*.

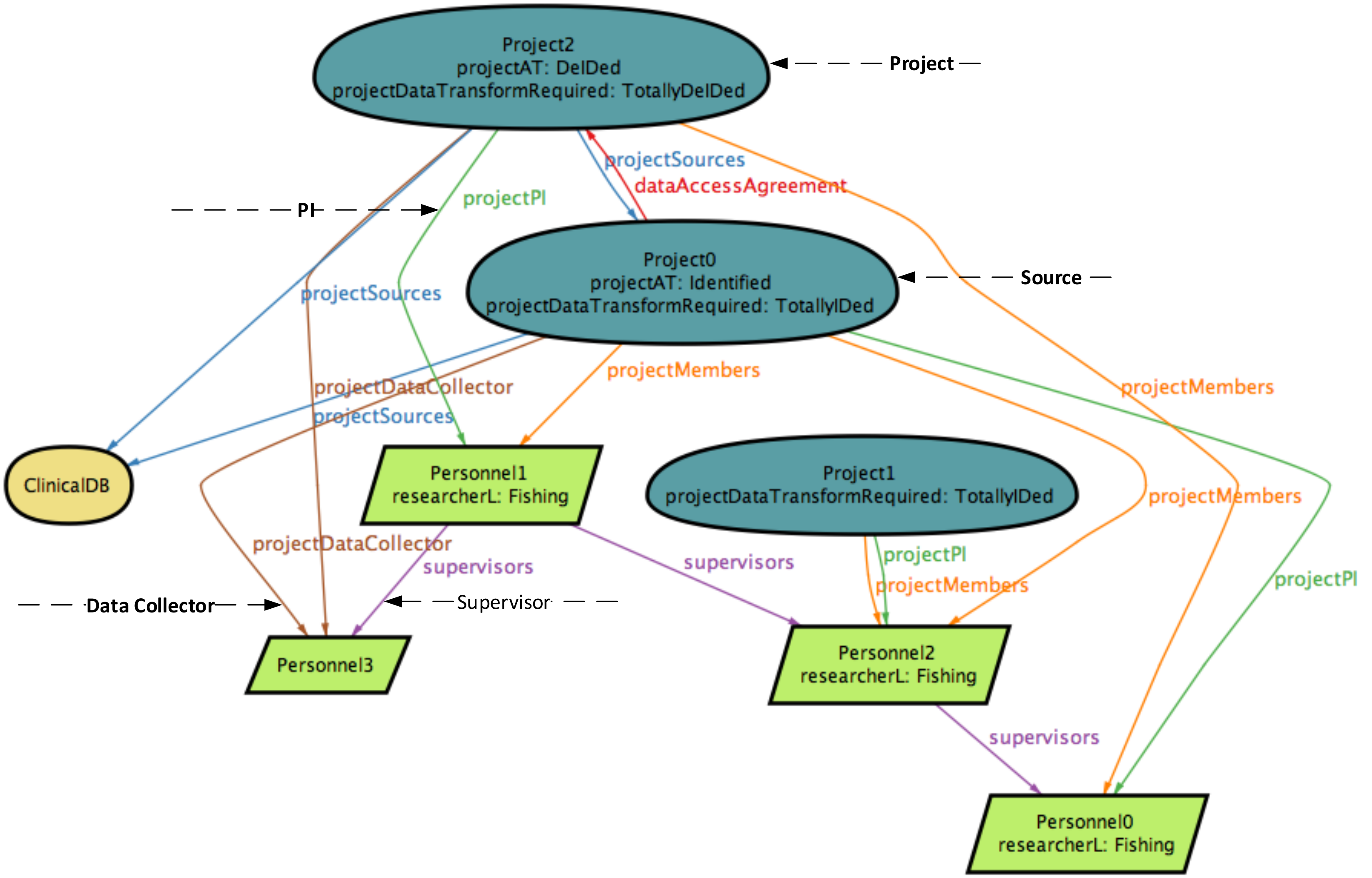


Figure 8.9: *DC* Conflict of Interest, Supervision of Project's Direct Source's *DC* by Project's *PI*: *Project2* has *Source Project0*, and *Project2*'s *PI*, *Personnel1*, supervises *Project0*'s *DC*, *Personnel3*.

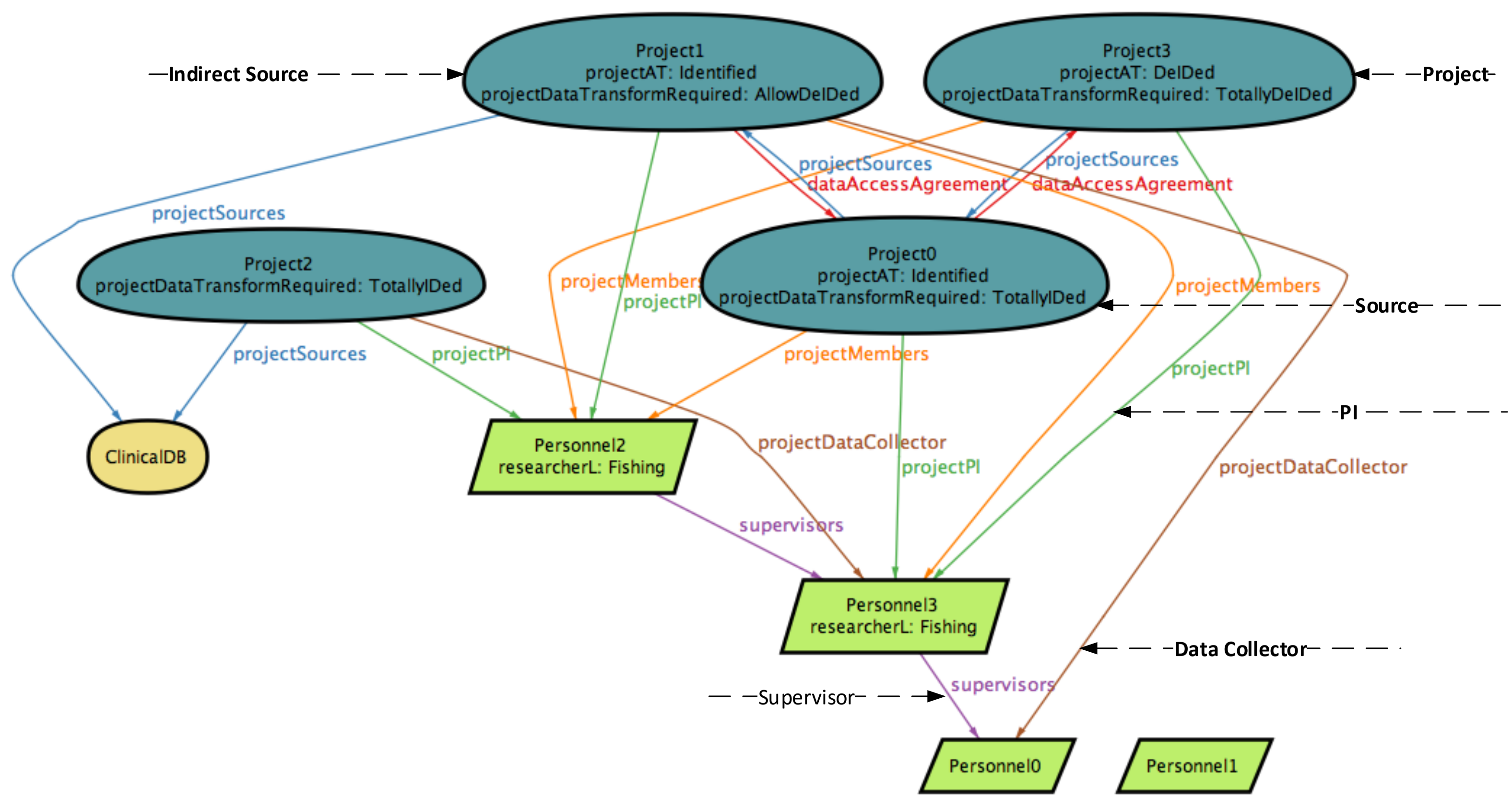


Figure 8.10: *DC* Conflict of Interest, Supervision of Project's Indirect Source's *DC* by Project's *PI*: *Project3* has indirect *Source Project1* and *Project3*'s *PI Personnel3* directly supervises *Project1*'s *DC Personnel0*.

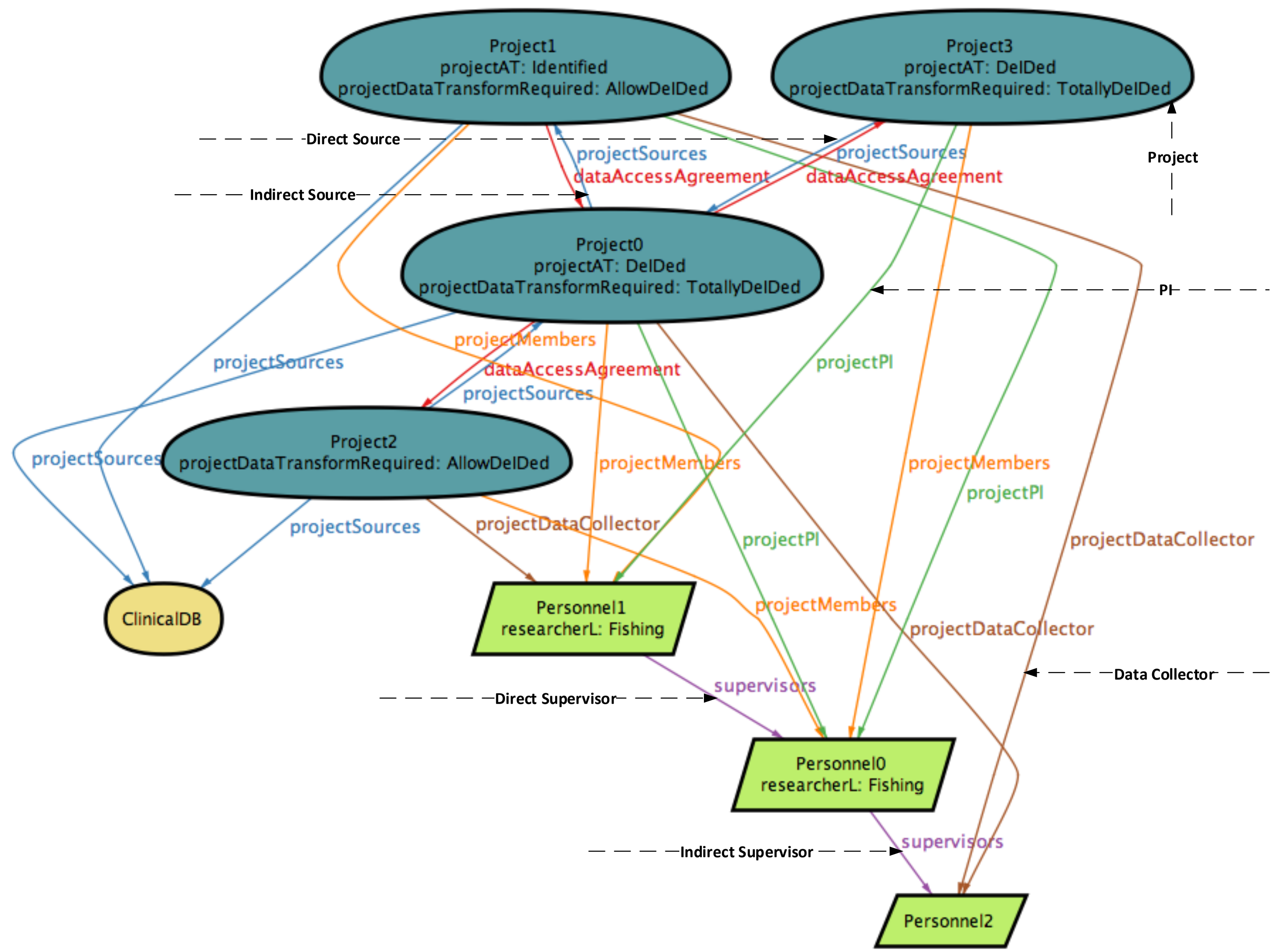


Figure 8.11: *DC* Conflict of Interest, Indirect Supervision of Project's Indirect Source's *DC* by Project's *PI*: *Project3* has indirect *Source Project1* and *Project3*'s *PI Personnel1* indirectly supervises *Project1*'s *DC Personnel2*.

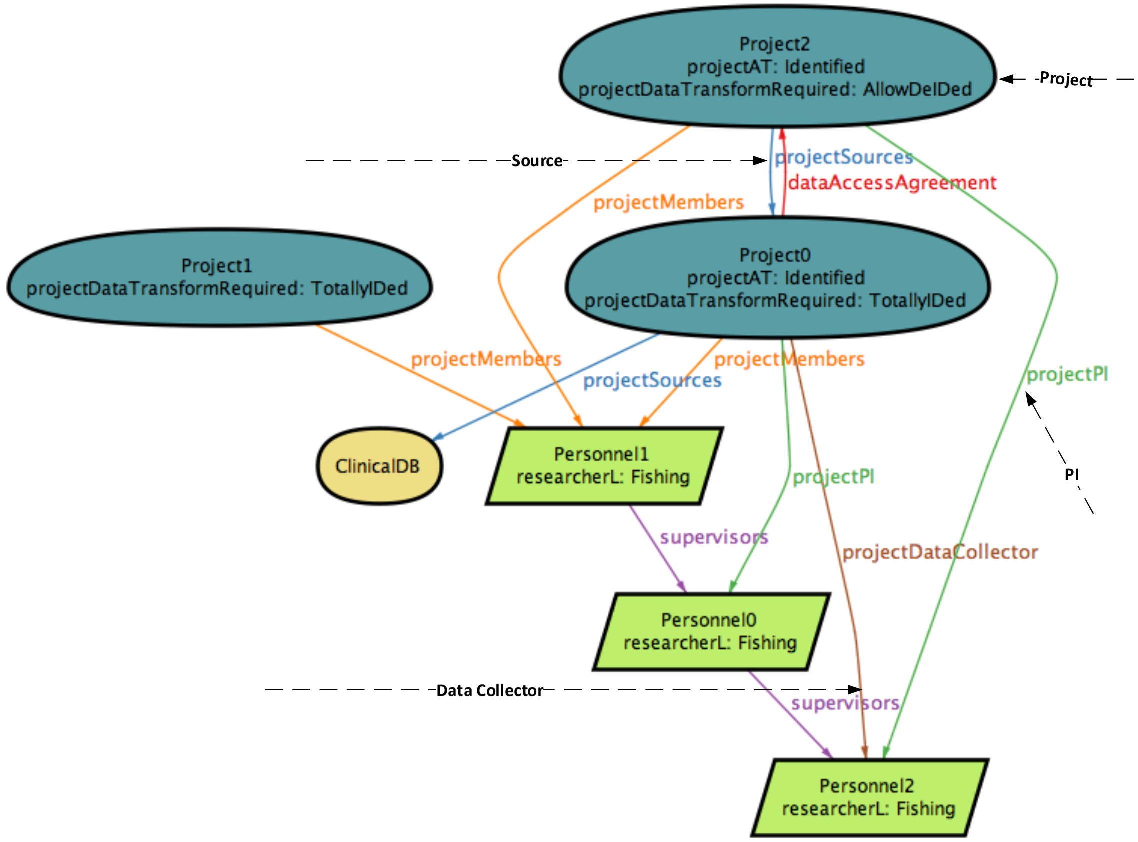


Figure 8.12: *DC* Conflict of Interest, Project's Direct *Source*'s *DC* is the same as the Project's *PI*: *Project2* has a *Source Project0*, and *Project2*'s *PI Project0*'s *DC* are the same, *Personnel2*.

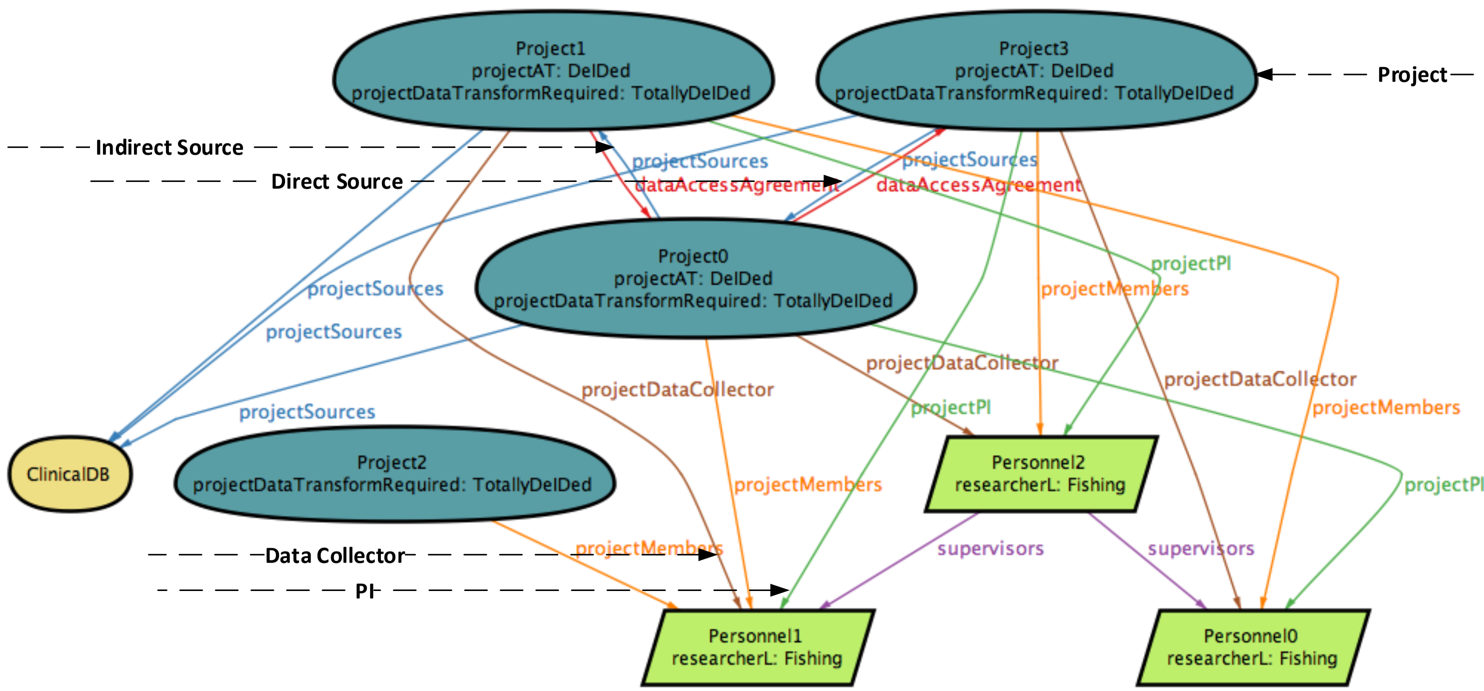


Figure 8.13: *DC* Conflict of Interest, Project's Indirect *Source*'s *DC* is the same as the Project's *PI*: *Project3* has indirect data source *Project1*, yet *Project3*'s *PI* is the same as *Project1*'s *DC Personnel1*.

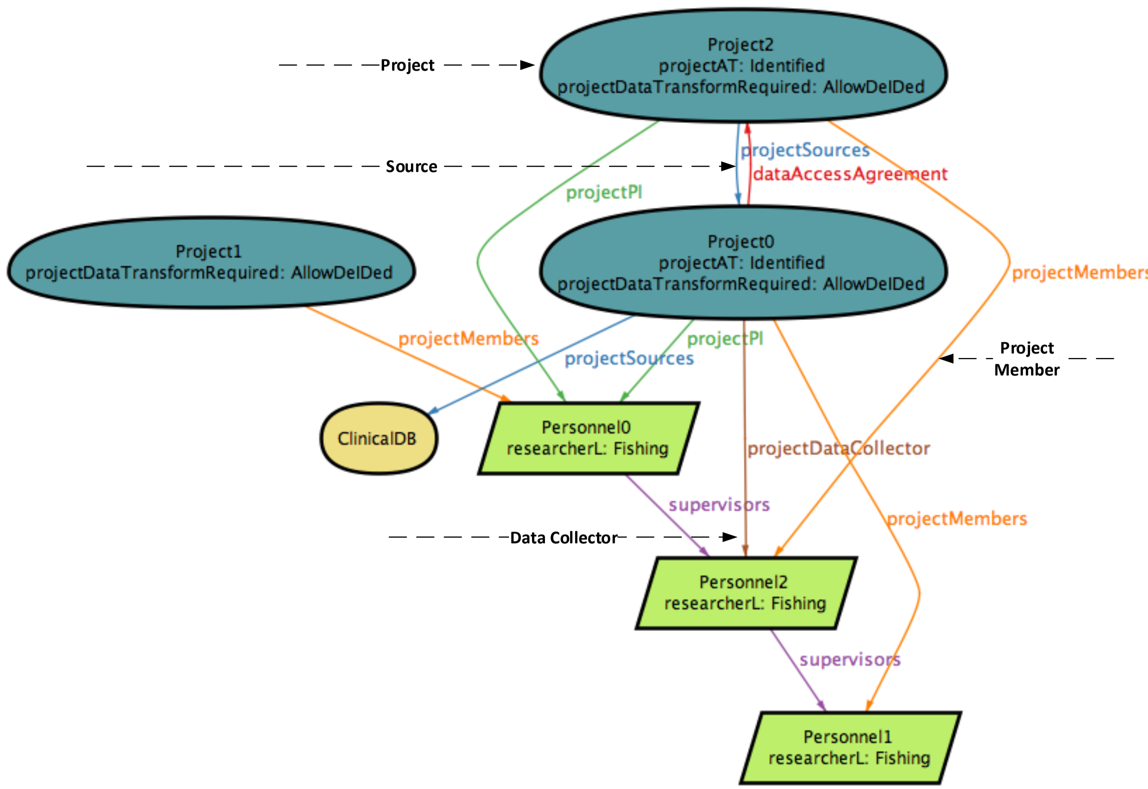


Figure 8.14: *DC* Conflict of Interest, Project's *PI* is the same as the *DC* for one of it Direct *Sources*: *Project2* has a *Source Project0* and one of *Project2*'s *PMs Project0*'s data collector are the same, *Personnel2*.

for *Project2*. The conflict of interest still exists if the supervisory relationship is indirect or if the *Source* was indirect. We show an example of this with the Alloy Analyzer partial instance in Figure 8.15 where we see that *Project3* has an approved access ticket, yet its one of it *PM*'s, *Personnel2* is the same as the *DC* for *Project1*, an indirect *Source* for *Project3*.

8.3.5 Eliminate *DC* Conflicts of Interest

In order to eliminate these conflict of interest situations from the system, we must:

1. update the *NoOverlapPITeamDC DecisionRule* in Figure 8.16 with the third and fourth situations so that there is never an overlap among the *PI*, *DC*, and *PM*s;
2. add a new *DecisionRule NoSupsInPIandDC* in Figure 8.16 for the first and second situations so that a *PI* never supervises the *DC* for any of its *Sources* and include this rule in the approval for an access ticket; and
3. update the conformance rule in the Alloy specifications with these four additional situations to the to ensure that there are no violations.

Figure 8.16 shows the new and updated *DecisionRule*s highlighted using the red dashed line. The complete Alloy specifications for slice 3, including the new and updated the decision rules, and NJH conformance rule, is in Appendix D.1.1.

8.4 Summary

We have shown two ways faults are commonly introduced into specifications and that HMCA uncovers the faults by showing conformance failures.

The first fault covers errors in logic that may arise because some specific logic connectors, i.e., *implies* and its stronger form *iff*, are not well understood. This fault is interesting because, while we showed that our specifications were correct with respect to the *DeIDed* access ticket in Chapter 7, when we extended our analysis to include the *Identified* access ticket and its two associated data transformations, there was non-conformance for the *DeIDed* access ticket. This non-conformance existed even though the specifications pertaining to the *DeIDed* access ticket remained the same.

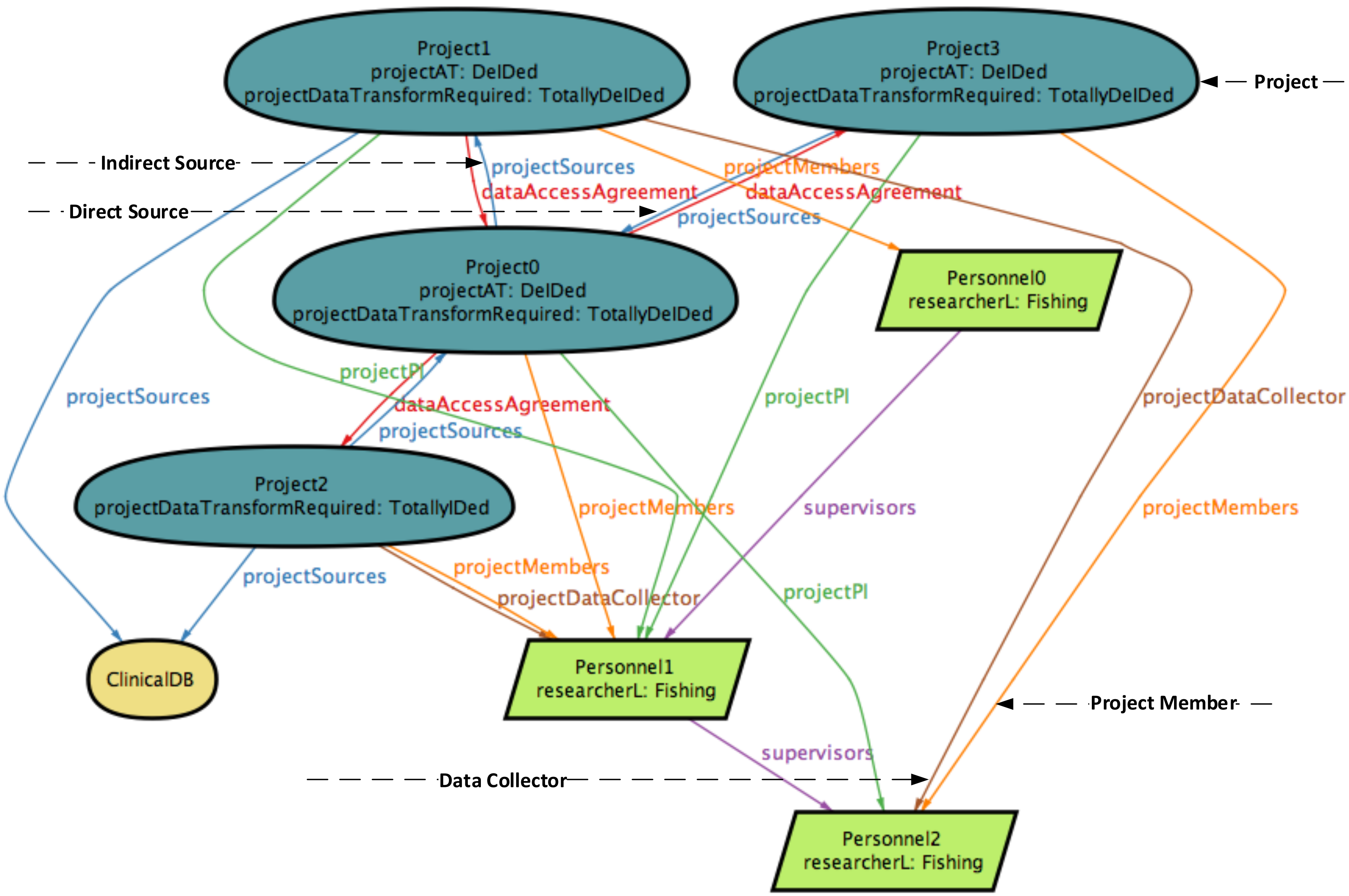


Figure 8.15: *DC* Conflict of Interest, Project's *PI* is the same as the *DC* for one of it Indirect *Sources*: *Project3* has indirect *Source Project1*, and one of *Project3*'s *PM*s is the same as *Project1*'s *DC Personnel2*.

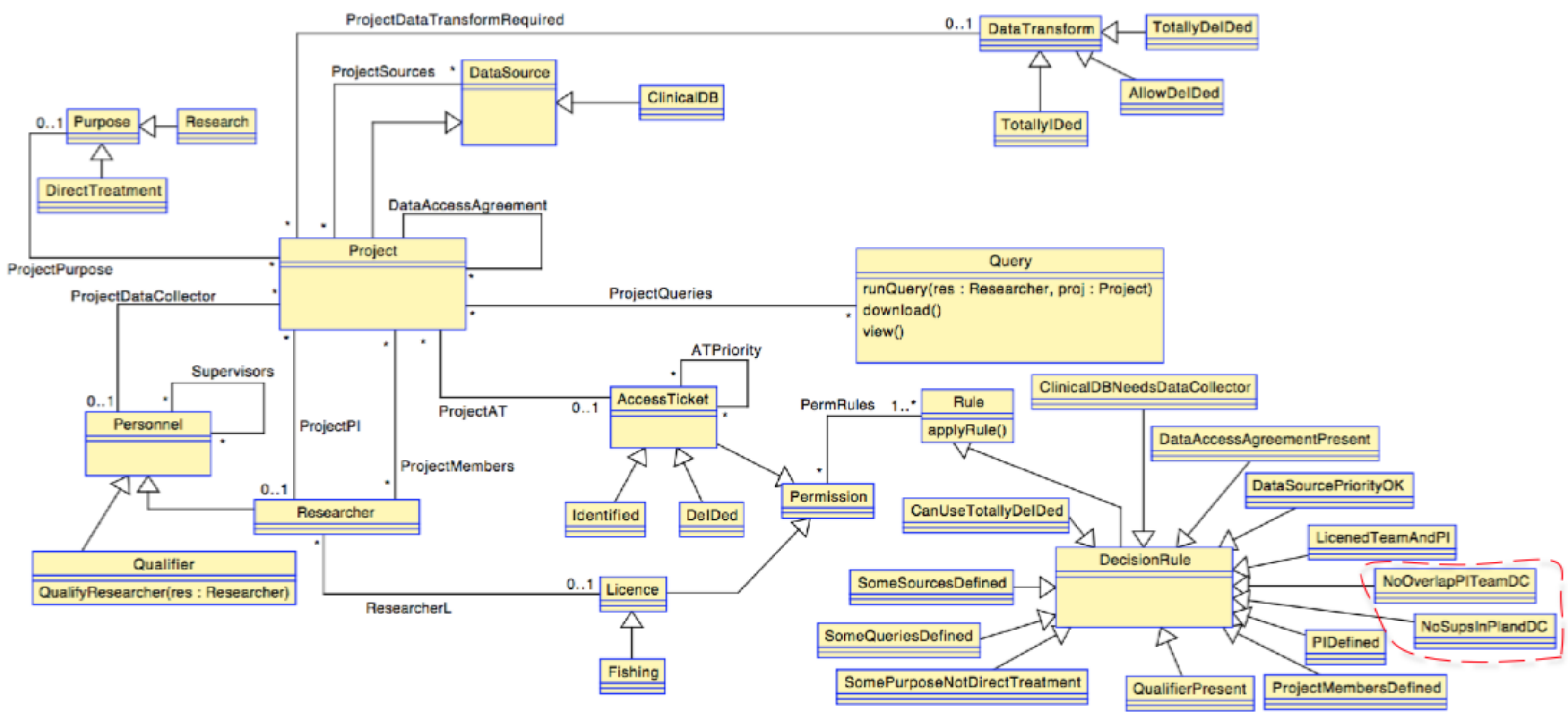


Figure 8.16: Updated Slice 3 with *DecisionRules* for Conflict Of Interest Situations Outlined by the Red Dotted line

An insight for this logic fault is that when more than one access tickets require the same data transformation, there needs to be a careful and intentional examination of whether the relationship between the access ticket together with the required data transformation and the format of the resulting data is a:

1. one-way relationship and in such a case *implies* is applicable, i.e., the former requires the latter, but the latter does not require the former; or

2. two-way relationship and in such a case *iff* is applicable, i.e., the former requires the latter, and the latter also requires the former.

In the NJH System this consideration is not only applicable to the *DeIDed* and the *Identified* access tickets, but it is applicable to other access tickets since some require similar transformations to those already discussed. For example, the *Coded* access ticket (see Section 3.4 for an explanation) mandates that some data that is returned by queries be *TotallyIdentified* and others, under a new data transformation, be *indirectly identifiable*. In this case the (new) *Coded* access ticket allows an *Identified* data transformation and we must evaluate whether the parts of the conformance rule in Listing 8.1 still holds when we add clauses for the new access ticket.

The second fault we discovered concerns a common way that specifications are incomplete: indirect relationships among objects are missed. Essentially, these missed relationships are transitions in the NFA rule representation (see sections 6.2 and 5.1.3) that should lead to an accepting state (non-conformance), yet are not specified to do so in the Alloy specifications. These indirect relationships can only be uncovered by computing all the ways objects can be related, i.e., computing relationship closures.

We note that we may refine these conflict of interest situations caused by indirect relationships further. For example, an organisation like the NJH may run into problems because satisfying conflict of interest conformance rules may require an increase in the number of personnel when the number of approved projects is increased. For example, a *DC* on one project can only take on the role of a researcher for another non-conflict of interest (directly and indirectly) project. If the *DC* has a conflict of interest with all the current projects, then other new personnel are required for this *DC* to take on the role of, say, a *PI* on a new project. However, acquiring new personnel may not be possible because of budgetary or other constraints. One solution for this is to include in the specifications the idea of project lifetimes, and specify conflicts of interest where project lifetimes overlap. Here, the work of [16, 83] that formalises overlapping lifetimes in whole-part relationships would be pertinent in understanding the ways projects lifetimes may overlap.

Validating HMCA is as important step is showing that conformance failures can be detected for some common types of faults. The addition of the *Identified* access ticket required that we add to the specifications model elements for data transformations and associate them with the appropriate access tickets. This increase of model elements did not significantly affect the conformance analysis. Another way to validate HMCA is to evaluate larger model slices due to, not just an increase in associations among current model elements, but an increase due to adding new classes and associations among new and older model elements. For this we will augment the NJH system with rules for protected populations, specifically children protected populations, in Chapter 9.

9. APPLYING HMCA TO CHILDREN AS PROTECTED POPULATIONS IN THE NJH

9.1 Introduction

HIPAA regulations mandate that sharing information on protected populations, such as children, pregnant women, foetuses and neonates, and prisoners must include additional protections over the kinds of protections allowed by a given access ticket. In this chapter we expand our model of the NJH system to include the HIPAA regulations as rules for the protection of children. The specific changes include:

1. an organisation's Institutional Review Board (IRB) is required to also consider rules that govern the use of children in research when approving access tickets; and
2. where approval has been given, additional rules give the conditions under which such data may be accessed.

We have chosen to model the rules governing access to data for children as this is important to the NJH due to the sensitive nature of accessing data for children. We discuss the HIPAA regulations concerning children and how they are realised in our model in Section 9.2 and a summary that includes a discussion on why our specification of the children protected population helps us to be able to extend the specification for other protected populations in Section 9.3.

9.2 Requirements for Protecting Children in the HIPAA Regulations

The HIPAA regulations for protected populations in [69] stipulate that when an IRB approves proposals for research, they must implement these additional protections for children included in research:

1. quantify the risk to the children such that if the risk is too great then no approval is issued to conduct the research; and
2. when approval is given:
   (a) to specify required additional assent from each child and consent from the parent, guardian, or the ward organisation responsible for the child; and

(b) to require that children who are wards be assigned an advocate who is not connected in any way to the research or the ward organisation.

We have updated the overall class model to include new model elements to capture the requirements for the children protected population. Recall that in chapters 7 and 8 we discussed that an access ticket is approved in slice 3 and queries are executed in slice 4. We have therefore re-sliced the overall model and will discuss Item 1 as it applies to slice 3 in Section 9.2.1 and Item 2 as it applies to slice 4 in Section 9.2.2.

9.2.1 Approving Access Tickets to Use Children Protected Populations

Figure 9.1 shows the new slice 3 that now supports approving access tickets requiring the use of children. The additional elements are enclosed using red dashed lines and annotated using grey shaded circles numbered 1 through 6. The following list of numbered items correspond to the numbered circles:

1. the *ProjectSpecialResearch* association is used by the project to indicate that the project application for an access ticket includes access to the protected populations indicated. Currently, only the *Children* protected population is supported, however, the model is set up to allow extending the *SpecialSubject* class with other populations.

2. the *ProjectSpecialResearchApproval* association records the IRB's decision on whether to grant the project approval to use the special populations requested by the project. We use the *Allow* class to indicate that approval has been given and the *DisAllow* class to indicate the approval has not been given. Each decision must be accompanied by an indication of the risk exposure represented by the *ResearchRisk* class: an approval is indicated by any of the *ChildrenResearchRisk* subclasses except the *RiskNotAllowed* that is reserved for decisions that are not approved.

3. the *IRBMembers* association indicates the members of the IRB. It is important to include this association in our model because the applying *DecisionRule* in Item 4b below requires it.

4. the *PermRules* association includes a new and an updated *DecisionRule* for issuing an access ticket:

   (a) the *SpecialResearchApproved* is a new *DecisionRule* that checks that all special populations indicated in Item 1 have approval in Item 2 before a project's access ticket can be approved; and

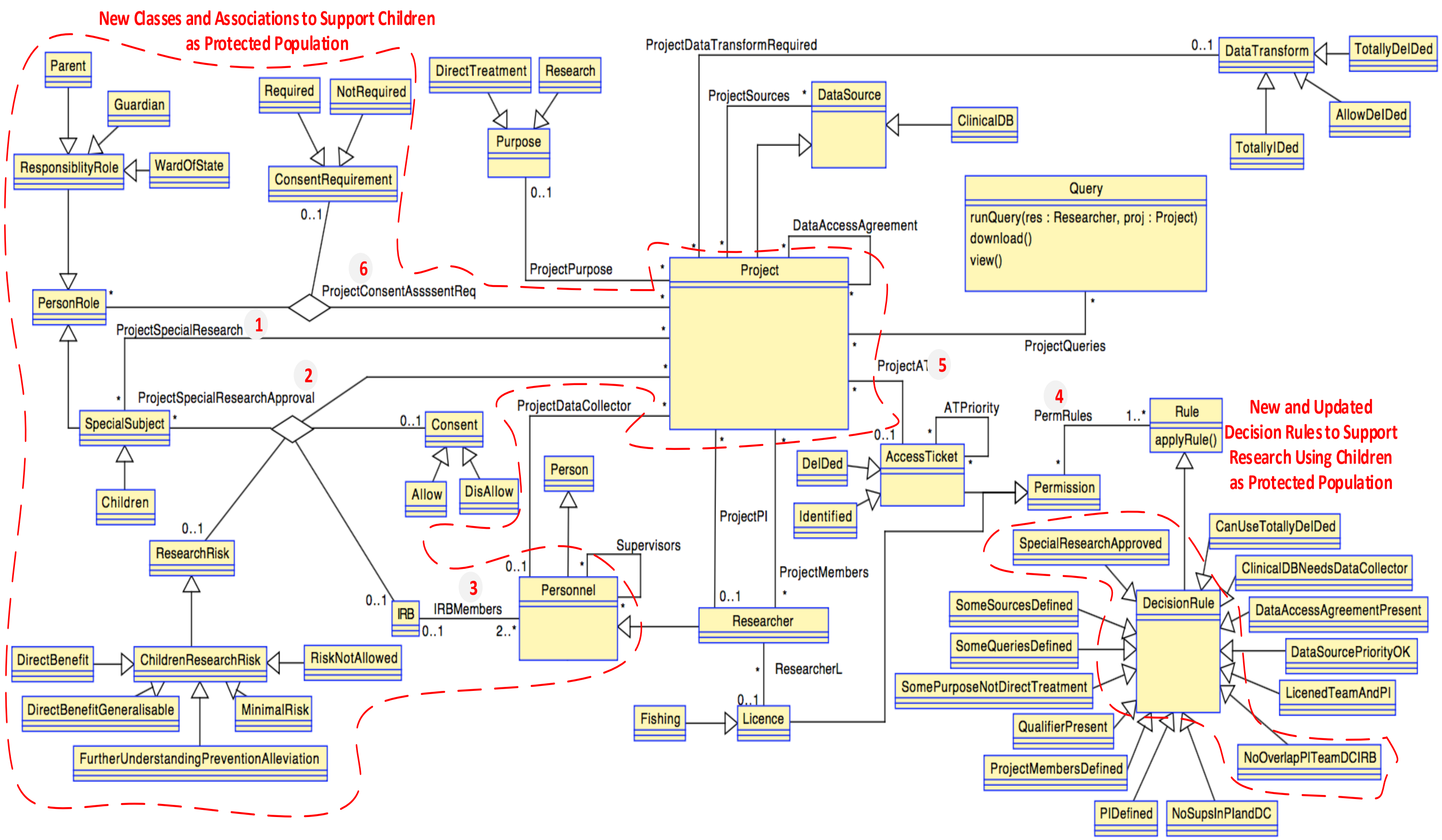


Figure 9.1: Updated Class Model for Slice 3 Supporting Children as a Protected Population (new class model elements outlined by the dashed red lines)

(b) *NoOverlapPITeamDCIRB* is an updated *DecisionRule* (previously *NoOverlapPITeamDC*) to now include that no IRB member is allowed to be a part of the project team[1].

5. if no applicable *DecisionRule* is violated, the project may be approved its access ticket application - a link between a project and an access ticket in the *ProjectAT* association records this. For example, the following set of object models for Figure 9.1 highlight important scenarios when we may approve or not approve an access ticket for a project:

   (a) Figure 9.2 is an object model that shows that *Project_1*'s access ticket is approved as no *DecisionRule* is violated (see annotation numbered 5). In this example, we note that the IRB has indicated that there is a *DirectBenefit* to the children and has therefore approved the request for *Project_1* to use children in their research (see numbered association 3). In addition, no *IRBMember* has a conflict of interest with the project, i.e., the personnel in association numbered 3 have no links with any of the personnel associated with the project.

   (b) Figure 9.3 is an object model that shows that *Project_1*'s access ticket cannot be approved because the IRB indicated that the risk was too great (see annotations numbered 2 and 5);

   (c) Figure 9.4 is an object model that shows that *Project_1*'s access ticket cannot be approved because the *IRBMember*, *Personnel1* is the project's *DataCollector* (follow annotation numbered 3 from *Personnel1* along the *ProjectDataCollector* association link to *Project_1*); and

   (d) Figure 9.5 is an object model that shows that *Project_1*'s access ticket cannot be approved because the IRB indicated that the risk was too great because the access ticket applied for is the *DeIDed* access ticket (see annotations numbered 3 and 5).

6. the *ProjectConsentAssentReq* association is required for all approved decisions in the *ProjectSpecialResearchApproval* association. The IRB uses this association to indicate whether each child and/or parent/guardian/ward organisation are required to give assent or consent respectively for the child's data to be used by the project. We see an example of this in Figure 9.2 where the IRB has indicated

[1] As with checking the conflict of interests for the old *NoOverlapPITeamDC* rule explained in sections 8.3.2, 8.3.3, and 8.3.4, the updated rule must check for direct and indirect linkages of an *IRBMember* through the closure of the *ProjectSources* associations.

that *Project_1* must get explicit assent and consent from the child and the parent/guardian/ward organisation of the child respectively as a precondition for including the child's data in their research (see annotations numbered 6).

9.2.2 Executing Queries With Access Tickets Approved for Children Protected Populations

In order to execute queries where a project uses protected populations, elements of the class model for approving an access ticket for children must be used. Specifically, we need to include the associations numbered 1, 3, and 6 from Figure 9.1. We show in Figure 9.6 the class model elements from slice 3 that overlap in slice where we execute a query for a project requiring the use of children.

Figure 9.7 shows the re-sliced slice 4 that now supports executing queries with access ticket for projects requiring the use of children. The additional elements are enclosed in the shaded region outlined by the red dashed line and grey shaded circles numbered 1 and 3 through 13. Note that the numbered annotations 1 and 3 through 6 are the same associations from slice 3 in Figure 9.1. The following list of numbered items correspond to the numbered circles:

1. We have already given an explanation for the *ProjectSpecialResearch* association in Section 9.2.1, Item 1. This association is needed so that we know when a project is allowed to access specific protected populations.

2. This association and corresponding annotation are not required in slice 4.

3. We have already given an explanation for the *IRBMembers* association in Section 9.2.1, Item 3. Though this association is not explicitly required in slice 4, we include it because of potential conflict of interest situations that can arise. We will return to this discussion in Section 9.2.2.1.

4. Instead of *DecisionRule*s as discussed in Section 9.2.1, Item 4, the *PermRules* association now links to *AccessRules*. Here, we include four new access rules to support the children protected population. In order to explain the rules, we use object models that are instances of Figure 9.7 in figures 9.8 through 9.12 to highlight examples where children data may be accessed because no *AccessRule* is violated

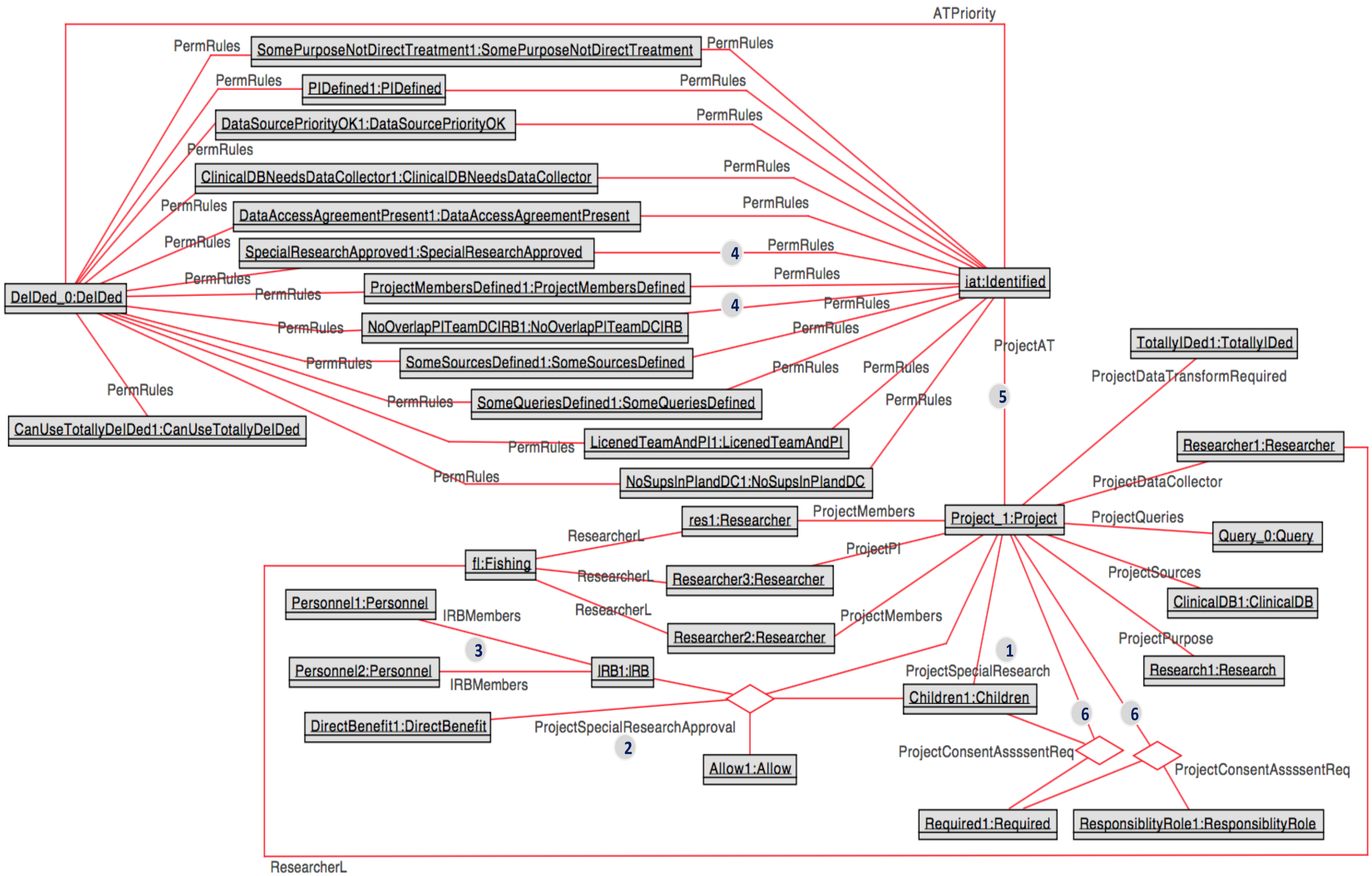


Figure 9.2: Slice 3 Object Model for approved *Identified* access ticket for *Project_1* using all *DecisionRules* (see annotation 5). Also to use the data for the children protected population, each child and parent/guardian/ward organisation of the cild must give explicit assent and consent respectively (see annotation numbered 6). Numbered annotations correspond to associations so numbered in Figure 9.1 and explained in Section 9.2.1.

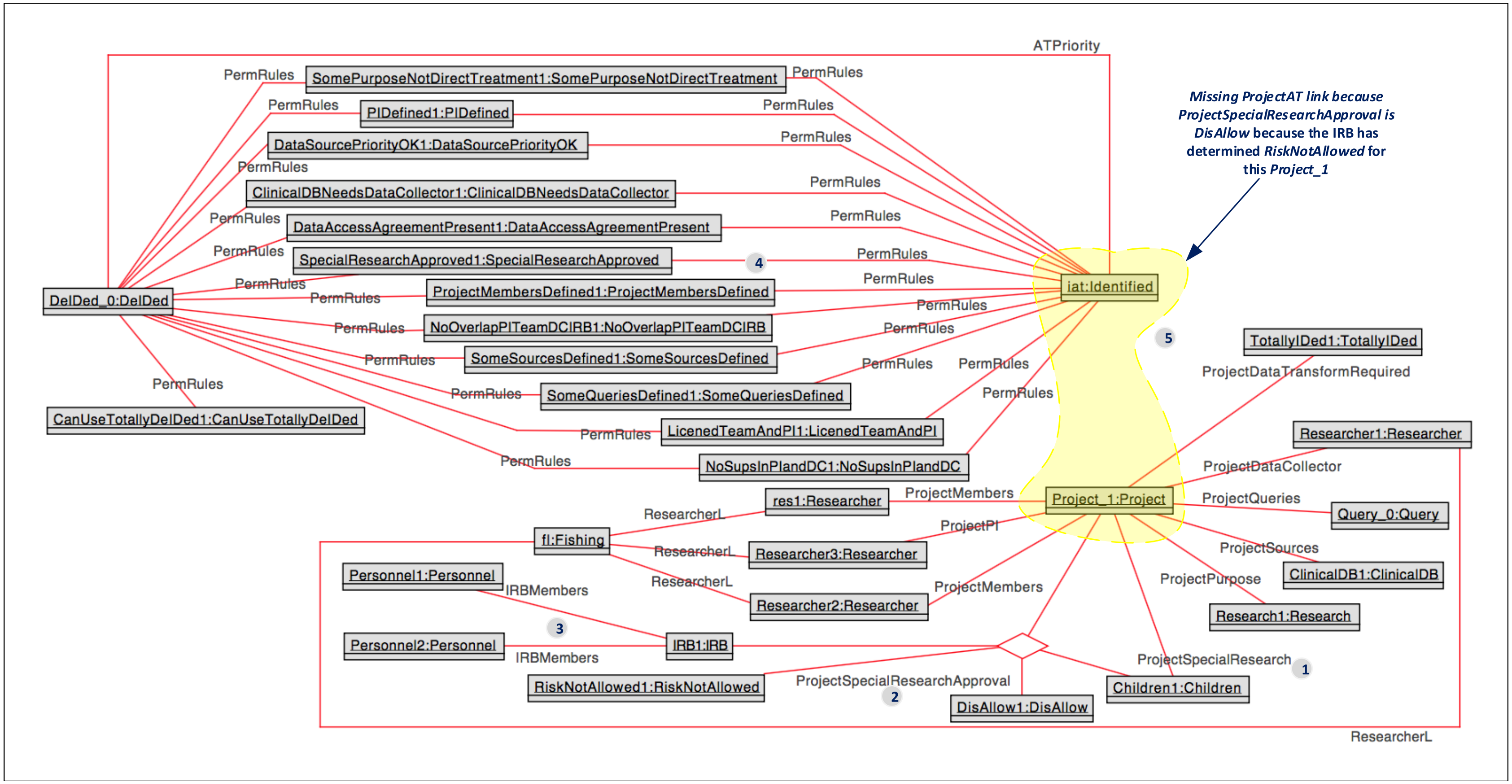


Figure 9.3: Slice 3 Object Model for Unapproved, i.e., cannot be approved, *Identified* access ticket for *Project_1* using new *DecisionRules* because the *IRB* has determined that *RiskNotAllowed*. Numbered annotations correspond to associations so numbered in Figure 9.1 and explained in Section 9.2.1.

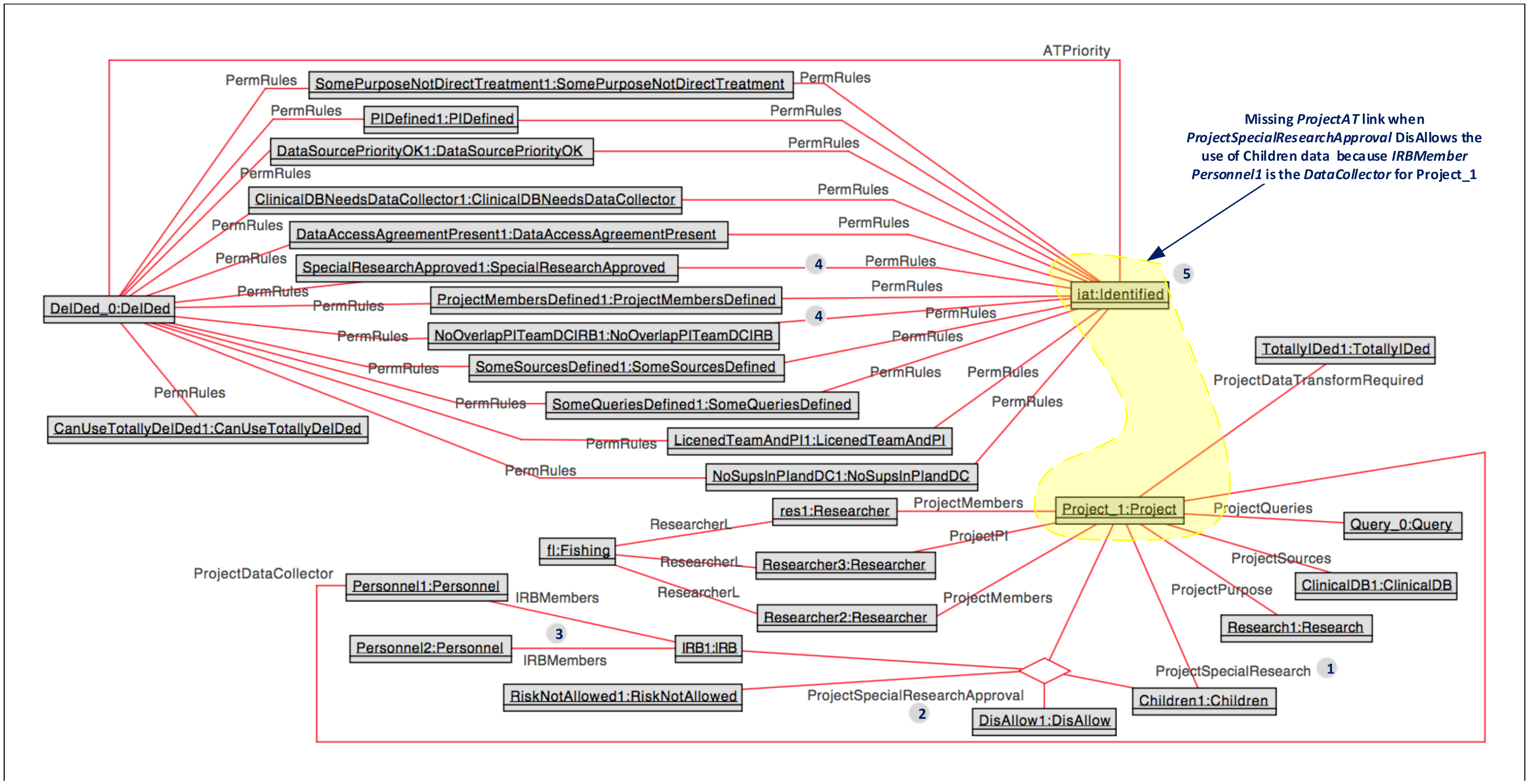


Figure 9.4: Slice 3 Object Model for unapproved, i.e., cannot be approved, *Identified* access ticket for *Project_1* using new *DecisionRules* because of a conflict of interest: *Personnel1* is an *IRBMember* and the *ProjectDataDollector* for *Project_1*. Numbered annotations correspond to associations so numbered in Figure 9.1 and explained in Section 9.2.1.

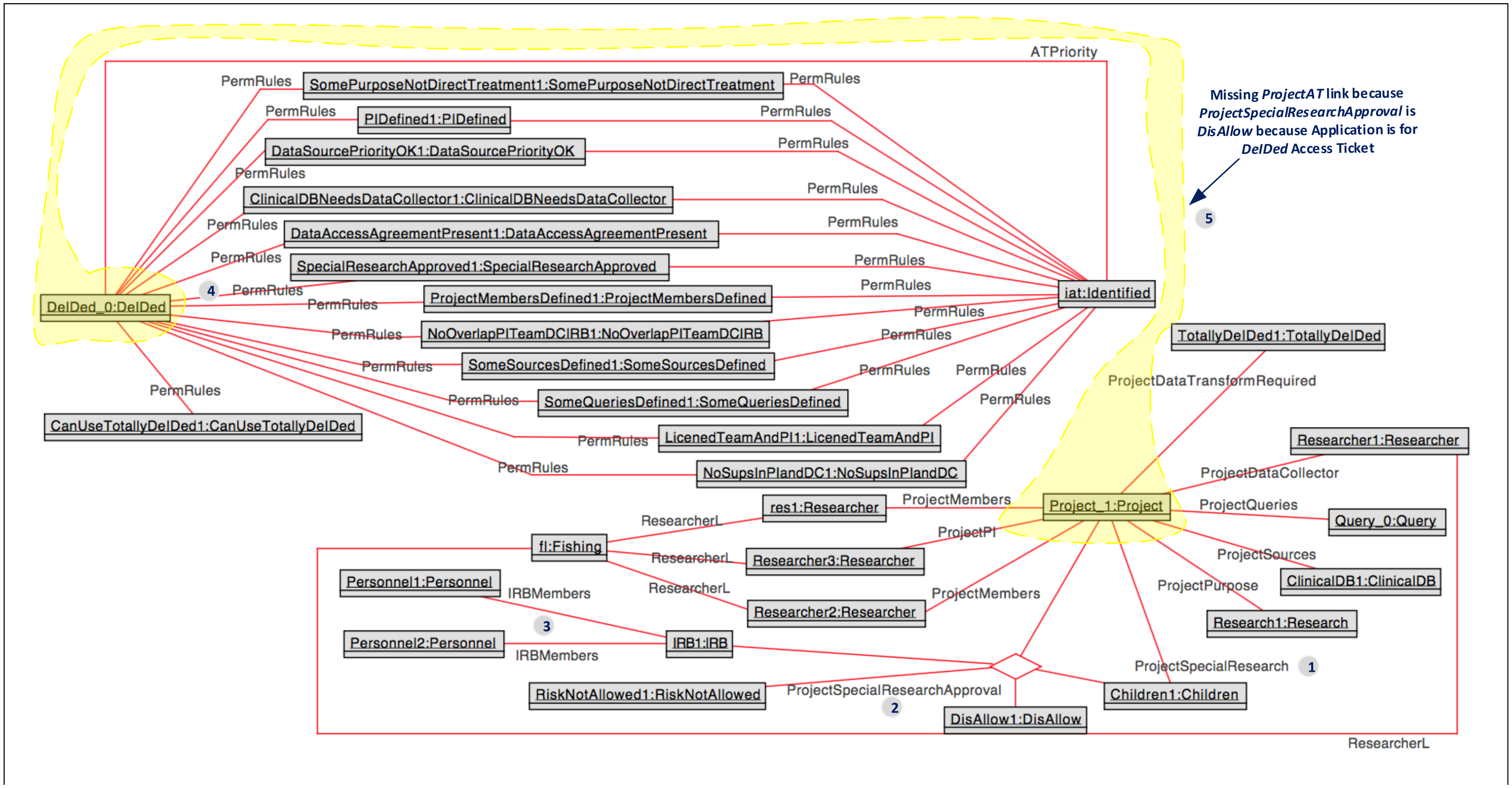


Figure 9.5: Slice 3 Object Model for Unapproved, i.e., cannot be approved, *DeIDed* Access Ticket for *Project_1* using New *DecisionRules* because a *DeIDed* access ticket cannot be used to access protected populations. Numbered annotations correspond to associations so numbered in Figure 9.1 and explained in Section 9.2.1.

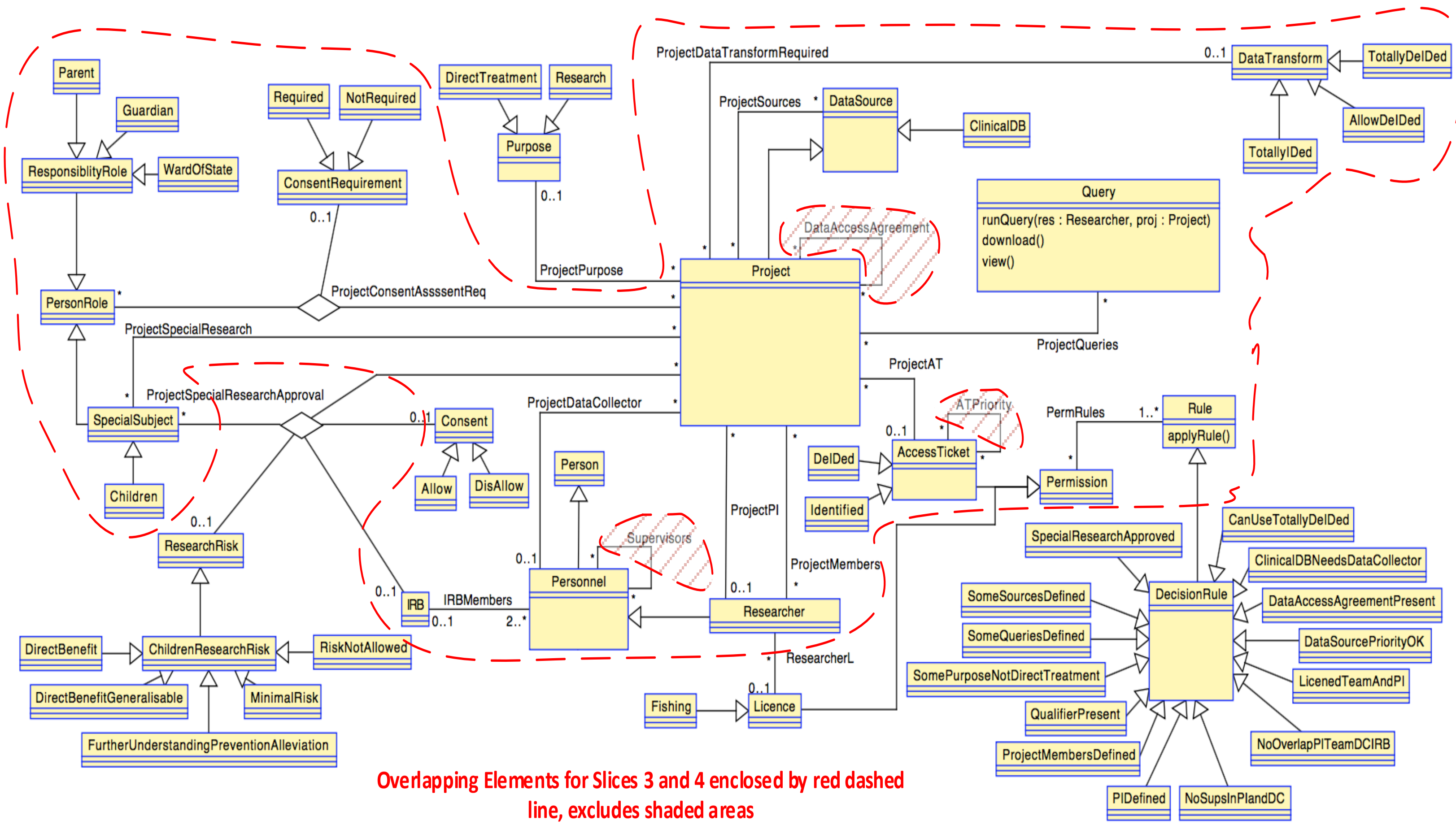


Figure 9.6: Class Model Elements from Slice 3 Overlapping in Slice 4 (outlined by the dashed red line)

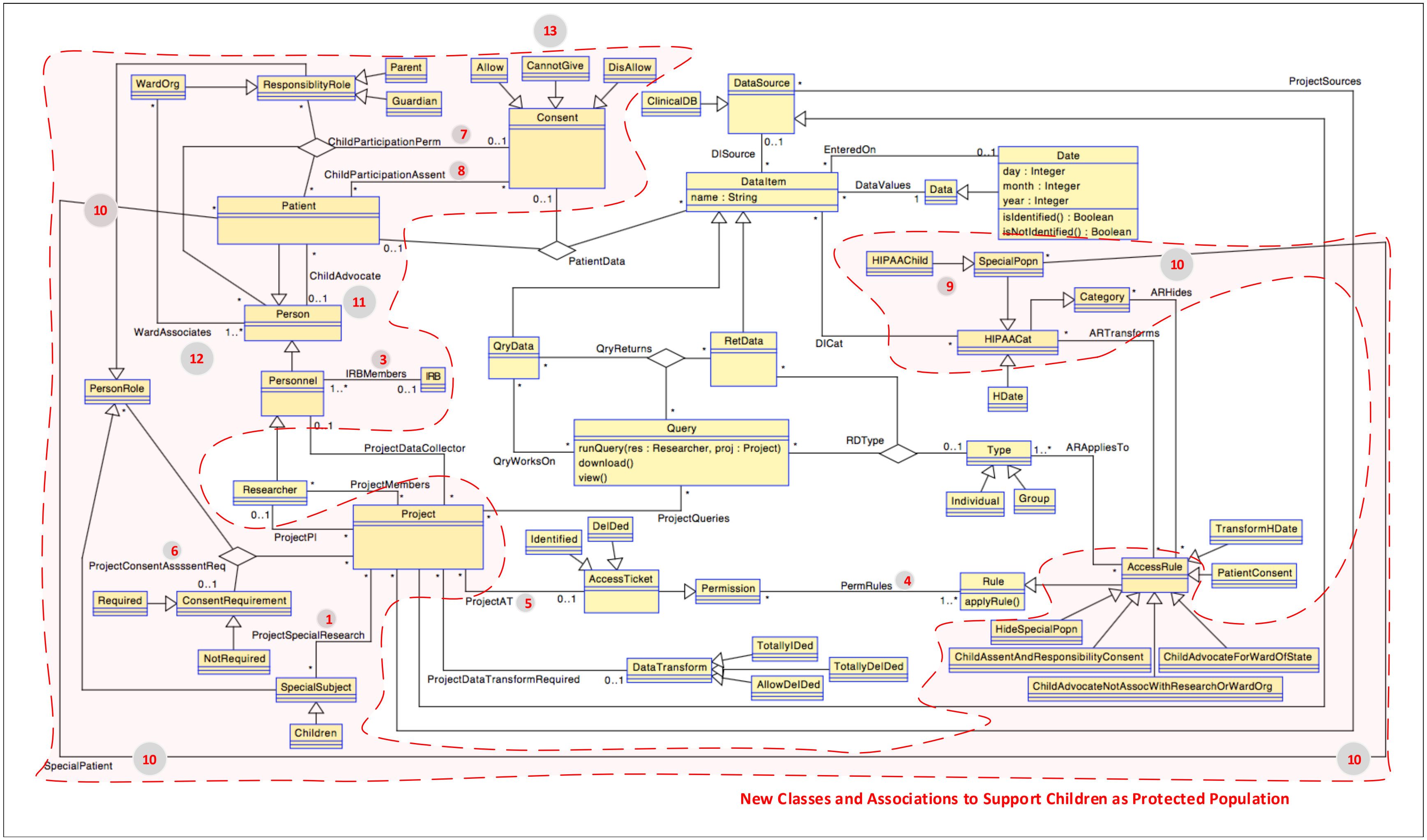


Figure 9.7: Updated Class Model for Slice 4 Supporting Children as Protected Population: new elements outlined by the dashed red line

and where children data may not be accessed because at least one *AccessRule* is violated. We list the examples here:

(a) No violation of access rules: we show in Figure 9.8 where *Query_0* successfully accesses the data for *Patient2* because none of the access rules have been violated. Since we have not yet discussed the *AccessRule*s, our intention is presenting this first is for comparison with the violations of access rules explained in items 4b through 4e and depicted in figures 9.9 to 9.12 below.

(b) Violation scenario 1: the *ChildAssentAndResponsibilityConsent* rule only allows access to a child's data if the assent/consent as required in the *ProjectConsentAssentReq* association is present in the associations numbered 7 and 8 (see items 7 and 8 below for a description of these associations). For example, Figure 9.9 shows that *Query_0* should never have access to *Patient2*'s data because this patient is a child and has not given assent.

(c) Violation scenario 2: the *ChildAdvocateForWardOfState* rule requires that a child who is the ward of any institution have an advocate. For example, Figure 9.10 shows that *Query_0* should never have access to *Patient2*'s data because though they are a ward of *WardOrg1* there is no person assigned as an advocate for them.

(d) Violation scenario 3: the *ChildAdvocateNotAssocWithResearchOrWardOrg* rule expresses that there should not be a conflict of interest between the person acting as the advocate for a child and those associated with the *WardOrg* to which the child belongs or with those conducting the research. For example, Figure 9.11 shows that *Query_0* should never have access to *Patient2*'s data, a ward of *WardOrg1*, because while they have an advocate (so rule *ChildAdvocateForWardOfState* is not violated), this advocate, *Personnel1*, is an associate of *WardOrg1*. Note that there is no conflict of interest with an advocate also serving as an *IRBMember* as shown for *Personnel1* (see annotations numbered 11 and 3).

(e) Violation scenario 4: the *HideSpecialPopn* rule ensures that for the *DeIDed* access ticket, all protected population should be inaccessible. For example, Figure 9.12 shows that *Query_0* should never have access to *Patient2*'s data because the access ticket for *Project_1*, under which *Query_0* executes, is *DeIDed*.

5. We have already given an explanation for the *ProjectAT* association in Section 9.2.1, Item 5. It is required in slice 4 to know the access ticket for a project.

6. We have already given an explanation for the *ProjectConsentAssentReq* association in Section 9.2.1 Item, 6. It is required in slice 4 to check the *ChildAssentAndResponsibilityConsent AccessRule*. An example of violating this rule has already been discussed in Item 4b above.

7. The *ChildParticipationPerm* indicates whether the child's parent/guardian/ward organisation has given consent for the child's data to be used in research. This consent is given if the *Consent* value is *Allow* and explicitly refused if the value is *DisAllow*. The *CannotGive* consent value is not applicable to this association.

8. The *ChildParticipationAssent* indicates whether the child has given assent to be used in research. This assent is given if the *Consent* value is *Allow*, explicitly refused if the value is *DisAllow*, and in cases where the child cannot explicitly agree to or refuse to participate in the research, the value is *CannotGive* (see Item 13 below for an expansion of this *Consent* value). In the case of the latter, the child's data can also be used in the research if the parent/guardian/ward organisation gives *Allow* consent.

9. We include special HIPAA categories for special populations that are used (e.g., *HDate*) to indicate that special rules apply to data associated with such categories. Here we include *HIPAAChild* to support identifying data that belongs to children. This class is a specialisation of *SpecialPopn* so that the model can be extended to support other protected populations.

10. Each patient that is included in a special population is indicated using the *SpecialPatient* association. For example, figures 9.8 through 9.9 show that *Patient2* is a child (see annotation numbered 10 in the figures).

11. The *ChildAdvocate* association is used to link a child to an advocate. This association is important in the checking of the *ChildAdvocateForWardOfState* and the *ChildAdvocateNotAssocWithResearchOrWardOrg* access rules as discussed in items 4c and 4d above respectively.

12. The *WardAssociates* association is used to link persons to a ward organisation. This is association is important in the checking of the *ChildAdvocateNotAssocWithResearchOrWardOrg* access rule as discussed Item 4d above.

13. We have included another subclass of *Consent* because the HIPAA regulations stipulate that while the child's assent should be sought, there may be cases when it cannot be given because the child is incapable of doing so. Therefore, the *CannotGive* subclass records this and is interpreted as allowing access to the child's data.

9.2.2.1 Potential Conflict of Interests Not Considered under HIPAA

While the *NoOverlapPITeamDCIRB DecisionRule* and the *ChildAdvocateNotAssocWithResearchOrWardOrg AccessRule* cover specific conflicts of interest among personnel involved in a project and persons associated with patients in special populations, an examination of the models seen so far shows the potential for additional situations not explicitly covered under the HIPAA regulations. For example, Figure 9.13 shows that *IRBMember*, *Personnel2*, is the parent for *Patient2* (see annotations numbered 3 and 7). In this situation, a potential conflict of interest arises because of the objectivity required by *IRBMembers* when approving access tickets for a project. As an extension of this idea, consider the situation where *Personnel2* is the *PI*, *DataCollector*, or *ProjectMember* for *Project_1*. Should *Query_0* be allowed to access the data for *Patient2*? While our method does not make a decision to restrict access in these scenarios, the exercise of modelling shows that we can potentially explore these relationships and uncover links not pre-determined to be problematic. This ability can help organisations avoid conflicts of interest.

9.3 Summary

We have shown how our model supports children as a protected population by extending the overall model and re-slicing to get new sliced models for slice 3 and slice 4. Additionally, we have discussed situations under which an access ticket should not be issued and when data should not be accessible even if an access ticket has been issued under the new rules for these populations. We also showed some areas where HIPAA is

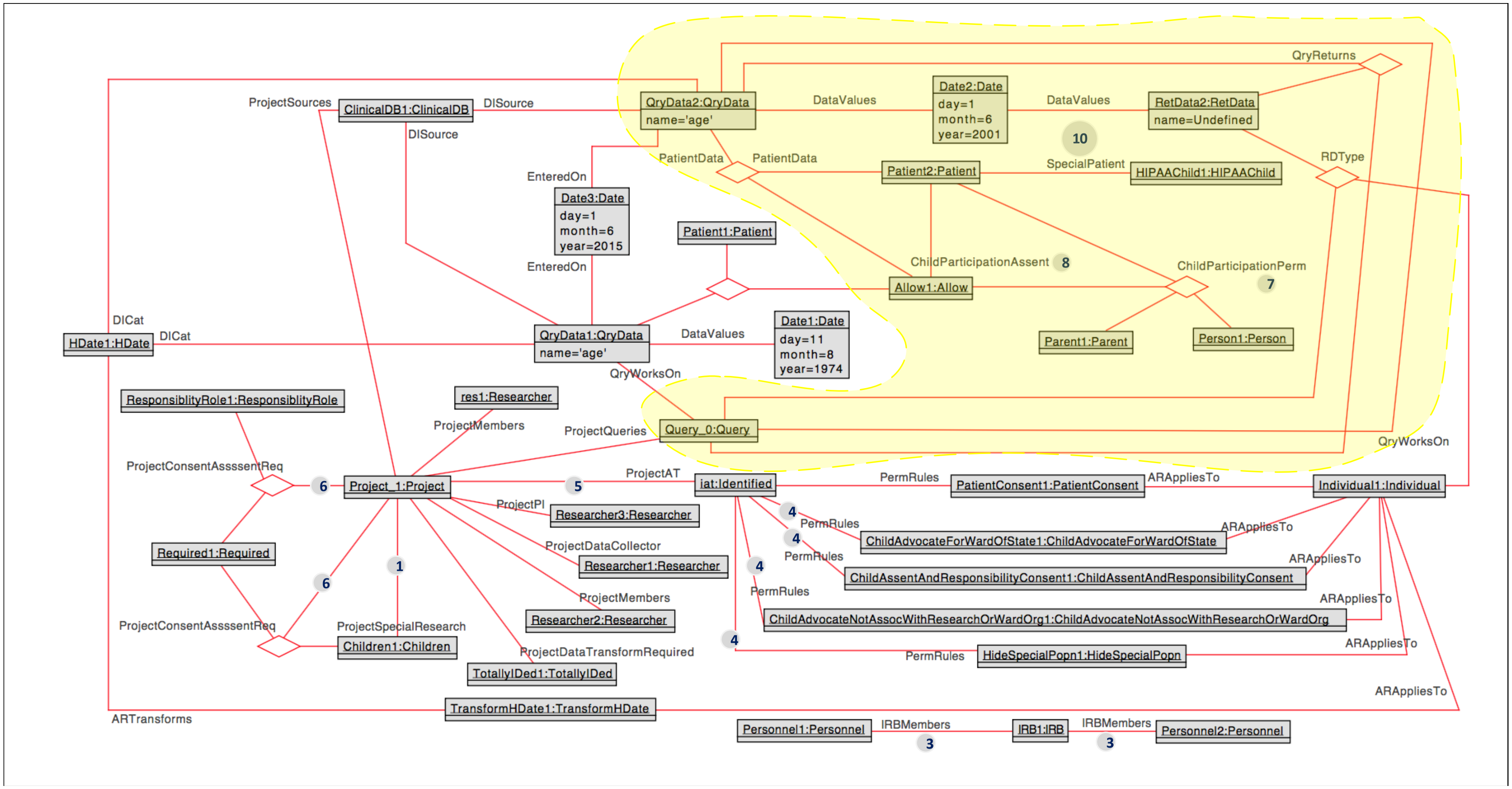


Figure 9.8: Object Model For Slice 4 showing that *Query_0* correctly accesses and returns *QryData2*, the data for *Patient2* identified as a *HIPAAChild*, because no *AccessRule* prohibits access (focus is on relationships in the area highlighted in yellow). Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

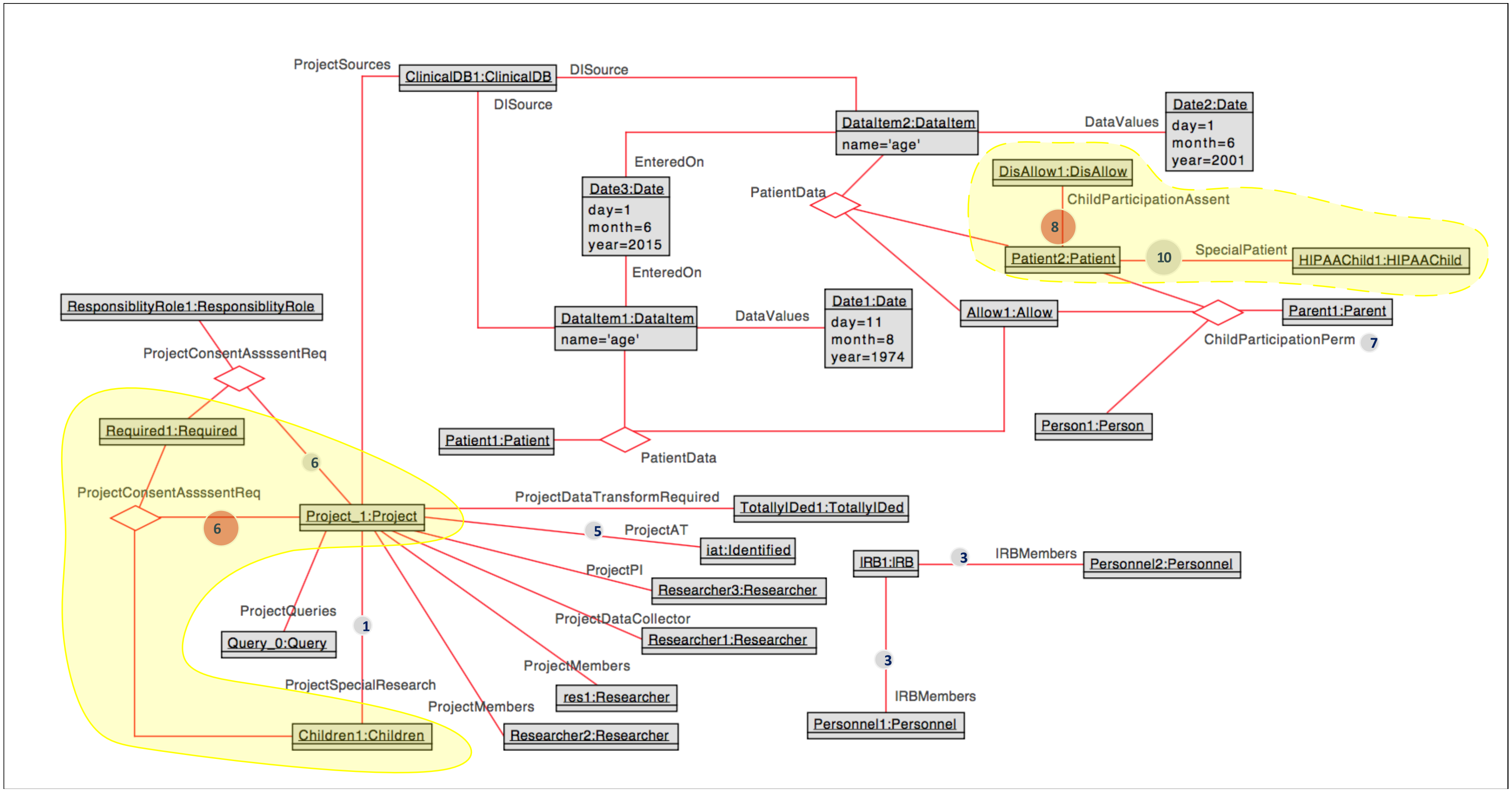


Figure 9.9: Access Denial Scenario 1: (Partial) Object Model for Slice 4 showing that *Query_0* must be denied access to *DataItem2* belonging to *Patient2* (focus is on relationships in the area highlighted in yellow). This is because the *ChildAssentAndResponsibilityConsent AccessRule* and the *ProjectConsentAssentReq* (see line annotated with 6) require that *Patient2* give *Allow* assent to participate in the research - yet the *ChildParticipationAssent* association link to *Patient2* (see association annotated with 8) shows *DisAllow*. Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

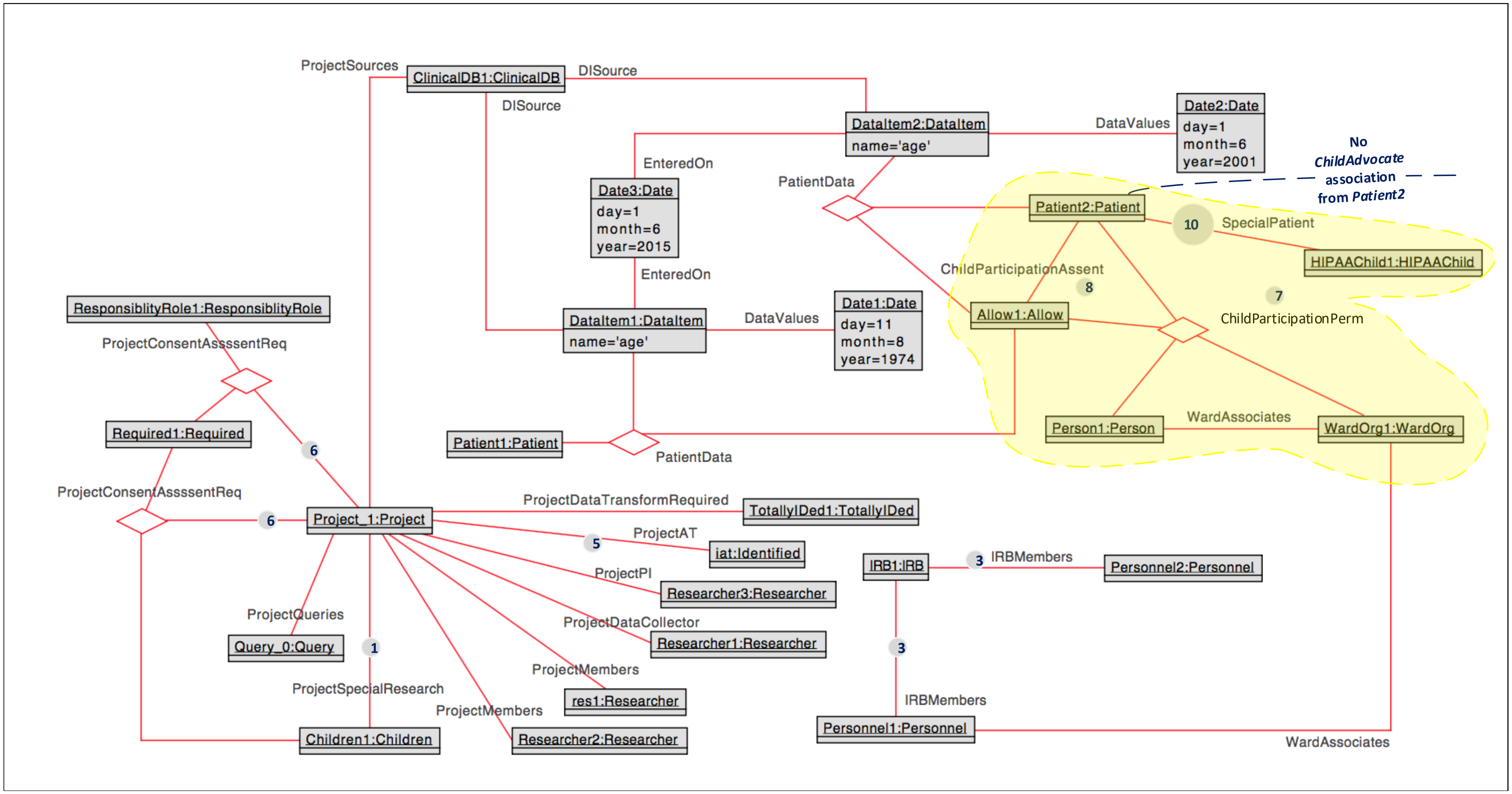


Figure 9.10: Access Denial Scenario 2: (Partial) Object Model for Slice 4 showing that *Query_0* must be denied access to *DataItem2* belonging to *Patient2* (focus is on relationships in the area highlighted in yellow). This is because the *ChildAdvocateForWardOfState AccessRule* requires that *Patient2*, a ward of *WardOrg1*, be associated with an advocate through the *ChildAdvocate*, yet this link is missing. Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

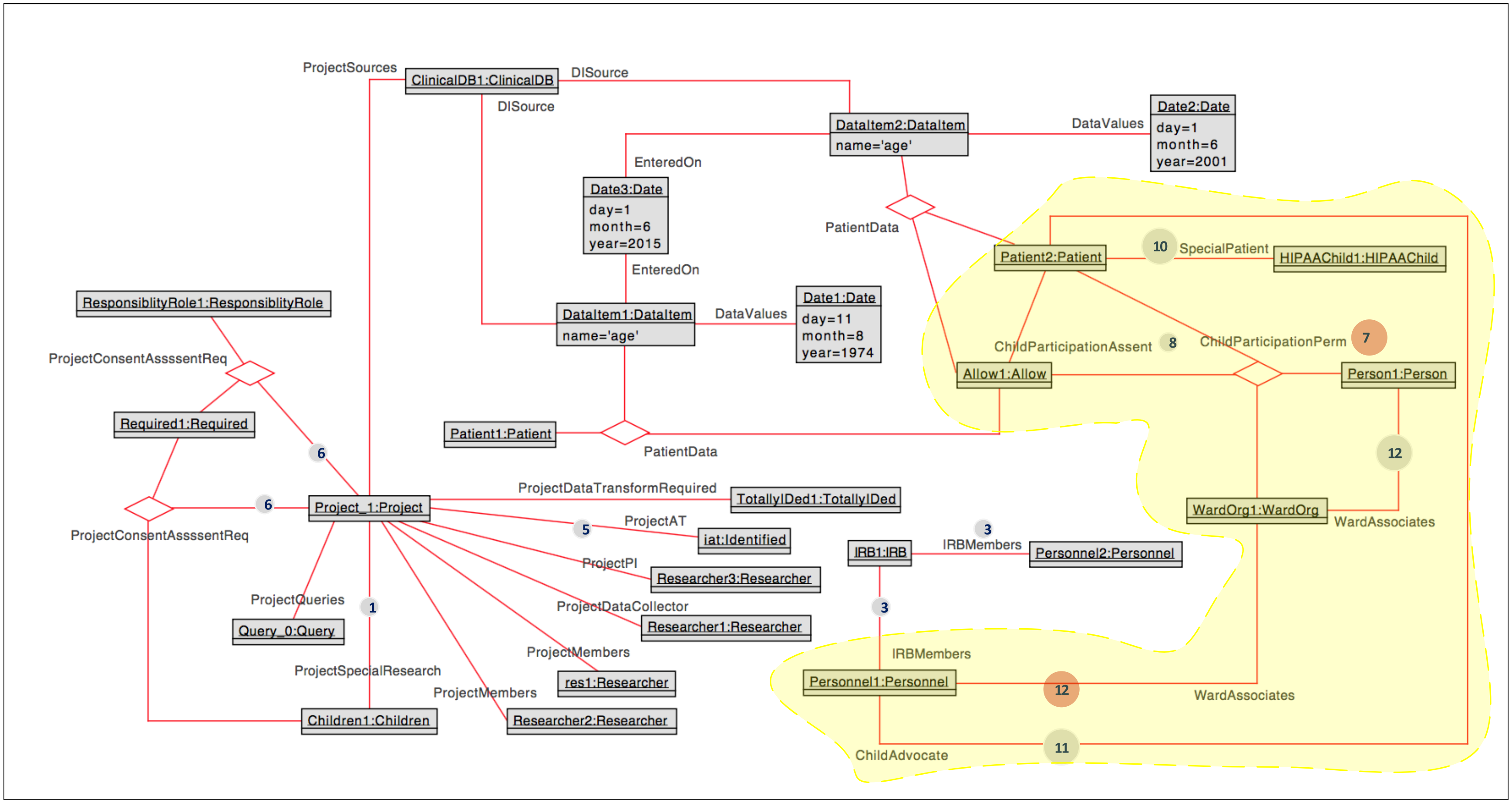


Figure 9.11: Access Denial Scenario 3: (Partial) Object Model for Slice 4 showing that *Query_0* must be denied access to *DataItem2* belonging to *Patient2* (focus is on relationships in the area highlighted in yellow). This is because the *ChildAdvocateNotAssocWithResearchOrWardOrg AccessRule* does not allow *Patient2*'s *Advocate Personnel1* (see line annotated with 11), to be associated with the institution that has responsibility for *Patient2* (see line annotated with 12 from *Personnel1* and *ChildParticipationPerm* annotated with 7). Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

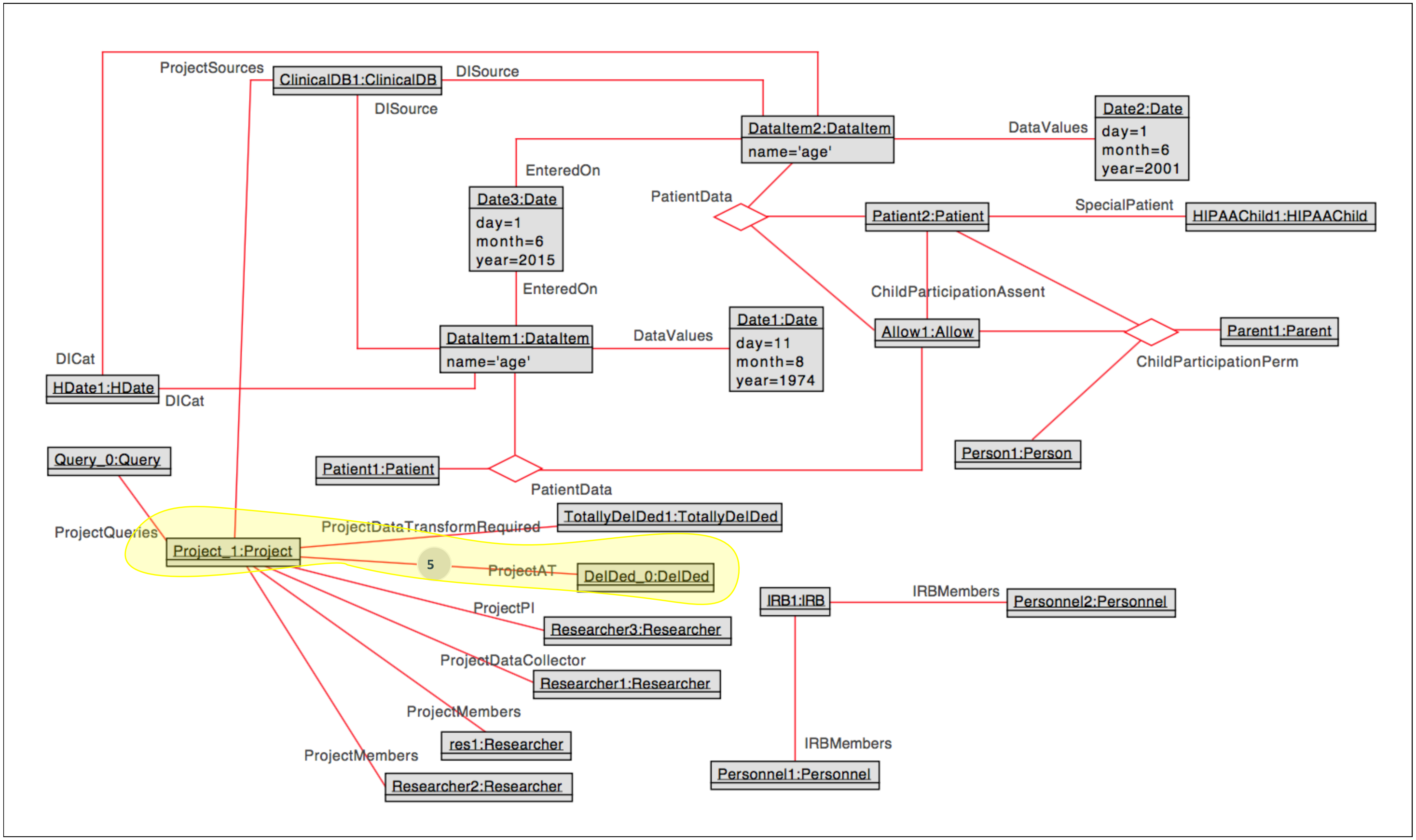


Figure 9.12: Access Denial Scenario 4: (Partial) Object Model for Slice 4 showing that *Query_0* must be denied access to *DataItem2* belonging to *Patient2* (focus is on relationships in the area highlighted in yellow). This is because the *HideSpecialPopulation AccessRule* does not allow a *DeIDed* access ticket (see line annotated with 5) to access protected populations. Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

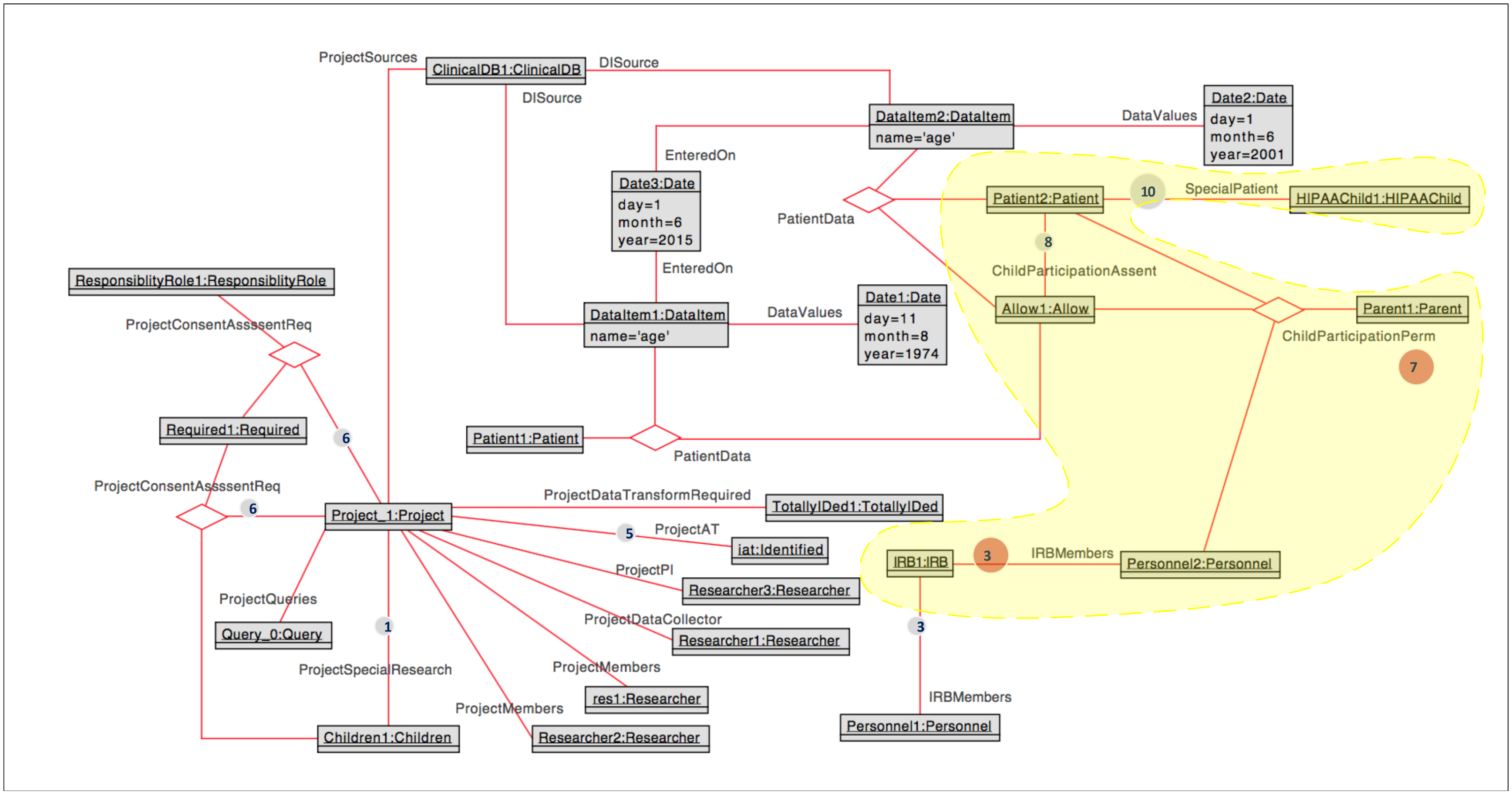


Figure 9.13: Potential Conflict of Interest: (Partial) Object Model for Slice 4 showing that the parent of *Patient2*, *Personnel2* (see association annotated with 7) is an *IRBMember* (see line annotated with 3). Focus is on relationships in the area highlighted in yellow. Numbered annotations correspond to associations so numbered in Figure 9.7 and explained in Section 9.2.2.

silent and yet our method revealed potential conflicts of interests, as we saw when a parent is an *IRBMember*. These may present areas for HIPAA to examine and improve the regulations.

We note that other conflicts of interest such as for the *NoOverlapPITeamDCIRB DecisionRule* may be refined. For example, it is usually the case when a conflict of interest arises, the *IRBMember* may abstain from contributing to a decision. In this case the system may record which *IRBMembers* contributed to the decision for the access ticket and use the *NoOverlapPITeamDCIRB DecisionRule* to ensure that there is no conflict with those contributing to a decision.

We modelled the rules only for children as a protected population, yet, as we have discussed in the sections 9.2.1 and 9.2.2 the model has been carefully presented to allow for extending it to other protected populations.

In the first instance in Section 9.2.1, we identified that, in general, model elements to support the granting of access ticket for any protected population are required for:

1. a project to indicate which special populations they require access to;
2. which decision rules applied to which special populations;
3. the IRB's decision;
4. whether an approval for the project's request for access to the special populations is required to approve the access ticket; and
5. when the IRB approves the project's request to use a specific protected population, whether the project needs to have the consent of each person in to the protected population for their data to be included in their research.

In the second instance in Section 9.2.2, we identified that, in general, model elements to support access to any protected population are required for:

1. identifying those in protected populations;
2. capturing the individual consent of those in protected populations; and
3. access rules that apply to any or specific protected populations.

Special relationships may exist for specific protected populations, e.g., children that are wards, that are not generalisable. Therefore, for each special population there may be specific model elements needed to support access to that population and these may be added to the model when such are encountered.

We noted in Chapter 8 that increasing the number of rules is another way to validate HMCA. For example, the increase of model elements for children protected population, specifically the associations among the *Person* class, may require a larger scope when analysing the current model slices (see Section 3.1 for our discussion on scope in the Alloy Analyzer) to avoid the conflicts of interest. Since analysis time may degrade for larger scopes, applying HMCA to the NJH System may use another level of slicing, i.e., slice per decision rule in order to avoid intractability. Specifying slicing criteria in different ways is already a feature of HMCA and a natural extension for dealing with intractability issues.

10. HOW TO APPLY HMCA

10.1 Introduction

HMCA is a method to analyse systems for conformance to laws and regulations, i.e., rule conformance analysis (RCA), where the details required to perform such analysis may make using current model checking tools intractable. In this dissertation we showed that analysing for conformance is possible without using large abstractions of data that would hide the details in system data models on which conformance is tested.

Applying HMCA in any domain requires that we first construct models of the system that may start out as informal models that guide the user to create more precise models of the process and data models that represent a more mature understanding of the domain. Using these models together with the requirements of the governing laws and regulations, we construct conformance rules that are used to both test and extract evidence of conformance adherence or conformance violation. After the construction, HMCA checks conformance to the rules using slicing of the models to ensure tractable analysis.

The slicing is driven by observing that:

1. separating the data for each process on a path in the process model gives better results in space and time than handling elements in memory for all the processes along a path; and

2. chaining the results from each process can be used to analyse rules that apply to the path.

Finally, when a path or data is shown to not satisfy a rule, we may highlight the entire path or isolate the process or data elements that caused the non-conformance.

This approach identifies the three phases of HMCA: 1) *construct* precise models for process, data, and conformance rules, 2) *analyse* conformance rules by slicing to decompose the analysis steps, re-composing the results in a form required by a model checker, and checking the result for conformance to the rules, and 3) providing *feedback* where rules cannot be satisfied.

Except for slicing that has been automated, our application of HMCA to the NJH system has been a manual process. The purpose of this chapter is to describe how a user may go about applying HMCA. We start by looking at HMCA in general by outlining its prerequisites in Section 10.2. Next, we outline for each phase 1) the prerequisites, 2) the steps to follow, 3) a discussion highlighting where the effort may be purely manual or can be automated, and 4) any requirements for tool support or applicable tools. We outline these in sections 10.3 through 10.5.

We note that this chapter is not meant to explain our theoretical proposal for HMCA. Such treatment may be found in Chapter 6 and should be used either as a prerequisite or co-requisite to this chapter.

10.2 Overall Prerequisites for Applying HMCA

The prerequisite for the general application of HMCA is a good understanding of model checking techniques especially as explained in the first four chapters of Baier and Katoen[14]. The focus should be on:

1. understanding why model checking may give intractable results - this will help the user to determine whether HMCA is a solution for RCA in their application domain; and
2. how to use and interpret:
   (a) a program graph (PG) as a model of the operations and data under analysis;
   (b) non-deterministic finite automata (NFA) as representations of rules to be analysed; and
   (c) a transition-system (TS) as evidence of actual operations and data states in the PG.

Each phase will have additional pre-requisites, and we outline them in the applicable subsections.

10.3 Construction Phase

The models in the construction phase may be categorised into 3 categories: process models, data model, and rule representation. The process models are activity models and entity views. The data model is the class model. The rule representation uses non-deterministic finite automata (NFA).

10.3.1 Prerequisites

The prerequisites for the construction phase of HMCA include a good understanding of:

1. UML models, specifically activity, class, and state machine models.

2. how to use OCL specifications to augment a class model with additional constraints, operation specifications, and queries;

3. how the semantics of a state machine may allow it to be linked to an activity model, i.e., how each operation in the former may be linked to a segment of the latter;

4. how the semantics of a state machine may allow it to be linked to a class model, i.e., how each abstract state in the former may be mapped to a concrete state that is a segment of the latter; and

5. NFAs, specifically how to identify and use accepting states as evidence of non-conformance.

10.3.2 Steps

10.3.2.1 Step 1: Construct UML Activity Model

A UML activity model is the beginning process model used in HMCA. From it we gain understanding of the activities that are important in the domain and how each activity impacts other activities. Of note in creating the activity model is that we must ensure that all possible values for decision nodes are modelled. This allows us to gain full understanding of all the possible paths in the system, we call this a *completeness* requirement.

We also note that an activity model may be large and complex, so we may construct it at a high level of abstraction and allow for activities to have nested activity models of the details of its internal flows. This analysis may continue for many levels of nesting. Whether or not this nesting is used, all the activity models and their associated elements have visibility within HMCA and can be linked to other models.

10.3.2.2 Step 2: Construct UML Class Model

A UML class model is the data model used in HMCA. In it we provide abstractions for the data that is required to understand the domain. In addition to classes, associations among classes, and the multiplicity constraints on the associations, we use OCL to add additional constraints not specifiable using the associations alone. The level of detail required in the class model is that of a design-level class model that includes operations with their pre-and post conditions specified using OCL. Constructing the class model may be iterative, i.e., we may return to update the class model after or during any of the steps in the construction phase, as we consider the details needed to support the activities and decisions in the activity and other models.

10.3.2.3 Step 3: Construct Individual Entity Views

We use UML state machine models to construct the entity views. Recall that an entity view represents how an entity interacts with the system and does so using a subset of the activities in the activity model. We therefore construct the state machine for an entity by identifying its:

1. abstract states and operations;
2. start and final states; and
3. adding edges among the states that are labelled with guards and operations that support advancing to the next state.

The *completeness* requirement mentioned for activity models in Section 10.3.2.1 also applies to state machines. Completeness ensures that an entity can move to the final state in a state machine without being permanently held up in an intermediate state. Alternatively, fulfilling the completeness requirement may mean we denote states as a final state where the values for variables in the guards exiting the state do not contain all the possible values that may be encountered. At this stage we have an *unlinked individual entity view*.

Since the operations and states we mention here are abstractions for segments in the activity and class models, constructing the entity views also involves providing traceability between the entity views and these models such that:

1. each operation, op_i, is linked to:
   (a) an activity model segment, am_{op}, which represents the concrete part of the system that implements it; and
   (b) a class model segment, cm_{op}, which contains the elements included in its pre- and post conditions.
2. states are linked to:

(a) a class model segment, cm_s; and

(b) activity model segments am_s where it is used or decided;

3. variables used in the guards are linked to a concrete representation, cm_v, which is a segment of the class model; in addition, we specify how to extract the value of the variables from the cm_v.

The same name for an operation, state, or variable and its associated values used in more than one entity view represents the same element, therefore, once we link an item in one entity view, it is also linked to the other entity views in which it is mentioned. We call the entity view that now has traceability to the activity and data models a *linked individual entity view*.

10.3.2.4 Step 4: Construct NFA Rules

Each conformance rule is represented as a NFA. The NFA uses the operations and states from the *individual entity views* created in Section 10.3.2.3 to specify conditions for advancing through the states. The careful construction of the rule means that we must:

1. identify accepting states; and

2. ensure that the condition, constructed using operations and states from the individual entity views leading to the accepting state, cannot also lead to non-accepting states.

10.3.2.5 Step 5: Generate RSEV and MRSEV

The final step in the construction phase is to create rule-specific entity views. We will create both a simple (more abstract) rule specific entity view (RSEV) and mapped (more concrete) rule-specific entity view, (MRSEV) for each rule.. They are generated by:

1. identifying the individual entity views created Section 10.3.2.3 that are required to check each rule; and

2. composing these entity views into a single rule-specific entity view.

From Chapter 6, recall that this composing relies on the individual entity views having common edges, i.e., when edges are labelled with the same operation, we may separately combine all the guards and next states using the logical *or* operator to create a single guard and a single state. The RSEV is created from the unlinked individual entity views, and the MRSEV is created from the linked individual entity views.

This makes the RSEV a more abstract representation that may be useful for sharing information with non-technical users. In model checking terms, the MSREV is the program graph we will use in analysis. We create traceability between the rules and their associated individual entity views, RSEV, and MRSEV by creating links among them. We note that a rule-specific entity view may be linked to more than one rule. Of course, the linking of the individual entity views to the activity and class model segments also achieves the linking of the the rule-specific entity views to these models as well.

10.3.3 Automation and Tool Support

The construction phase is mostly manual, yet we require a workbench where all the models can be supported in the same tool. While tools exists to create one or more of the UML models used in HMCA (by the same tool), no such tool exists that support our procedure to augment the activity and state machine models to maintain traceability among the models. We have therefore identified the requirements for tool support in the construction phase of HMCA below:

1. *graphing functionality*: since the models used are essentially graphs, we need functionality such as those provided by the Eclipse Modelling Framework to create and maintain these graphs;

2. *OCL language support*: we may use the functionality provided by the USE tool or an alternate way to include OCL specifications in the class model;

3. *extracting linked model segments*: while linking the models as described in the steps of the construction phase in Section 10.3.2 is a manual process, the extraction of the applicable model segment may be automated.

10.4 Analysis Phase

10.4.1 Prerequisites

The prerequisites for the analysis phase of HMCA include a good understanding of:

1. *slicing* as a technique to decompose specifications into smaller pieces in a bid to speed-up analysis; in the context of HMCA the benefit of slicing is to eliminate intractable analysis in model checking;

2. the similarities in the semantics of UML class models and Alloy models that allow the former to be represented as the latter;

3. the Alloy language and the Alloy Analyzer for writing and executing queries on specifications; and

4. model checking: specifically program graphs, using NFAs, know how program graphs are *unfolded* into a transition system, and how to check the satisfaction of an NFA on a transition system.

10.4.2 Steps

10.4.2.1 Step 1: Model Slicing

The first step in the analysis phase is to perform slicing. Recall from Section 6.3 that slicing is used to obtain tractable analysis in HMCA. A slice is created based on operations. Slicing is performed on the class model. Therefore, the slicing criteria involves copying all the elements from the class model that an operation needs into a new class model slice. For HMCA the elements, cm_i, needed for each op_i are those in:

1. cm_{op}, for its pre- and post conditions as discussed in Section 10.3.2.3;

2. all the cm_v's, for all the variables included in an operation's guards on all the edges where the operation is used as discussed in Section 10.3.2.3; and

3. all the cm_s's, for all next states that can be entered as discussed in Section 10.3.2.3.

Each cm_i is a class model segment that is transformed into an equivalent Alloy model, aa_i. This equivalence excludes the additional constraints imposed by all the OCL constraints and/or some multiplicity constraints such as those with specific numerical bounds beyond using 0..1, 1, $*$, or 1..$*$. These additional constraints must be added manually to the Alloy model, and this is done in the next step. We also create links among each operation, cm_i, and aa_i.

10.4.2.2 Step 2: Alloy Specification and Analysis

We add to the aa_i:

1. constraints to generate well-formed instances;
2. operation pre- and postconditions (for the operation that the slice represents); and
3. queries that extract the final states of an operation when the operation specification executes.

While we may not need to say much about the first two items, it is important to elaborate more on Item 3. In order to determine the possible and actual final states of an operation we must add Alloy *predicates* and *assertions* to the Alloy model. We are trying to determine which next states of an operation are possible, and we must do this for both those that would cause any applicable conformance rule to enter accepting and non-accepting states. Applicable rules are those rule NFAs that use this operation.

Predicates may be used to query for non-accepting states, i.e., an instance returned shows that the state can be reached. We must do this for all the ways an accepting state is possible. For example if the clause

$$a \ \vee \ (b \wedge c)$$

is the condition for a non-accepting state, then we must ensure that we can generate an instance for each way that the clause can return true.

While we may also use predicates to query for accepting states, it is best to use an assertion. Assertions are used to tell us whether certain conditions are ever possible, i.e., Alloy produces a counterexample if the conditions are possible, and no counterexample if they are not. In terms of the above clause, Alloy returns a counterexample if it is possible to for the clause to return false.

Alloy generated instances from predicates, and counterexamples from assertions, serve as the evidence of states occurring. Therefore we must link a state to a predicate or assertion with the understanding that an instance from the predicate indicates that it is possible, and no counterexample from the assertion indicates it is not possible. In this way we are able to extract from the Alloy specification the final states for an operation.

10.4.2.3 Step 3: Generating the TS

Since we now know the final states for an operation, we may use these final states to unfold the MSREV into a transition system. This unfolding is a model checking algorithm that gives the concrete execution of the MSREV (the program graph). It therefore contains only the reachable states for the possibilities presented in the MSREV. We link the transition system created to its MSREV. We note the final states for an operation may apply to more than one MSREVs, and it is possible in HMCA to have partial unfolding of these until each operation is analysed. In this way, we may analyse only the operations contained in a single MSREV, and show conformance to its associated rules in a stepwise or iterative manner.

10.4.2.4 Step 4: Check Conformance Rule

An NFA captures the conformance rule in such a way that it is used to detect if any of its accepting states are present in the transition system. Essentially, it specifies a pattern that is matched against a transition system. The pattern matching algorithm starts at the first state in the transition system and checks if the pattern presented in the NFA is able to reach its accepting state. This is how HMCA checks for conformance. We are guaranteed that if the transition system shows a path to the accepting state it will be found. If any such path exists, the conformance check returns that the transition system shows rule non-conformance, otherwise rule conformance is confirmed. Checking conformance is halted when the first accepting state is encountered and HMCA moves to its feedback stage.

10.4.3 Automation and Tool Support

Most of the complexity in HMCA is in the processing required in the analysis phase. While we have done the analysis manually, we can achieve automation for the tasks that, given certain inputs, can execute without additional intervention from the user.

10.4.3.1 Manual Tasks

The manual tasks in this phase are to provide:

1. *slicing criteria*;
2. *additional formal specifications* in the Alloy model; and
3. linking of predicates from the Alloy model to non-accepting states in applicable rules, and assertions to accepting states.

10.4.3.2 Automated Tasks

Automation can be realised in:

1. *slicing* to:
   (a) extract a class model slice, cm_i, for each operation, op_i, in accordance to the slicing criteria determined in Section 10.4.3.1;
   (b) link the each cm_i with its associated op_i;
   (c) transform each cm_i into an equivalent Alloy specification, aa_i.
2. *analyse* each aa_i to *extract* its final states:
   (a) use the Alloy Analyzer to determine the final states possible in each slice; and
   (b) since the Alloy Analyzer is a separate tool, we must be able to import the final states of each operation back into a workbench such as one discussed in Section 10.3.3 in order to construct the transition system.
3. *construct* the transition system: organise the final states into a transition system; and
4. *check* the conformance rule: determine whether the accepting states of the NFA are present in the transition system or not.

Our contributions are *slicing* and *extracting* the final states. We note that:

1. our implementation for *slicing* using operations as the *slicing criteria* has been developed for HMCA in the *Eclipse* environment;

2. the writing and executing of Alloy specifications is also supported in the *Eclipse* environment;

3. the algorithms of the other tasks, i.e., *constructing* the transition system and *checking* conformance, may also be developed in the *Eclipse* environment either as a new implementation or relying on libraries from known model checking tools.

10.5 Feedback Phase

10.5.1 Prerequisites

The prerequisites for the feedback phase of HMCA include a good understanding of:

1. the similarities of the semantics between UML class models and Alloy models that allow an instance (or counterexample) in the latter to be represented as an object model that is an instance of the former; and

2. the USE tool with its associated SOIL and ASSL languages for specifying class models and generating object models respectively.

10.5.2 Steps

10.5.2.1 Step 1: Extract Alloy Counterexample

Since we know the point in the transition system where the non-conformance occurs and the aa_i where non-conformance occurs, we may extract the counterexample, aac_i. We save the aac_i to an XML representation using the functionality provided in the Alloy Analyzer.

10.5.2.2 Step 2: Generate UML Object Models

Recall that in the analysis phase we generated an aa_i from each cm_i. We use this cm_i to guide the creation of an UML object model, omc_i, from the aac_i. This creation relies on the correspondence between the semantics of Alloy and class models that allows an instance in the former to be transformed into an object model of the latter, and vice versa. We note that we will have a one-to-one mapping for the elements in the aac_i to the elements in its corresponding omc_i for both the identifier, attribute values, and type. Since we have not offered a proof that the aa_i is equivalent to its associated cm_i, it is important to have an extra step to ensure that the omc_i satisfies its associated cm_i. If the omc_i cannot satisfy the cm_i, we know that either the elements and/or constraints in the aa_i or the cm_i are incorrect and this must be addressed before continuing.

10.5.2.3 Step 3: On-Demand Feedback

We implemented HMCA to provide feedback to the user in an on-demand fashion. The user may request to see a progression of omc_is that led to the non-conformance. For example, if the non-conformance occurred in slice cm_i, from the MSREV we can know the trace of its previous class model slices that led to the non-conformance observed in cm_i. This (reverse) trace is the sequence:

$$< ..., cm_{i-2}, cm_{i-1}, cm_i >$$

where each class model previous to cm_i is called a cm_j. We generate an object model, omc_j, that satisfies each cm_j, starting from $j-1$, in the trace as the user requests. Each omc_j must contain the overlapping elements from its (immediate) next omc_{j+1} in the above trace.

10.5.2.4 Step 4: Update Models (and Re-Analyse)

A counterexample occurring in a particular state in the transitions system may be a symptom of a fault that occurs in and is carried over from a previous state. Viewing the object models helps the user to identify where the fault lies: by identifying a problem in an omc_i, the links maintained in HMCA give the associated aa_i, cm_i, op_i, am_{op}, and entity views (since we know the rule being analysed). Understanding what changes are required in the models to show conformance to a rule is the job of the user/domain expert. If any changes are made, HMCA should be used to re-analyse the conformance rule.

10.5.3 Automation and Tool Support

10.5.3.1 Automated Tasks

Automation supports the following tasks to:

1. *extract* the counter-example into an XML representation;
2. *transform* the aac_i to an object model, omc_i: we use the ASSL and SOIL languages provided in the USE tool to drive the construction of the object models (see Section 10.5.3.2 for more details) and once these are created they may be reused; and
3. *generate* additional object models: we also use the languages in the USE tool to construct these additional object models (see Section 10.5.3.2 for more details) and once these are created they may be reused.

While we have used manual steps to convert the aac_i to the omc_i, we have implemented procedures to generate additional object models using the languages mentionned. We note that the *Eclipse* environment provides integration of the functionality from both the Alloy Analyzer and the USE tools to accomplish these tasks.

10.5.3.2 Manual Tasks

In addition to updating models as discussed in Section 10.5.2.4, the major manual task is the implementation of the algorithm to generate each omc_i. We outlined the algorithms for generating the feedback in Section 7.5.

An important guideline for generating each omc_i is to ensure that the constraints in its corresponding cm_i are satisfiable and do not disallow the adding of object and/or links. The algorithms may need extra tweaking that may not be generalisable, but instead depend on the elements and multiplicity constraints in each cm_i. One strategy is to add elements and the constraints that restrict those elements incrementally to the omc_i, checking satisfiability of its associated cm_i with each addition.

For example, using the ASSL language provided in the USE tool to generate the omc_i, constraints imposed by multiplicities must be satisfied for adding objects and links among them; if constraints are not satisfied, adding these elements is disallowed. This is because ASSL commands search for a configuration of objects and links to create that satisfy the constraints. In contrast, using the SOIL language (also provided for generating object models in the USE tool) does not disallow objects and links that do not satisfy the multiplicity constraints, but this may result in an omc_i that does not satisfy its corresponding cm_i because multiplicity constraints are violated. However, using SOIL is ideal when converting the initial aac_i to an omc_i because of the one-to-one correspondence between the elements in the models.

In some cases, it may be that the constraints imposed by the multiplicities do not allow for any algorithm to generate an omc_i that satisfies its corresponding cm_i. If this occurs, the only solution is to relax the multiplicity on the association end in the system class model (constructed in Section 10.3.2.2) such that we use the most generous multiplicity constraint, i.e., $*$, and write OCL constraints to enforce the desired multiplicity. In any of these scenarios, the USE tool allows scrips that can load class models, call ASSL procedures, execute SOIL commands, load/unload constraints, and check constraints as the user desires.

11. INSIGHTS FROM APPLYING HMCA IN THE NJH SYSTEM

In this chapter we offer a review of insights that may be helpful when applying HMCA to other application domains, tools, and complexity management.

11.1 Impact of New Information on Previously Defined Rules

When we add new operations and states to our models, it is important to know whether these new elements can impact previously defined rules. For example, when we included information about the *Identified* access ticket with its two types of required data transformation in Section 8.2, the rule for the *DeIDed* access ticket was impacted and this required that we update all the models to account for this new information. The lesson here is that our specifications may be weakened if we do not consider how new information affects what we have previously shown to be correct.

11.2 Managing Specification Size Complexity

Our experience has shown that managing the Alloy specifications for each slice and maintaining consistency across specifications is challenging because the specifications themselves may be many pages long and contain many overlapping elements. For the latter, many mistakes may be introduced because of the need to repeat certain model elements in different slices. Therefore, we suggest a continuous refactoring of the specifications to use the capabilities of both the Alloy Analyzer and the USE tools to first define specifications incrementally and then to include/import/add them to the specifications for the current slice. For example, in the Alloy Analyzer, when two slices overlap, we may extract the overlapping elements into a separate file and use the *open* command to add them to the specifications for each slice. This functionality offers a kind of *encapsulation* to manage the complexity of specifications. The *open* command as described is also included in the USE tool.

11.3 Understanding Tool Nuances: Translating Alloy Specifications into OCL Specifications

The analyst must be aware of the different semantics of each language. These semantics guide what abstractions are made and how to understand them in the chosen languages. We discuss three such areas

Listing 11.1: Defining *DataAccessAgreement* in Alloy

```
abstract sig DataSource{}
sig Project extends DataSource{}
sig NJH {
  projects: set Project,
  ...
  /* p1->p2 means p1 gives p2 access to data produced by p1 */
  dataAccessAgreement: projects -> projects,
  ... }
```

Listing 11.2: Defining *DataAccessAgreement* in the USE for OCL

```
abstract class DataSource end
class Project < DataSource end
association DataAccessAgreement between
    Project[*] role owner
    Project[*] role user
end
```

for understanding: 1) closures, 2) intra-associations, and 3) multiplicities on ternary relations for the specification languages used in HMCA In our discussions we will use Figure 11.1, a previous (and now outdated) class model for the NJH system.

11.3.1 Reasoning About Closures

Associations where both the source and the destination are the same class, require that we compute the association closure to reason about how a class instance relates to itself and to other instances of that class. For example, let's take the *DataAccessAgreement* annotated with A in Figure 11.1. In Alloy this association is defined as a binary relation and we show this in Listing 11.1. In OCL this is similarly defined in Listing 11.2 using the syntax of the UML Specification Environment (USE) tool. So far we have not encountered much difference in the specification languages.

Since we know that no project requires a data access agreement with itself, we add a constraint to ensure that a well-formed model does not contain these self relationships in the *DataAccessAgreement* association. In Alloy, this is defined in Listing 11.3 to say that when we compute the closure of the relation, it is irreflexive. The *irreflexive* definition is shown in Listing 11.4 and is a part of modules supplied with the Alloy Analyzer.

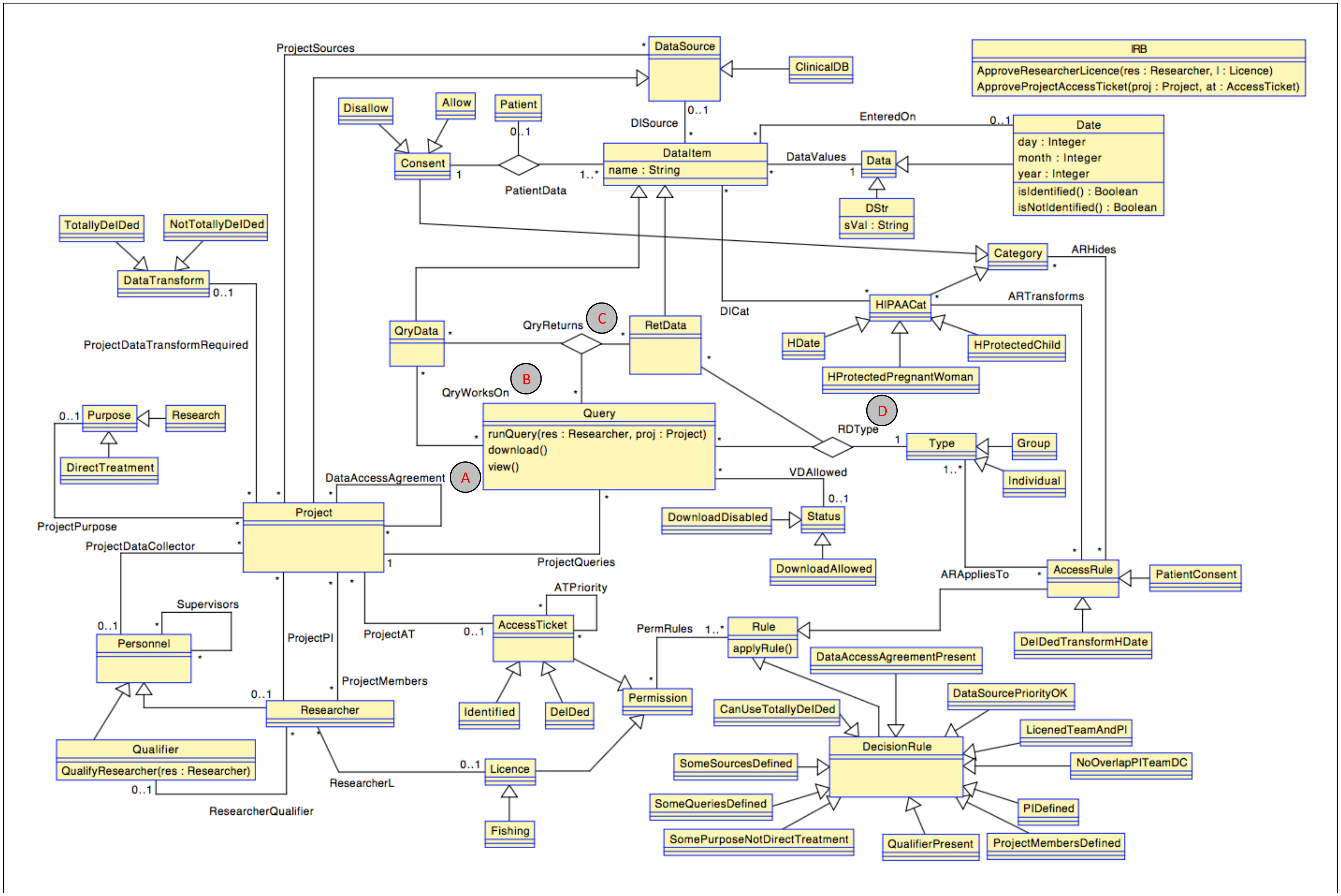


Figure 11.1: Class Model for discussing tool nuances

Listing 11.3: Defining Constraint for *DataAccessAgreement* in Alloy

```
sig NJH{...} {
   ...
   /* no project has a data access agreement with itself */
   irreflexive[^dataAccessAgreement] }
```

Listing 11.4: Defining Irreflexive Binary Relations in Alloy

```
/** r is irreflexive */
pred irreflexive [r: univ -> univ] {
   /**
   iden contains all reflexive binary associations for the signatures in the model
   & is set intersection */
   no iden & r }
```

In USE, the definition of this constraint is defined differently, since we must navigate the relationship to define its closure. Recall that the roles, i.e., each association end, in the *DataAccessAgreement* were named in the OCL definition in Listing 11.2. So, we start at the *owner* association-end, calculate its closure with (and by) navigating to the *user* association-end, and specify that this closure does not contain the *owner*, i.e., no self associations. We show this definition in OCL in Listing 11.5.

A comparison with defining the constraint first in Alloy and then in OCL using USE is that:

1. in Alloy we do not need to use navigation to reason about the contents of the association as Alloy treats the association as a set of 2-tuples and can apply set/relational/functional algebra to reason about it; this is called set semantics; and

2. translating this constraint to OCL was not as straightforward due to OCL semantics requiring navigation to compute the contents of the association; this is called navigation semantics.

This difference posed a greater challenge when dealing with constraints among associations, discussed below.

Listing 11.5: Defining Constraint for *DataAccessAgreement* in OCL

```
context Project
inv invDataAccessAggreement:
    owner->closure(user)->excludesAll(owner)
```

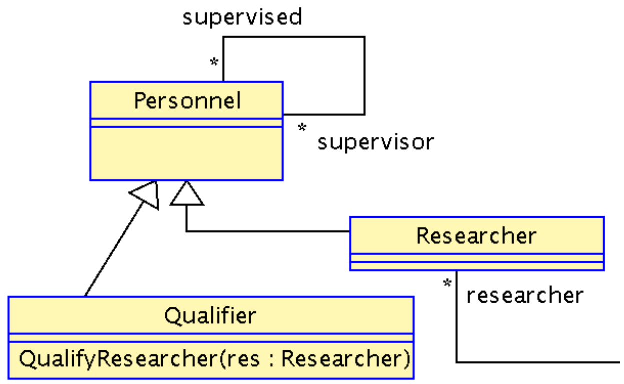


Figure 11.2: Supervisors Association in S_3

Without sufficient documentation it's hard to determine the correct usage for a predefined operation. The OCL operator *closure* computes the transitive closure of a binary association. To understand the challenge, consider the *Supervisor* association shown in Figure 11.2. In order to say that this association should be *acyclic*, a common mistake is to say (in OCL) that:

$$supervised.closure(supervisor) -> excludes(self)$$

However, on closer inspection, this is incorrect because it does allow loops. In fact, the statement can never be true because the closure (always) include *self* because the navigation to check the property starts and ends at the same place. This mistake may be made because the modeler thinks that both ends of the association need to be traversed and hence include, both association ends when writing the invariant.

As a (more concrete) example, consider:

$$Personnel = \{p_1, p_2, p_3\},$$

and

$$Supervisors = \{(p_1, p_2), (p_2, p_3)\}$$

then while both p_1 and p_2 have the *supervisor* role and both p_2 and p_3 have the *supervised* role, consideration should be given to whether p_2 that has both roles, could have a cycle. In order to get to p_2 we must navigate to the *supervised* association end and check if p_2 could supervise themselves through the transitive closure of other *supervised* traversable from p_2.

The corrected invariant is:

$$supervised.closure(supervised) -> excludes(self)$$

Alternatively, using an equivalent argument as given above for traversing the *supervised* association end, the invariant may be expressed using the *supervisor* association end:

$$supervisor.closure(supervisor)- > excludes(self)$$

Both forms are equivalent. Therefore, the *closure* must traverse along the same association end to correctly specify the acyclic invariant.

11.3.2 Intra Association Constraints

Typically, when classes are involved in more than one association, there are constraints that affect how an instance of the class in one association relates to the same instance of the class in another association. For example, let's examine the *QryWorksOn* (B), *QryReturns*, and *RDType* associations identified by *B*, *C*, and *D* respectively in Figure 11.1. *QryWorksOn* is needed to identify which *DataItem*s are used in a *Query*. Since not all instances of *DataItem* that a query works on are returned, *QryReturns* (C) shows which *DataItem* instances from a *Query* are actually used to derive data returned by the query. Further, *QryReturns* is used to show that some *DataItem* objects returned may be transformed, i.e. *QryData* and *RetData* are different with respect to their associated *Data*. In order to show conformance later on, it is important to link in *QryReturns* each *RetData* (r_i) in a *Query* (q) with the set of *QryData* (qd_i's) from which it was derived. *RDType* (D) is needed to state whether each *RetData* returned by a query is computed from an *Individual DataItem* or a from *Group* of *DataItem* because different conformance rules may apply to each type. Implicit in the multiplicities in *QryReturns* and *RDType* is that both *QryData* and *RetData* instances could be associated with more than one query.

Here, three constraints are important:

1. $(q, qd_i, r_1) \in QryReturns \;\rightarrow\; (q, qd_i) \in QryWorksOn$;

2. every (q, r_1) pair found in $QryReturns$ is also in $RDType$; and

3. if r_1 is linked to several qd_i's for the same q in *QryReturns* then

   $(q, r_1, Group) \in RDType$

   else

   $(q, r_1, Individual) \in RDType$

Listing 11.6: Defining *QryReturns* and *QryWorksOn* in Alloy

```
sig NJH {
   ...,
   dataItems: set DataItem,
   queries: set Query,
   types: set Type,
   ...
   /* a query can work on any kind of data item */
   qryReturns: queries -> dataItems -> dataItems,

   /* return data type, has 0or 1type */
   RDType: queries -> retItems -> lone types,
   ... }
```

Listing 11.7: Defining Constraint for Relationship between *QryReturns* and *QryWorksOn* in Alloy

```
sig NJH{...} {
   all
      q: queries,
      r: retItems |
   let
      /* QryData linked to r */
      qrq = (r.(q.(qryReturns))) {

   /* individual type */
   some q -> r -> Individual & njh.RDType iff
      #qrq = 1

   /* group type */
    some q -> r -> Group & njh.RDType iff
      #qrq > 1 } } }
```

The first two constraints are relatively easy to write for both Alloy and OCL. Therefore, our focus is on the third constraint. We'll hereafter refer to this constraint as c_3. For Alloy we show the definition of the associations in Listing 11.6 and c_3 in Listing 11.7. In Listing 11.7 *qrq* is computed for each (q, r) pair. We ensure that if $\#qrq = 1$ then the correct *Type* corresponding to the (q, r) pair in *RDType* is *Individual* and if $\#qrq > 1$ then the correct *Type* for the pair is *Group*.

In OCL defining c_3 is not as straightforward as in Alloy. For example, given:

1. the definition of the associations in Listing 11.8;

2. with respect to r_1, *QryReturns* contains

$$\{(q_1, qd_1, r_1), (q_2, qd_1, r_1), (q_2, qd_2, r_1), (q_3, qd_3, r_1)\}; \text{ and}$$

Listing 11.8: Definition of Associations for *QryReturns*, *QryWorksOn* and *RDType* in USE

```
association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association QryWorksOn between
    Query[*] role query
    QryData[*] role qryData
end

association RDType between
    Query[*] role rd_qry
    RetData[*] role rd_data
    Type[1] role type
end
```

3. $QryWorksOn = \{(q_1, qd_1), (q_1, qd_2), (q_2, qd_1), (q_2, qd_2), (q_3, qd_3)\}$

c_3 should ensure that for r_1, *RDType* contains :

$$\{(q_1, r_1, Individual), (q_2, r_1, Group), (q_3, r_1, Individual)\}$$

However it is impossible to specify c_3 without adding another constraint to the model to specify that each r_i is returned by only one query. We give an explanation in Section 11.3.2.1 and the reworked specification for c_3 in Section 11.3.2.2.

11.3.2.1 Why c_3 is Difficult to Specify.

Let's propose that the constraint in Listing 11.9 correctly specifies c_3. We note that navigation semantics required us to navigate through both the *QryReturns* association to get the set of *RetData* to constrain and the *RDType* association to constrain the same set of *RetData*'s corresponding *Type*. If instead, we navigated to the *Type* class by going through the *rData* association-end and then to the *type* association-end, we get a *Bag* of *Type* instead of a single *Type*. This is because each instance of *RetData* may be returned by more than one query, and though the same, may be computed differently.

The next step is to use the intersection of both the *qData* and the *qryData* to get to the set *QryData* that *RetData* derives from. However, with the assignments given to *QryReturns* and *QryWorksOn* above, this specification for c_3 computes that for q_1, the qd_i's that the r_1 is derived from is the set $\{qd_1, qd_2\}$ and

Listing 11.9: Incorrect Definition of Constraint between *QryReturns RDType* in OCL

```
context Query
inv invRDType:
   rd_data = rData and
      rData->forAll(r |
         /* since no iff we have to write both ways */
         ((r.qData->intersection(qryData)->size()=1 implies
            self.type->select(
               oclIsTypeOf(Individual)=true).rd_data->includes(r))
         and
         (self.type->select(
            oclIsTypeOf(Individual)=true).rd_data->includes(r) implies
               r.qData->intersection(qryData)->size()=1 ))
         and
         /* again, since no iff we have to write both ways */
         ((r.qData->intersection(qryData)->size()>1 implies
            self.type->select(
               oclIsTypeOf(Group)=true).rd_data->includes(r))
         and
         (self.type->select(
            oclIsTypeOf(Group)=true).rd_data->includes(r) implies
               r.qData->intersection(qryData)->size()>1 ))
      )
```

would incorrectly enforce $(q_1, r_1, Group)$ in *RDType*! However this is different from what *QryReturns* tells us, i.e., the singleton instance qd_1. If the specification is rewritten to use the set of *QryData* that q_1 used to derive all its r_i's by using

$$self.qData(intersection(qryData))$$

where $self$ refers to *Query*, the problem still exists if (q_1, qd_2, r_2) was included in *QryReturns*. We have a delima!

11.3.2.2 Making c_3 Specifiable in OCL

After the detailed examination of how to specify that *for q_1, r_1 is derived only from qd_1*, the only solution is to add that *each r_i can only be returned by one q_i*. We add this constraint in Listing 11.10. Further, *RDType* can be simplified to the specification in Listing 11.11. Finally, we restate c_3:

if r_1 is linked to several qd_i's in *QryReturns*

then

$(r_1, Group) \in RDType$

Listing 11.10: Definition of Constraint between *QryReturns RDType* in OCL

```
context RetData
inv retDataInOneQuery:
    qry->size()<=1
```

Listing 11.11: Definition of Constraint between *QryReturns RDType* in USE

```
association RDType between
   RetData[*] role rd_data
   Type[1] role type
end
```

else

$$(r_1, Individual) \in RDType$$

We show the OCL specification for the restated c_3 in Listing 11.12. The lesson when dealing with intra-associations constraints is that the comparison between Alloy and OCL requires the analyst to keep in mind that in OCL navigating through more than one association may produce a *Bag* or *Set* rather than a single instance.

11.3.2.3 Semantics and Scoping Constraints that Affected c_3

The way the association is written in Alloy helps us to use a smaller scope because each *RetData* may be assigned to more than one query. However, this way to model *QryReturns* made it difficult to specify the original c_3 in OCL. The Alloy Analyzer uses optimisation when generating instances to try to generate the minimal set possible to satisfy all the constraints specified in the model. In USE, an object model may

Listing 11.12: Definition of Constraint between *QryReturns RDType* in OCL

```
context Query
inv invRDType:
    rData->forAll(
        if qData->size()=1 then
            type->select(oclIsTypeOf(Individual)=true)->size=1
        else
            type->select(oclIsTypeOf(Group)=true)->size=1
        endif
    )
```

also be optimised in this way. However, as we have shown, additional thought is required to correctly model the same association or relationship in Alloy and class models respectively because of the semantics of each specification language.

11.3.3 Ternary Relations and Multiplicities

During the translation of Alloy to OCL, we discovered that the multiplicities in a ternary relationship in Alloy are semantically different from the interpretation in the USE tool. For example, the *RD_Type* association, shown in Figure 11.3a, has a multiplicity at the *Type* end of 1. In Alloy, this association is modelled as:

```
RDType: queries -> retItems -> one types
```

and may be interpreted as:

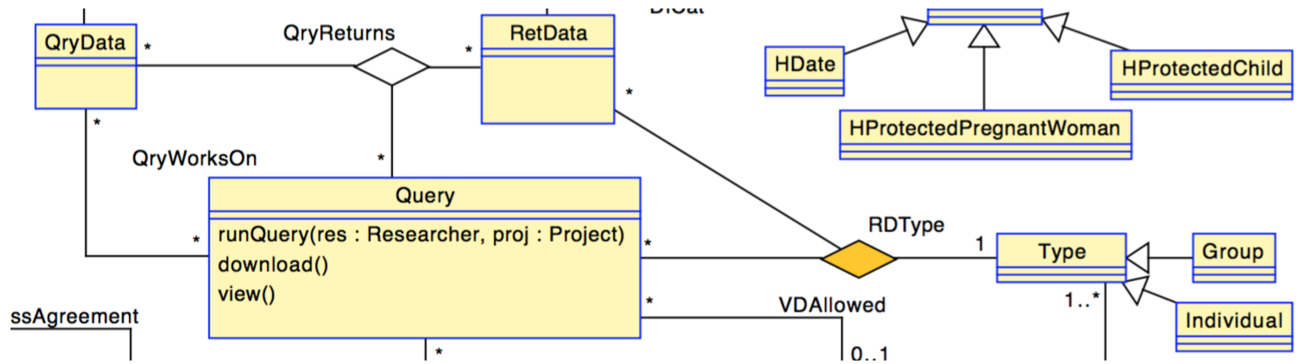
each *Query* and RetData pair is linked to exactly 1 *Type*, i.e., either a *Group* or and *Individual*.

This interpretation of the association is consistent in USE except where *Type* has subclasses. When subclasses of *Type* exists, this invariant on the multiplicity becomes, each *Query* and RetData pair has exactly 1 of each of the subclasses of *Type* (and *Type* if it is not abstract). In order to specify the originally intent, the multiplicity at the *Type* end had to be relaxed as shown in Figure 11.3b. In addition, since we intended that each *RetData* requires a *Type*, this was included as an invariant.

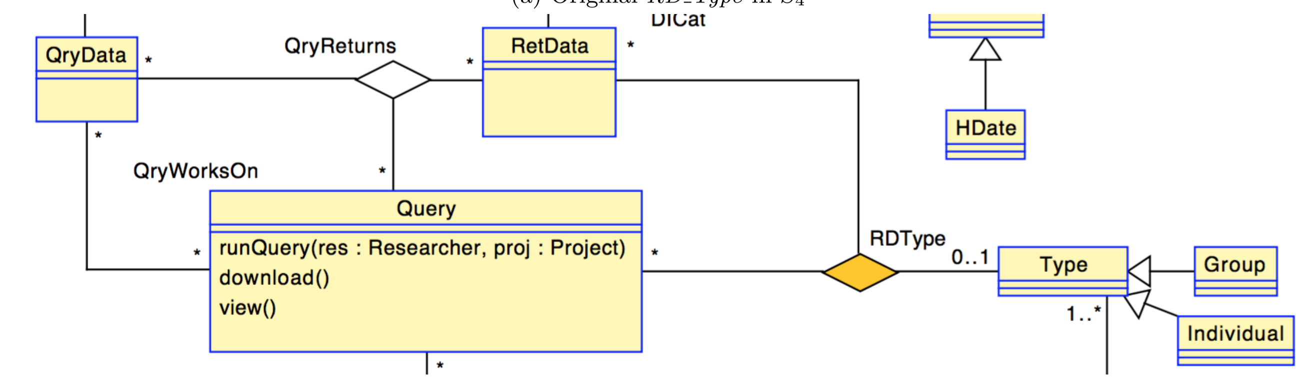
Analysis of the original specification in the *USE* tool showed the nuance in USE. This means that one has to be careful when specifying the multiplicity for associations involving more than two classes.

11.4 Summary

In this chapter we recapped some insights from applying HMCA for RCA in the NJH system. While understanding the impact of new information on previously defined rules and managing the complexities of specification size are important, the major impact was with working through the nuances of the formal specification languages. While the Alloy language and class models with OCL constraints have many similarities as specification languages and in their associated tools, their semantic differences influence how we should approach modelling activities. More information on these differences may be explored in [15].



(a) Original *RD_Type* in S_4



(b) Updated *RD_Type* in S_4

Figure 11.3: Slice: Partial slice of $S4$ highlighting the *RDType* Association.

12. CONCLUSIONS AND FUTURE DIRECTIONS

Model checking is used for RCA because it allows the exhaustive examination of system models to show conformance to rules. While the current model checking tools allow us to easily analyse process-aware rules, they have challenges when analysing data-aware rules because of a state-space explosion that may cause the analysis to be incomplete. For data-aware rules, using large abstractions ensure that the model checking tools complete their analysis. However, using large abstractions may hide the details needed to check conformance to the data-aware rules. In addition to the explosion of the state space, the current model checking tools are not suited for analysing complex data relationships. We proposed HMCA to overcome these challenges.

12.1 HMCA Contribution Conclusions

Model checking is used for RCA because it allows the exhaustive examination of system models to show conformance to rules. While the current model checking tools allow us to easily analyse process-aware rules, they have challenges when analysing data-aware rules because of a state-space explosion that may cause the analysis to be incomplete. For data-aware rules, using large abstractions ensure that the model checking tools complete their analysis. However, using large abstractions may hide the details needed to check conformance to the data-aware rules. In addition to the explosion of the state space, the current model checking tools are not suited for analysing complex data relationships. We proposed HMCA to overcome these challenges.

In response to the state-space explosion, the main contribution of HMCA is to analyse data-aware rules where current model checking tools fail. For HMCA, we show how to get results, i.e., analysis of rules can be completed when using model checking techniques to analyse data-aware rules without hiding the details in system models. Before this research, such analysis of data-aware rules was impossible at the level of details used in HMCA, yet this was important because the details are needed to show conformance to rules such as those extracted from the privacy requirements in the HIPAA regulations. We describe HMCA as *hybrid* because it allows exhaustive model-based verification/analysis within a certain scope.

Since HMCA has its underpinnings in model checking techniques, we show how HMCA:

1. *construct*s design-level abstractions of the system under analysis and how to map conformance rules to these abstractions ;
2. *decomposes the analysis* when checking each conformance rule by applying *model slicing* to produce slices of the system state that avoids encountering a state-space explosion;
3. *uses the Alloy Analyzer* to provide an exhaustive and scoped analysis of each slice; and
4. *provides on-demand and detailed feedback* from the slices where the system shows non-conformance to a rule.

In addition to providing a demonstration HMCA in the NJH system, we provided evaluations of HMCA by using the NJH system:

1. to show how HMCA can be used to detect:
   (a) common logic flaws in new conformance rules that result in non-conformance; and
   (b) underspecification of conditions in the pre- or post conditions of an operations that uncovers ways certain states are incorrectly allowed in the transition system.
2. for incorporating additional conditions that must be checked for conformance by including the privacy requirements for the children protected population.

Evaluating HMCA in these ways shows another contribution as it helps to validate that non-conformance can be found even when complex data relationships exist in the models under analysis.

We also provided a description of the steps that other users may follow to implement HMCA in other domains. Finally, we gave insights gained from our practical application of HMCA in the NJH that may be helpful, especially to draw awareness to situations where similar but differing semantics in formal specifications languages may impact specification in ways that are unexpected. Our description of steps and insights is important for HMCA to be a next step in developing tools based on model checking for RCA.

12.2 Limitations of HMCA

Factors that limit HMCA's ability to produce correct results include:

1. Having *correct models* that are a true reflection of the system under analysis and includes asking *how do we know that they satisfy the specifications*?

2. *Accurately interpreting of regulations*, such as those in HIPAA, and translating them into conformance rules.

3. *Providing the required elements in a slice.* While slicing gives us a smaller sized model and allows us to avoid a state-space explosion that does not allow analysis to complete for data-aware rules, a limiting factor for slicing is providing the correct slicing criteria. Currently we an operation's guard together with it's pre- and post conditions for this criteria. However if they are not specified correctly we may be performing analysis on a slice that has too little or too much details. In the case of the latter, we may not be analysing the correct state or have hidden paths (see Section 12.3.2).

4. The *abstractions, memory, and scope* required to perform the analysis using the Alloy Analyzer. The Alloy Analyzer become a limiting factor when the time needed to analyse each slice increases to the due to the size of the slice or the memory bounds are reached without completing the analysis because of the scope required. One of the ways to reduce the limits is to recognise that more complex rules may require the use of finer grained slices and this is translates into specifying the entity views using operations that will result in small slices

For the first two limitations, we must rely on the domain experts to confirm correct interpretation of the models.

12.3 Future directions

We outline some areas where HMCA can benefit from additional research. The areas discussed in sections 12.3.1 to 12.3.3 were first outlined as challenges to RCA in Section 1.2 and should be referenced for additional details.

12.3.1 Analysing Changed and Conflicting Rules

Changed rules can be addressed by using HMCA to re-analyse the rules. One of the ways HMCA can be used is to track the changes in rules, system conformance to the rules, and to include ways to judge the level of conformance of the system to the rules. We noted in our related work how metrics such as *weak and strong conformance* (see Section 2.1.2) are used judge the *level of conformance* in systems. These and other metrics may be used or developed in areas where conformance may be measured on different levels or systems in a particular domain are being compared.

When rules conflict, one of the ways HMCA may be used is in detecting such conflicts by identifying the conditions that make satisfying them mutually exclusive. This would be a further way to validate HMCA to be able to uncover these situations that have impossible system states. These conflicts may be deemed as an over-specification of the model.

12.3.2 Hidden path analysis

Hidden paths may exist when path possibilities are not well understood or constrained by what is specified in the process-aware rules and data-aware rules. The rules may focus on the allowed paths and how changes in the systems state are effected along the paths. In addition, the rules may restrict those path possibilities that should not be allowed. However, hidden paths in either of these categories may exist. We may discover hidden paths where the results from local analyses may be recombined to create paths not documented in the system activity diagram. Finding hidden paths are important and may be of high value because they may cause rules to be violated, or reveal that other rules are needed.

12.3.3 Alternate Rule Representations

In addition to showing how to represent rules from laws and regulations in [17, 27, 30, 65], other approaches, specifically using 1) automaton [63, 71], 2) logic [5, 66], and 3) patterns [13, 50, 86] have been used to represent conformance rules. Patterns are useful as abstractions of rule specifications. They also can be used as rule specification notations, and finally they can provide guidance to the modeller as to which elements need to be included in the specifications (i.e. specification strategies).

While we used both LTL and Dwyer's patterns (see Section 2.3.2.1) when evaluating the model checking tools and the Alloy Analyzer in Chapter 4, HMCA uses NFA to specify the rules. Dwyer's Patterns provide an alternative way to represent the conformance rules. For example, to define the *DeIDed* conformance rule we may use the *Absence* pattern to specify that the <*Viewing, Identified*> state should never be observed when a de-identified access ticket is used to view a query's results.

Since Dwyer's patterns have underpinnings in temporal logic, we may:

1. transform the pattern rule representations to linear temporal logic, and then to NFA, or
2. independent of patterns, use linear temporal logic to specify the rules, and then transform them to NFAs.

A next logical step is to prove equivalence for the (same) rule in each of the representations. This requires the use of other model checking techniques (e.g. Bisimulation [51]).

12.3.4 How much Feedback to Show

In the summary of Chapter 7 we discussed that we may identify semantics for what the feedback shown to the user should contain (see Section 7.7 for the details). Such semantics can help in designing suitable user interfaces. It requires continuous evaluation and may be specific to each domain in which HMCA is used.

13. BIBLIOGRAPHY

[1] (2015, October) The Alloy Model Analyzer, http://alloy.mit.edu/alloy/. [Online]. Available: http://alloy.mit.edu/alloy/

[2] (2015, July) HIPAA Administrative Simplification Statute and Rules. [Online]. Available: http://www.hhs.gov/ocr/privacy/hipaa/administrative/

[3] (2015, July) HIPAA Violations and Enforcement. [Online]. Available: http://www.ama-assn.org/ama/pub/physician-resources/solutions-managing-your-practice/coding-billing-insurance/hipaahealth-insurance-portability-accountability-act/hipaa-violations-enforcement.page?

[4] (2015, June) UPPAAL. [Online]. Available: http://www.uppaal.org

[5] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni, "Expressing and verifying business contracts with abductive logic programming," in *Normative Multi-agent Systems, 18.03. - 23.03.2007*, ser. Dagstuhl Seminar Proceedings, G. Boella, L. W. N. van der Torre, and H. Verhagen, Eds., vol. 07122. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/901

[6] K. Anastasakis, "A model driven approach for the automated analysis of uml class diagrams." Ph.D. dissertation, University of Birmingham, Birmingham, 2009.

[7] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software & Systems Modeling*, vol. 9, no. 1, p. 69, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10270-008-0110-3

[8] A. Andoni, D. Daniliuc, and S. Khurshid, "Evaluating the "small scope hypothesis"," MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, Tech. Rep., 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.7702&rep=rep1&type=pdf

[9] É. André, C. Choppy, and T. Noulamo, "Modelling timed concurrent systems using activity diagram patterns," in *Knowledge and Systems Engineering - Proceedings of the Sixth International Conference KSE 2014, Hanoi, Vietnam, 9-11 October 2014*, ser. Advances in Intelligent Systems and Computing, V. Nguyen, A. Le, and V. Huynh, Eds., vol. 326. Springer, 2014, pp. 339–351. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11680-8_27

[10] É. André, C. Choppy, and G. Reggio, "Activity diagrams patterns for modeling business processes," in *Software Engineering Research, Management and Applications [selected papers from the 11th International Conference on Software Engineering Research, Management and Applications, SERA 2013, Prague, Czech Republic, August 7-9, 2013].*, ser. Studies in Computational Intelligence, R. Y. Lee, Ed., vol. 496. Springer, 2013, pp. 197–213. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-00948-3_13

[11] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, "Model projection: simplifying models in response to restricting the environment," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 291–300. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985834

[12] O. Angiuli, J. Blitzstein, and J. Waldo, "How to de-identify your data," *Communications of the ACM*, vol. 58, no. 12, pp. 48–55, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2814340

[13] A. Awad and M. Weske, "Visualization of compliance violation in business process models," in *Business Process Management Workshops, BPM 2009 International Workshops, Ulm, Germany, September 7, 2009. Revised Papers*, ser. Lecture Notes in Business Information Processing, S. Rinderle-Ma, S. W. Sadiq, and F. Leymann, Eds., vol. 43. Springer, 2009, pp. 182–193. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12186-9_17

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[15] M. Balaban, P. Bennett, K. H. Doan, G. Georg, M. Gogolla, I. Khitron, and M. Kifer, "A comparison of textual modeling languages: OCL, Alloy, FOML," in *OCL@MoDELS*, 2016. [Online]. Available: http://oclworkshop.github.io/2016/papers/OCL16_paper_3.pdf

[16] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J. M. Bruel, "Formalization of the whole-part relationship in the unified modeling language," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 459–470, May 2003.

[17] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum, "Privacy and contextual integrity: Framework and applications," in *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 2006, pp. 184–198. [Online]. Available: http://dx.doi.org/10.1109/SP.2006.32

[18] D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu, "Monitoring compliance policies over incomplete and disagreeing logs," in *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds., vol. 7687. Springer, 2012, pp. 151–167. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35632-2_17

[19] D. A. Basin, F. Klaedtke, and S. Müller, "Monitoring security policies with metric first-order temporal logic," in *SACMAT 2010, 15th ACM Symposium on Access Control Models and Technologies, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*, J. B. D. Joshi and B. Carminati, Eds. ACM, 2010, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/1809842.1809849

[20] J. Becker, P. Bergener, P. Delfmann, and B. Weiß, "Modeling and checking business process compliance rules in the financial sector," in *Proceedings of the International Conference on Information Systems, ICIS 2011, Shanghai, China, December 4-7, 2011*, D. F. Galletta and T. Liang, Eds. Association for Information Systems, 2011. [Online]. Available: http://aisel.aisnet.org/icis2011/proceedings/projmanagement/12

[21] J. Becker, P. Delfmann, H.-A. Dietrich, M. Steinhorst, and M. Eggert, "Business process compliance checking – applying and evaluating a generic pattern matching approach for conceptual models in the financial sector," *Information Systems Frontiers*, pp. 1–47, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10796-014-9529-y

[22] J. Becker, P. Delfmann, S. Herwig, and L. Lis, "A generic set theory-based pattern matching approach for the analysis of conceptual models," in *Conceptual Modeling - ER 2009, 28th International Conference on Conceptual Modeling, Gramado, Brazil, November 9-12, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. H. F. Laender, S. Castano, U. Dayal, F. Casati, and J. P. M. de Oliveira, Eds., vol. 5829. Springer, 2009, pp. 41–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04840-1_6

[23] P. Bennett, W. Sun, W. Ted, G. Georg, I. Ray, and M. G. Kahn, "Analyzing regulatory conformance in medical research systems using multi-paradigm modeling," in *Joint Proceedings of the 3rd International Workshop on the Globalization Of Modeling Languages and the 9th International Workshop on Multi-Paradigm Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, GEMOC+MPM@MoDELS 2015, Ottawa, Canada, September 28, 2015.*, ser. CEUR Workshop Proceedings, B. Combemale, J. DeAntoni, J. Gray, D. Balasubramanian, B. Barroca, S. Kokaly, G. Mezei, and P. V. Gorp, Eds., vol. 1511. CEUR-WS.org, 2015, pp. 22–31. [Online]. Available: http://ceur-ws.org/Vol-1511/paper-MPM01.pdf

[24] A. Birukou, V. D'Andrea, F. Leymann, J. Serafinski, P. Silveira, S. Strauch, and M. Tluczek, "An integrated solution for runtime compliance governance in SOA," in *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, ser. Lecture Notes in Computer Science, P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, Eds., vol. 6470, 2010, pp. 122–136. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17358-5_9

[25] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux, “Modeling model slicers,” in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds., vol. 6981. Springer, 2011, pp. 62–76. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8_6

[26] G. Buchgeher and R. Weinreich, “Towards continuous reference architecture conformance analysis,” in *Software Architecture - 7th European Conference, ECSA 2013, Montpellier, France, July 1-5, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. Drira, Ed., vol. 7957. Springer, 2013, pp. 332–335. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39031-9_32

[27] O. Chowdhury, A. Gampe, J. Niu, J. von Ronne, J. Bennatt, A. Datta, L. Jia, and W. H. Winsborough, “Privacy promises that can be kept: a policy analysis method with application to the HIPAA privacy rule,” in *18th ACM Symposium on Access Control Models and Technologies, SACMAT ’13, Amsterdam, The Netherlands, June 12-14, 2013*, M. Conti, J. Vaidya, and A. Schaad, Eds. ACM, 2013, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2462410.2462423

[28] A. Cunha, A. Garis, and D. Riesco, “Translating between alloy specifications and UML class diagrams annotated with OCL,” *Software & Systems Modeling*, vol. 14, no. 1, pp. 5–25, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0353-5

[29] P. Delfmann, M. Steinhorst, H. Dietrich, and J. Becker, “The generic model query language GMQL - conceptual specification, implementation, and runtime evaluation,” *Inf. Syst.*, vol. 47, pp. 129–177, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.is.2014.06.003

[30] H. DeYoung, D. Garg, L. Jia, D. K. Kaynar, and A. Datta, “Experiences in the logical specification of the HIPAA and GLBA privacy laws,” in *Proceedings of the 2010 ACM Workshop on Privacy in the Electronic Society, WPES 2010, Chicago, Illinois, USA, October 4, 2010*, E. Al-Shaer and K. B. Frikken, Eds. ACM, 2010, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1866919.1866930

[31] L. B. R. dos Santos, V. A. de Santiago Junior, and N. L. Vijaykumar, “Transformation of UML behavioral diagrams to support software model checking,” in *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2014, Grenoble, France, 12th April 2014.*, ser. EPTCS, B. Buhnova, L. Happe, and J. Kofron, Eds., vol. 147, 2014, pp. 133–142. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.147.10

[32] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property specification patterns for finite-state verification,” in *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, M. A. Ardis and J. M. Atlee, Eds. ACM, 1998, pp. 7–15. [Online]. Available: http://doi.acm.org/10.1145/298595.298598

[33] ——, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 1999 International Conference on Software Engineering, ICSE’ 99, Los Angeles, CA, USA, May 16-22, 1999.*, B. W. Boehm, D. Garlan, and J. Kramer, Eds. ACM, 1999, pp. 411–420. [Online]. Available: http://portal.acm.org/citation.cfm?id=302405.302672

[34] Earnest and Young. (2015, November) The Volcker Rule: Covered funds, investment activity and affiliated transactions. [Online]. Available: http://www.ey.com/Publication/vwLUAssets/EY-4_steps-to-Volcker-Rule-compliance/$FILE/ey-4-steps-to-Volcker-Rule-compliance.pdf

[35] R. Eshuis, “Symbolic model checking of UML activity diagrams,” *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 1–38, January 2006.

[36] Federal Trade Commission. (2015, November) Financial institutions and customer information: Complying with the safeguards rule. [Online]. Available: https://www.ftc.gov/tips-advice/business-center/guidance/financial-institutions-customer-information-complying

[37] F. Fernandes and M. Song, “UML-Checker: An approach for verifying UML behavioral diagrams,” *JSW*, vol. 9, no. 5, pp. 1229–1236, 2014. [Online]. Available: http://dx.doi.org/10.4304/jsw.9.5.1229-1236

[38] D. Garg, L. Jia, and A. Datta, "Policy auditing over incomplete logs: theory, implementation and applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046726

[39] G. Georg, P. Bennett, and W. Sun, "Example for MODELS MPM workshop paper," June 2015, internal document.

[40] C. Giblin, S. Müller, and B. Pfitzmann, "From Regulatory Policies to Event Monitoring Rules: Towards Model-Driven Compliance Automation," IBM Research GmbH. Zurich Research Laboratory, Tech. Rep., 2006.

[41] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software & Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10270-005-0089-y

[42] H. Grönniger, D. Reiss, and B. Rumpe, "Towards a semantics of activity diagrams with semantic variation points," in *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6394. Springer, 2010, pp. 331–345. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16145-2_23

[43] O. M. Group, "OMG unified modeling language specification," Standard Released by the OMG Group, Tech. Rep., September 2013. [Online]. Available: http://www.omg.org/spec/UML/2.5/Beta2/PDF/

[44] N. J. Health, "Map of integrated bioinformation and specimen centre research support," *Internal NJH Document*.

[45] G. J. Holzmann, *The SPIN Model Checker - Primer and Reference Manual.* Addison-Wesley, 2004.

[46] V. H. Huynh and A. N. T. Le, "Process mining and security: Visualization in database intrusion detection," in *Intelligence and Security Informatics - Pacific Asia Workshop, PAISI 2012, Kuala Lumpur, Malaysia, May 29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Chau, G. A. Wang, W. T. Yue, and H. Chen, Eds., vol. 7299. Springer, 2012, pp. 81–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30428-6_7

[47] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149

[48] ——, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012.

[49] D. Jackson, I. Schechter, and I. Shlyakhter, "Alcoa: the alloy constraint analyzer," in *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, C. Ghezzi, M. Jazayeri, and A. L. Wolf, Eds. ACM, 2000, pp. 730–733. [Online]. Available: http://doi.acm.org/10.1145/337180.337616

[50] H. Jacobsen, V. Muthusamy, and G. Li, "The PADRES event processing network: Uniform querying of past and future events (das PADRES ereignisverarbeitungsnetzwerk: Einheitliche anfragen auf ereignisse der vergangenheit und zukunft)," *it - Information Technology*, vol. 51, no. 5, pp. 250–261, 2009. [Online]. Available: http://dx.doi.org/10.1524/itit.2009.0549

[51] P. Jancar, "Bisimulation equivalence of first-order grammars is Ackermann-hard," *CoRR*, vol. abs/1312.3910, 2013. [Online]. Available: http://arxiv.org/abs/1312.3910

[52] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, and P. Dadam, "On enabling data-aware compliance checking of business process models," in *Conceptual Modeling - ER 2010, 29th International Conference on Conceptual Modeling, Vancouver, BC, Canada, November 1-4, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Parsons, M. Saeki, P. Shoval, C. C. Woo, and Y. Wand, Eds., vol. 6412. Springer, 2010, pp. 332–346. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16373-9_24

[53] F. Kordon, A. Linard, M. Beccuti, D. Buchs, L. Fronc, L. Hillah, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, E. Paviot-Adet, F. Pommereau, C. Rodríguez, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf, "Model checking contest at Petri Nets, report on the 2013 edition," *CoRR*, vol. abs/1309.2485, 2013. [Online]. Available: http://arxiv.org/abs/1309.2485

[54] F. Kordon, A. Linard, D. Buchs, M. Colange, S. Evangelista, L. Fronc, L. Hillah, N. Lohmann, E. Paviot-Adet, F. Pommereau, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf, "Raw report on the model checking contest at Petri Nets 2012," *CoRR*, vol. abs/1209.2382, 2012. [Online]. Available: http://arxiv.org/abs/1209.2382

[55] F. Kordon, A. Linard, D. Buchs, M. Colange, S. Evangelista, K. Lampka, N. Lohmann, E. Paviot-Adet, Y. Thierry-Mieg, and H. Wimmel, "Report on the model checking contest at Petri Nets 2011," *Transactions on Petri Nets and Other Models of Concurrency*, vol. 6, pp. 169–196, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35179-2_8

[56] K. Lano and S. K. Rahimi, "Slicing of UML models using model transformations," in *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, ser. Lecture Notes in Computer Science, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6395. Springer, 2010, pp. 228–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16129-2_17

[57] ——, "Slicing techniques for UML models," *Journal of Object Technology*, vol. 10, pp. 11: 1–49, 2011. [Online]. Available: http://dx.doi.org/10.5381/jot.2011.10.1.a11

[58] L. T. Ly, D. Knuplesch, S. Rinderle-Ma, K. Göser, H. Pfeifer, M. Reichert, and P. Dadam, "Seaflows toolset - compliance verification made easy for process-aware information systems," in *Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers*, ser. Lecture Notes in Business Information Processing, P. Soffer and E. Proper, Eds., vol. 72. Springer, 2010, pp. 76–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17722-4_6

[59] L. T. Ly, S. Rinderle, and P. Dadam, "Semantic correctness in adaptive process management systems," in *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102. Springer, 2006, pp. 193–208. [Online]. Available: http://dx.doi.org/10.1007/11841760_14

[60] L. T. Ly, S. Rinderle-Ma, P. Dadam, and B. Pernici, "Design and verification of instantiable compliance rule graphs in process-aware information systems," in *Advanced Information Systems Engineering, Proceedings*, vol. 6051. Heidelberger Platz 3, D-14197 Berlin, Germany: Springer-Verlag Berlin, 2010, pp. 9–23.

[61] L. T. Ly, S. Rinderle-Ma, K. Göser, and P. Dadam, "On enabling integrated process compliance with semantic constraints in process management systems - requirements, challenges, solutions," *Information Systems Frontiers*, vol. 14, no. 2, pp. 195–219, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10796-009-9185-9

[62] L. T. Ly, S. Rinderle-Ma, D. Knuplesch, and P. Dadam, "Monitoring business process compliance using compliance rule graphs," in *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Meersman, T. S. Dillon, P. Herrero, A. Kumar, M. Reichert, L. Qing, B. C. Ooi, E. Damiani, D. C. Schmidt, J. White, M. Hauswirth, P. Hitzler, and M. K. Mohania, Eds., vol. 7044. Springer, 2011, pp. 82–99. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25109-2_7

[63] F. M. Maggi, M. Montali, M. Westergaard, and W. M. P. van der Aalst, "Monitoring business constraints with linear temporal logic: An approach based on colored automata," in *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, ser. Lecture Notes in Computer Science, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., vol. 6896. Springer, 2011, pp. 132–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23059-2_13

[64] S. Maoz, J. O. Ringert, and B. Rumpe, "Semantically configurable consistency analysis for class and object diagrams," *CoRR*, vol. abs/1409.2313, 2014. [Online]. Available: http://arxiv.org/abs/1409.2313

[65] M. J. May, C. A. Gunter, and I. Lee, "Privacy APIs: Access control techniques to analyze and verify legal privacy policies," in *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, pp. 85–97. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CSFW.2006.24

[66] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst, "Monitoring business constraints with the event calculus," *ACM TIST*, vol. 5, no. 1, p. 17, 2013. [Online]. Available: http://doi.acm.org/10.1145/2542182.2542199

[67] M. Montanari, E. Chan, K. Larson, W. Yoo, R. H. Campbell, J. Camenisch, S. FischerHubner, Y. Murayama, A. Portmann, and C. Rieder, "Distributed security policy conformance," in *Future Challenges In Security and Privacy For Academia and Industry*, vol. 354. Heidelberger Platz 3, D-14197 Berlin, Germany: Springer-Verlag Berlin, 2011, pp. 210–222.

[68] Object Management Group. (2015, December) Object Constraint Language (OCL). [Online]. Available: http://www.omg.org/spec/OCL/

[69] U. D. of Health and H. Services, "Code of Federal Regulations, Title 45, public welfare, department of health and human services, part 46, protection of human subjects," https://www.hhs.gov/ohrp/regulations-and-policy/regulations/45-cfr-46/index.html, July 2009. [Online]. Available: https://www.hhs.gov/ohrp/regulations-and-policy/regulations/45-cfr-46/index.html

[70] Office of Ethics and Compliance: Human Research Protection Program, University of California, San Francosco. (2015, April) The Human Research Protection Program, Definitions. [Online]. Available: http://irb.ucsf.edu/definitions

[71] M. Pesic and W. van der Aalst, "A declarative approach for flexible business processes management," in *Business Process Management Workshops*, ser. Lecture Notes in Computer Science, J. Eder and S. Dustdar, Eds. Springer Berlin Heidelberg, 2006, vol. 4103, pp. 169–180. [Online]. Available: http://dx.doi.org/10.1007/11837862_18

[72] C. A. Petri, "Communication with automata," Ph.D. dissertation, Universität Hamburg, 1966.

[73] A. Raschke, "Translation of UML 2 activity diagrams into finite state machines for model checking," in *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, Patras, Greece, August 27-29, 2009, Proceedings*. IEEE Computer Society, 2009, pp. 149–154. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SEAA.2009.60

[74] R. Rashidi-Tabrizi, G. Mussbacher, and D. Amyot, "Legal requirements analysis and modeling with the measured compliance profile for the goal-oriented requirement language," in *Sixth International Workshop on Requirements Engineering and Law, RELAW 2013, 16 July, 2013, Rio de Janeiro, Brasil*, D. Amyot, A. I. Antón, T. D. Breaux, A. K. Massey, and P. P. Swire, Eds. IEEE Computer Society, 2013, pp. 53–56. [Online]. Available: http://dx.doi.org/10.1109/RELAW.2013.6671346

[75] S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6896. Springer, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23059-2

[76] A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Inf. Syst.*, vol. 33, no. 1, pp. 64–95, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.is.2007.07.001

[77] Senate Banking Committee, "Gramm-Leach-Bliley Act - disclosure of nonpublic personal information," 1999. [Online]. Available: https://www.ftc.gov/tips-advice/business-center/privacy-and-security/gramm-leach-bliley-act

[78] A. Shaikh, R. Clarisó, U. K. Wiil, and N. Memon, "Verification-driven slicing of UML/OCL models," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859038

[79] A. Shaikh, U. K. Wiil, and N. Memon, "Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams," *Adv. Software Engineering*, vol. 2011, pp. 370 198:1–370 198:18, 2011. [Online]. Available: http://dx.doi.org/10.1155/2011/370198

[80] M. Steinhorst, P. Delfmann, and J. Becker, "vGMQL - Introducing a visual notation for the generic model query language GMQL," in *Short Paper Proceedings of the 6th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM 2013), Riga, Latvia, November 6-7, 2013.*, ser. CEUR Workshop Proceedings, J. Grabis, M. Kirikova, J. Zdravkovic, and J. Stirna, Eds., vol. 1023. CEUR-WS.org, 2013, pp. 146–155. [Online]. Available: http://ceur-ws.org/Vol-1023/paper14.pdf

[81] W. Sun, "Using slicing techniques to support scalable rigorous analysis of class models," Ph.D. dissertation, Colorado State University, 2015.

[82] W. Sun, R. B. France, and I. Ray, "Contract-aware slicing of UML class models," in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, Eds., vol. 8107. Springer, 2013, pp. 724–739. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41533-3_44

[83] H. B. K. Tan, L. Hao, and Y. Yang, "On formalization of the whole-part relationship in the unified modeling language," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1054–1055, Nov 2003.

[84] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, R. S. Mans, A. K. A. de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. E. Verbeek, and A. J. M. M. Weijters, "ProM 4.0: Comprehensive support for *Real* process analysis," in *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*, ser. Lecture Notes in Computer Science, J. Kleijn and A. Yakovlev, Eds., vol. 4546. Springer, 2007, pp. 484–494. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73094-1_28

[85] S. K. L. M. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, J. Vanthienen, R. Meersman, H. Panetto, T. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, and T. Sellis, "Event-based real-time decomposed conformance analysis," in *On the Move To Meaningful Internet Systems: Otm 2014 Conferences*, vol. 8841. Heidelberger Platz 3, D-14197 Berlin, Germany: Springer-Verlag Berlin, 2014, pp. 345–363.

[86] M. Weidlich, H. Ziekow, J. Mendling, O. Günther, M. Weske, and N. Desai, "Event-based monitoring of process execution violations," in *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, ser. Lecture Notes in Computer Science, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., vol. 6896. Springer, 2011, pp. 182–198. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23059-2_16

[87] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

A.1 Promela Model

Listing A.1: *NJH Promela model for approving an access ticket using the NoSupsInPIandDC decision rule*

```
/******************************************************************************
 * Purpose: NJH RCA Analysis
 *
 * Author: Phillipa Bennett
 *
 * Answering the question :
 *  1. Can we use Spin to answer - what is Tractable RCA?
 *  2. How can we use spin for for process order mutations?
 *
 * Date created: March 15, 2016
 *
 * Version: 1
 *
 * Parameters: updated March 23, 2016
 *  Safety: safety, +invalid endstates violation, +assertion violations
 *  Storage mode: exhaustive, +collapse compression
 *  Search mode: depth first + partial order reduction, iterative, unreachable
 *  Advanced parameters
 *      Extra compile options: -O2 -DVECTORSZ=3072 -DMA=2000
 *      Physical memory available: 7000
 *      Estimates state search space: 1000
 *      Maximum search depth: 100000000
 *      Extra runtime options:
 *
 * TBD -
 *  1. Find a way to specify alternate end states - low priority
 *
 * Need to write about -
    1. idea of how spin can be used for process order mutations.
******************************************************************************/

/******************************************************************************
 * Define
******************************************************************************/

#define PROC_BITS 3   // number of bits needed to represent process
#define PROCS 7       // this will depend on final Activity diagram used


#define PROJ_BITS 2// number of bits required to access project in projects
#define PROJS 4   // spin's current max is an unsigned n-bit where n = 8

#define SUPERS_BIT 5  // number of bits required to access supervisors
#define SUPERS 32    // number of persons needing supervisors

/******************************************************************************
 * Declarations
******************************************************************************/

mtype {deidentified, identified, none} // permission types

```

```
typedef Proc_Run{
    bool executed[PROCS];} /* helps to check process pre-requisites,
                           * except for apply */

typedef Supervisor {
    unsigned s_id : SUPERS_BIT;}

typedef Project {
    mtype access_ticket;
    bool data_collector_present = false;
    unsigned
        pi: SUPERS_BIT,
        data_collector : SUPERS_BIT;
    bool submit = 0; // prerequisite for apply process
    Proc_Run runs;}

Project projects[PROJS];
bool approve_and_decline = false;

Supervisor sups[SUPERS]; /* e.g. sups[12] = 56means supervisor of researcher
                          * with r_id 12is researcher with r_id 56*/
unsigned sup_root : SUPERS_BIT; /* this is the root of the sups tree */

/* In order to get an array of unsigned, I needed this workaround */
typedef Unsigned {unsigned id: SUPERS_BIT;}

bool init_complete = false;

/*****************************************************************************
 * LTL
*****************************************************************************/
/* ensures that we have some nondeterminism in the approve and decline of
    projects
    */
ltl ltl1 {
    /* infinitely executing the statement in approve with label app
        implies (ensures) we infinitely execute the statement labeled dec
        in approve(), implies (ensures) we infinitely execute the statement
        labeled dec in decline() */
    []<>approve@app ->
        ([]<>approve@dec && []<>decline@dec)
}

/*****************************************************************************
 * NEVER claims
*****************************************************************************/

/* **********
    A project must not be both approved and declined over this
    simulation/verification */
never noApproveDeclineOnSameProject{
   true;
   do
   :: approve_and_decline -> break;
   :: else -> skip;
   od;
}

/*****************************************************************************
 * Inline
*****************************************************************************/

```

```
/* **********/
inline add_supervisors_3bit() {
    //numbers generated from https://www.random.org/sequences/
    // root is 1
   sup_root = 1;
   sups[1].s_id = 1; sups[6].s_id = 1;
   sups[2].s_id = 6; sups[0].s_id = 6;
   sups[4].s_id = 2;
   sups[5].s_id = 0;
   sups[3].s_id = 4; sups[7].s_id = 4;
}

/* **********/
inline add_supervisors_5bit() {
    //numbers generated from https://www.random.org/sequences/
    // root is 1
    sup_root = 1;
    sups[1].s_id = 1; sups[14].s_id = 1;

    sups[11].s_id = 14; sups[20].s_id = 14;

    sups[17].s_id = 11; sups[5].s_id = 11; sups[15].s_id = 11;
    sups[29].s_id = 11; sups[9].s_id = 11;

    sups[2].s_id = 20;

    sups[22].s_id = 17;

    sups[4].s_id = 5; sups[31].s_id = 5;

    sups[23].s_id = 15; sups[24].s_id = 15;

    sups[10].s_id = 29; sups[21].s_id = 29;

    sups[28].s_id = 9; sups[8].s_id = 9;

    sups[26].s_id = 2; sups[25].s_id = 2;

    sups[13].s_id = 22; sups[3].s_id = 22;

    sups[6].s_id = 4;

    sups[27].s_id = 31; sups[18].s_id = 31; sups[16].s_id = 31;
    sups[12].s_id = 31;

    sups[19].s_id = 23; sups[7].s_id = 23; sups[30].s_id = 23;

    sups[0].s_id = 24;
}

inline check_supervisor_assignments() {

    for (m: 0..(SUPERS-1)) {
        if
        :: m == sup_root ->
            assert(sups[m].s_id == m);
        :: else ->
            assert(sups[m].s_id != m );
        fi;
    }
}

```

```
/* **********/
inline set_process_bit() {
    d_step{
        // update the process bit
        projects[project].runs.executed[id] = 1;
        // assert
        assert(projects[project].runs.executed[id] == 1&&
            projects[project].runs.executed[dependsOn] == 1);
    }
}

inline check_approve_conditions () {
    approve_project =
        projects[project].access_ticket != none &&
        ( !projects[project].data_collector_present == true ||
         !(
          // common supervisor
          (sups[projects[project].data_collector].s_id ==
           sups[projects[project].pi].s_id)

          // data collector supervisor is project's pi
          || (sups[projects[project].data_collector].s_id ==
              projects[project].pi)

          //  pi supervisor is project's data collector
          || (sups[projects[project].pi].s_id ==
              projects[project].data_collector)
          )
         );
}

/*****************************************************************************
 * Processes
 *****************************************************************************/

/***************************************/
 active proctype apply () {
     unsigned
        dependsOn : PROC_BITS = 0,
        id : PROC_BITS = 0,
        project : PROJ_BITS;

    //init_complete == true;
    /* end: */
    again:
        select(project: 0..3);
        if
        :: projects[project].runs.executed[id] == 0&&
                projects[project].submit == 1->
            /* progress: */ set_process_bit();
        :: else -> skip;
        fi
     goto again;
}

/***************************************/
 active proctype approve () {
    // process changes these values
    unsigned project : PROJ_BITS;
    bool approve_project;

    // process does not change these values
```

```
    unsigned
        dependsOn : PROC_BITS = 0,
        id : PROC_BITS = 1;

    //init_complete == true;
    /* end: */
    again:
        approve_project = true;
        select(project: 0..3);
        if
        :: projects[project].runs.executed[dependsOn] == 1&&
                projects[project].runs.executed[id] == 0->

            check_approve_conditions();
            if
            :: approve_project == true ->
                app: /* progress: */ {set_process_bit();}
            :: else ->
                dec: {projects[project].access_ticket = none;}
            fi;
        :: else -> skip;
        fi;
     goto again;
}

/************************************/
active proctype decline () {
    // process does not change these values
    unsigned
        dependsOn : PROC_BITS = 0,
        id : PROC_BITS = 2;

    // process changes this value
    unsigned project: PROJ_BITS
    bool approve_project;

    //init_complete == true;
    /* end: */
    again:
        approve_project = true;
        select(project: 0..3);
        if
        :: (projects[project].runs.executed[dependsOn] == 1&&
                projects[project].runs.executed[id] == 0) ->
            check_approve_conditions();
            if
            :: approve_project == true ->
                dec: /* progress: */ {set_process_bit();}
            :: else -> skip;
            fi;
        :: else -> skip;
        fi;
    goto again
}

/************************************/
proctype proc (byte id, dependsOn) {
            //(unsigned dependsOn: PROC_BITS, id: PROC_BITS ) {
    unsigned project : PROJ_BITS;

    //init_complete == true;
    /* end: */
```

```
    again:
        select(project: 0..3);
        if
        :: projects[project].runs.executed[id] == 0&&
                projects[project].runs.executed[dependsOn] == 1->
            /* progress: */ set_process_bit();
        :: else -> skip;
        fi
    goto again;
}

/************************************/
active proctype check_approve_and_decline() {
    // process changes this values
    unsigned project : PROJ_BITS;

    //init_complete == true;
    /* end: */
    again:
        select(project: 0..3);
        assert(projects[project].submit == 1);
        if
        :: projects[project].runs.executed[1] == 1&&
                projects[project].runs.executed[2] == 1->
            approve_and_decline = 1;
            //assert(false);
        :: else -> skip;
        fi;
        goto again;

}

/************************************/
init{
    unsigned
        // for choosing values non-deterministicly
        n : SUPERS_BIT = 0;
    // for counters, cannot use unsigned type vor variables used in for loops?
    byte l, m;

    add_supervisors_5bit();
    check_supervisor_assignments();
    m = 0;

    for (l: 0..(PROJS-1)) {

        for (m: 0..(PROCS-1)) {
            projects[l].runs.executed[m] = false;
        }

        if
        :: projects[l].access_ticket = deidentified;
        :: projects[l].access_ticket = identified;
        fi

        // choose project's pi
        select(n: 0..31);
        projects[l].pi = n;

        // choose whether project has data collector
        if
        :: projects[l].data_collector_present = false
```

```
        :: projects[l].data_collector_present = true

        fi

        if
        :: projects[l].data_collector_present == true ->
            // ensure data collector chosen will not overlap with pi
            choose_n_again: {
                select(n: 0..31);
                if
                :: projects[l].pi == n ->
                    goto choose_n_again;
                :: else -> skip;
                fi;
            }

            // assign data collector
            projects[l].data_collector = n;
            assert(projects[l].pi != projects[l].data_collector);
        :: else -> skip;
        fi
        projects[l].submit = 1;
    }

    init_complete = true;

    // query
    run proc (3, 1);

    // transform
    run proc (4, 3);

    // view
    run proc (5, 4);

    // download
    run proc(6, 4);
}
```

A.2 Alloy Models

The model for the full NJH system used in the motivation is presented in four parts, Listing A.2 through Listing A.5.

Listing A.2: *Full NJH structural model, i.e., without additional constraints, operation specifications, or conformance rules. These are added in Listing A.3 through Listing A.3*

```
/********** ********** ********** ********** ********** ********** **********
   Begin Structural Model, NJH
/********** ********** ********** ********** ********** ********** **********/
module NJH

/********** ********** ********** ********** **********
   base abstract signatures
********** ********** ********** ********** **********/
abstract sig
   Category,
   Data,
   DataSource,
   DataTransform,
   Permission,
   Purpose,
   Rule,
   Status,
   Type {}

/********** ********** ********** ********** **********
   extended abstract signatures
********** ********** ********** ********** **********/
abstract sig
   AccessTicket,
   Licence
extends Permission{}

abstract sig
   AccessRule,
   DecisionRule
extends Rule {}

abstract sig HIPAACat extends Category{}
abstract sig Consent extends Category{}

/********** ********** ********** ********** **********
   unextended concrete signatures
********** ********** ********** ********** **********/
sig Day,
   Month,
   Name,
   Patient,
   Personnel, // this cannot be abstract
   Query,
   Year {}

sig DataItem {
   name: Name}

```

```
/********** ********** ********** ********** **********
   extended concrete signatures
********** ********** ********** ********** **********/
one sig
   DeIDedTransformHDate,
   IdentifiedDoesNotTransformHDate,
   PatientConsent
   //ProtectedChild,
   //ProtectedPregnantWomen
extends AccessRule {}

one sig
   CanUseTotallyDeIDed,
   DataAccessAgreementPresent,
   DataSourcePriorityOK,
   LicenedTeamAndPI,
   NoOverlapPITeamDC,
   NoSupsInPIandDC,
   PIDefined,
   ProjectMembersDefined,
   QualifierPresent,
   SomePurposeNotDirectTreatment,
   SomeQueriesDefined,
   SomeSourcesDefined
extends DecisionRule {}

one sig
   Allow,
   Disallow
extends Consent {}

one sig
   TotallyDeIDed,
   NotTotallyDeIDed
extends DataTransform {}

sig Project extends DataSource{}
one sig ClinicalDB extends DataSource{}

one sig
   HDate,
   HProtectedChild,
   HProtectedPregnantWoman
extends HIPAACat {}

one sig Fishing extends Licence {}

one sig DeIDed,
   Identified
extends AccessTicket {}

one sig
   DirectTreatment,
   Research
extends Purpose{}

one sig
   DownloadAllowed,
   DownloadDisabled
extends Status {}

one sig
```

```
  Group,
  Individual
extends Type {}

sig Date extends Data {
  day: lone Day,
  month: lone Month,
  year: Year }{
  //  day iff month also exists
  some day iff some month }
sig dStr extends Data {}
/********** ********** ********** ********** **********
  subset concrete signatures
********** ********** ********** ********** **********/
sig
  Qualifier,
  Researcher
in Personnel{}

//changed extends to in, due to identified access ticket
sig
  QryData,
  RetData
in DataItem {}

/********** ********** ********** ********** **********
  NJH Closed System
********** ********** ********** ********** **********/
sig NJH {
  accessRules: set AccessRule,
  accessTickets: set AccessTicket,
  categories: set Category,
  consents: set Consent,
  dataItems: set DataItem,
  dates: set Date,
  decisionRules: set DecisionRule,
  hCats: set HIPAACat,
  licences: set Licence,
  patients: set Patient,
  permissions: set Permission,
  personnel: set Personnel,
  projects: set Project,
  purposes: set Purpose,
  qryItems: set QryData,
  qualifiers: set Qualifier,
  queries: set Query,
  researchers : set Researcher,
  retItems: set RetData,
  rules: set Rule,
  sources: set DataSource,
  statuses: set Status,
  transforms: set DataTransform,
  types: set Type,
  values: set Data,

  /* access rules applies to these types.  */
  ARAppliesTo: accessRules -> some types,

  /* access rule transforms data linked to this hipaa category */
  ARTransforms: accessRules -> hCats,

  // access rule hides these categories if they are disallowed by Consent
```

```
ARHides: accessRules -> categories,

/* helps to determine
   1. if data from a project can be used as a data source */
ATPriority : accessTickets -> accessTickets,

// p1->p2 means p1 gives p2 access to data produced by p1
dataAccessAgreement: projects -> projects,

/* data items must a value or not. */
dataValues: dataItems -> one values,

/* each data item is linked to a perticular hipaa category. we do not need to
   link the retitems because we know the DICat of retItems through the
   RDFromQD relation */
DICat: (dataItems - retItems) -> hCats,

/* not neccessary to have a direct (i.e. one-to-one) link between retItems
   and sources becaues retItems may be grouped. Data sources of retItems
    are found through the RDFromQD relation */
DISource: dataItems -> one sources,

enteredOn: dataItems -> lone dates,

/* not neccessary to have a direct (i.e. one-to-one) link between retItems
   and patients becaues retItems may be grouped. Patients associated
   with retItems are found through the RDFromQD relation */
patientData: patients one -> some qryItems -> one consents,

/* permission has applicable decision and access rules that must be
   applied to approve the licence or to access the data. */
permRules: permissions -> some rules,

/* project access tickets, each one has at most one */
projectAT: projects -> lone accessTickets,

/* project data collector, each project has at most one */
projectDataCollector: projects -> lone personnel,

projectDataTransformRequired: projects -> one transforms,

/* project team members */
projectMembers: projects -> researchers,

/* project principal investigator */
projectPI: projects -> lone researchers,

/* project purpose */
projectPurpose: projects -> lone purposes,

/* project queries */
projectQueries: projects one -> queries,

/* project sources, could be other projects too */
projectSources: projects -> sources,

// a query can work on any kind of data item
qryReturns: queries -> dataItems -> dataItems,

// a query can return any kind of data item
qryWorksOn: queries -> dataItems,

```

```
   /* returned data from query data, each piece of retdata is derived from
      at most 1qryitem because we are only working on the Individual Type
      right now.
      Hoewever because we are using different access tickets, qryItems
      may be linked to more than one return types. The max is 2because
      we have two fifferent Transform rules*/
   //RDFromQD: retItems -> one qryItems,

   /* return data type, has 0or 1type */
   RDType: queries -> retItems -> lone types,

   /* researcher licence */
   researcherL: researchers -> lone licences,

   /* researcher qualifier, at most one qualifier */
   resQualifier: researchers -> lone qualifiers,

   /* supervisors, each personnel has at most one supervisor */
   supervisors: personnel lone -> personnel,

   /* determines is query results meets conformance and the next
      operation, i.e. view/download is allowed */
   VDAllowed: queries -> lone statuses }

/********** ********** ********** ********** ********** ********** **********
      End Structural Model, NJHg
/********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** ********** ********** **********
   These are not a part of the model. The provide sanity
   checks before going on to write the operation specifications
/********** ********** ********** ********** ********** ********** **********/
/********** ********** ********** ********** **********
    any instance of the model
********** ********** ********** ********** **********/
private pred show (njh: NJH) {}
run show for 7but exactly 15Rule, 1NJH expect 1
```

Listing A.3: *Full NJH structural model: adding constraints. Imports Listing A.2 on line 24.*

```
1  /********** ********** ********** ********** ********** ********** **********
2     Begin Structural Model With (Generator) Invariants, NJHg
3  
4     Executing any of the predicates or assertions requires a
5     minimum of 13rules
6  
7     To do:
8     17/04/2016
9     To add invariants for
10    1. how an AT is obtained - done 25/04/2016
11    2. for how runQuery changes
12       qryWorksOn,
13       qryReturns,
14       RDType,
15       enteredOn
16    3. How Update Conformance works with qryReturns
17 
18 /********** ********** ********** ********** ********** ********** **********/
19 module NJHg
20 
21 /********** ********** ********** ********** **********
22    imports
23 /********** ********** ********** ********** **********/
24 open NJH
25 open util/relation
26 open util/ternary
27 
28 /********** ********** ********** ********** **********
29   INVARIANTS
30    separating the invariants for each set,
31    relation, or related sets and relations
32    allows for easier decomposition later on
33    when doing slicing
34 ********** ********** ********** ********** **********/
35 // this signature is exported from the model, it is used in inv[]
36 pred generator (njh: NJH) {
37    all
38       njh: NJH |
39 
40    //for sets
41    invCategory[njh] and
42    invDatItems[njh] and
43    invDates[njh] and
44    invPermissions[njh] and
45    invPersonnel[njh] and
46    invRules[njh] and
47    invSources[njh] and
48 
49    // for relations
50    invARAppliesTo[njh] and
51    invATPriority[njh] and
52    invARHides[njh] and
53    invARTransforms[njh] and
54    invDataAccessAggreement[njh] and
55    invDISource[njh] and
56    invEnteredOn[njh] and
57    invPatientDataAndDICat[njh] and
58    invPermRules[njh] and
59    //invProjectAT and
```

```
   invProjectDataCollector[njh] and
   invProjectSources[njh] and
   invQryReturns[njh] and
   invQryWorksOn[njh] and
   invRDType[njh] and
   invResearcherL[njh] and
   invResQualifier[njh] and
   invSupervisors[njh] and
   invVDAllowed[njh] and
   setPredefinedConfigurations[njh] }

/********** ********** ********** ********** **********
  Some Functions and Predicates to be reused
********** ********** ********** ********** **********/

fun DeIDedDateTransform (d: Date): Date {
   {ri: Date |
      no ri.day and
      no ri.month and
      ri.year = d.year }}

fun IdentifiedDateTransform (d: Date): Date {
   { ri: Date | ri = d }}

pred identifiedDate (d: Date) {
   some d.day }

/********** ********** ********** ********** **********
  Set Invariants,
   ordered alphabetically as best as possible
********** ********** ********** ********** **********/

private pred invCategory (njh: NJH) {
   njh.categories =
      njh.consents + njh.hCats }

private pred invDatItems (njh: NJH) {
   (njh.qryItems + njh.retItems) in njh.dataItems}

private pred invDates (njh: NJH) {
   // closed system constraint - any date is a part of the set of dates
   njh.dates = (njh.values & Date) + ran[njh.enteredOn]

   all
      d: Date |
   (d in njh.dates and identifiedDate[d]) implies
      DeIDedDateTransform[d] in njh.dates}

private pred invPermissions (njh: NJH) {
   njh.permissions = njh.accessTickets + njh.licences }

private pred invPersonnel (njh: NJH) {
   (njh.researchers + njh.qualifiers) in njh.personnel}

private pred invRules (njh: NJH) {
   njh.rules = njh.accessRules + njh.decisionRules }

private pred invSources (njh: NJH) {
   njh.projects in njh.sources }

/********** ********** ********** ********** **********
  Relation Invariants,
```

```
   ordered alphabetically as best as possible
********** ********** ********** ********** **********/

// replaces TransFormHDateAppliesToIndividual in old specifications
private pred invARAppliesTo [njh: NJH] {
   some njh.ARAppliesTo &
      DeIDedTransformHDate-> Individual }

private pred invATPriority (njh: NJH) {
      irreflexive[^(njh.ATPriority)] }

private pred invARHides (njh: NJH) {
   no njh.ARHides & njh.ARTransforms }

// DeIDedTransformHDate applies to HDate HIPAACat
private pred invARTransforms (njh: NJH) {
   some njh.ARTransforms & DeIDedTransformHDate -> HDate }

 //p1->p2 means p1 gives p2 access to data produced by p1
private pred invDataAccessAggreement (njh: NJH) {
   // no project has a data access agreement with itself
   irreflexive[^(njh.dataAccessAgreement)]

   /* a project with a data access agreement with another
      project has that project as a data source */
   ~(njh.dataAccessAgreement) in njh.projectSources }

private pred invDISource1 (njh: NJH) {
   all
      s: njh.sources |
   some s & Project
   // project can only have retItems as data items
   implies
      njh.DISource.s in njh.retItems
   // otherwise no retitems as data items
   else
      njh.DISource.s in (njh.dataItems - njh.retItems) }

/* we can trace every ri back to some (set of) patientData qi (qis)
   and if any of the qi's is linked to HDate, and the access ticket used
   to create the ri is DeIDed, the ri must also be de-identified. */
private pred invDISource2 (njh: NJH) {
   all
      da: (njh.DISource).(njh.projects) |
   some njh.qryReturns.da implies
   some da -> ClinicalDB & njh.DISource.(njh.projectSources) }

private pred invDISourceAndEnteredOn (njh: NJH) {
   all
      di: njh.dataItems |
   some di.(njh.DISource) & ClinicalDB implies
      identifiedDate[di.(njh.enteredOn)] }

private pred invDISource (njh: NJH) {
   invDISource1[njh] and
   invDISource2[njh] and
   invDISourceAndEnteredOn[njh]}

private pred invEnteredOn (njh: NJH) {
   // dataItems in Patient data
   all
      di: mid[njh.patientData] | {
```

```
   // each has a date entered, we don't care if retItems are not in enteredOn?
   some di.(njh.enteredOn) and
   // each enteredOn data has a day and month (constraint in Date signature
   //   ensures that month is non-empty iff day is non-empty)
   some di.(njh.enteredOn.day) }}

 // replaces AllDatesCorrectlyCategorised in old specifications
private pred invPatientDataAndDICat[njh: NJH] {
   /* All dates in patient data are correctly categorised
      as HDate HIPAACat */
   all
      di: mid[njh.patientData] |
   some di.(njh.dataValues) & Date implies
      some di.(njh.DICat) & HDate }

// replaces TransformHDateIsDeIDedRule in old specifications
private pred invPermRules (njh: NJH) {
   // DeIDedTransformHDate is linked with DeIDed access ticket
   some njh.permRules &
      DeIDed -> DeIDedTransformHDate and
   // (so far) only the DeIDed access ticket has the DeIDedTransformHDate rule
   njh.permRules.DeIDedTransformHDate = DeIDed }

private pred invProjectAT (njh: NJH) {
   // ********** for approve project access ticket
   all
      p: njh.projects |
   let
      dr =
         CanUseTotallyDeIDed +
         DataAccessAgreementPresent+
         DataSourcePriorityOK +
         LicenedTeamAndPI +
         NoOverlapPITeamDC +
         NoSupsInPIandDC +
         PIDefined +
         ProjectMembersDefined +
         SomePurposeNotDirectTreatment +
         SomeQueriesDefined +
         SomeSourcesDefined,
      di = dr - CanUseTotallyDeIDed,
      d = DeIDed,
      i = Identified,
      pat = njh.projectAT |

   some p.pat implies (

   // specific for DeIDed access tickets
      some p -> d & pat implies (
         // kind of Transformation access ticket allows
         some p->TotallyDeIDed & njh.projectDataTransformRequired and
         // rules that apply to the DeIDed access ticket
         d.(njh.permRules) & njh.decisionRules = dr )

   and

   // specific for Identified access tickets
      some p -> i & pat implies (
         // kind of Transformation access ticket allows
         some p -> NotTotallyDeIDed & njh.projectDataTransformRequired and
         // rules that apply to the DeIDed access ticket
         d.(njh.permRules) & njh.decisionRules = di )
```

```

and

   all
      ps: p.(njh.projectSources) & njh.projects | {
   // application of the DataAccessAgreementPresent Decision Rule
   some ps -> p & njh.dataAccessAgreement and
   /* application of the DataSourcePriorityOK Decision Rule

      if access ticket being considered has priority over
      the access tickets of any of its project sources
      (i.e. other projects) }then we cannot approve the
      project because the data returned would not be at
      the level required */
   no (d+i) -> ps.(njh.projectAT) & njh.ATPriority }
and

let
   team = p.(njh.projectMembers),
   pi = p.(njh.projectPI),
   dc = p.(njh.projectDataCollector) | {

all
   r: (team + pi) | {
/* application of the LicenedTeamAndPI Decision Rule
      each pi and team member has a licence */
   some r.(njh.researcherL) }and
/* application of the NoOverlapPITeamDC Decision Rule
      1. neither pi nor dc are a part of project team */
   no (pi + dc) & team and
      // 2. pi and da are not the same
   no pi & dc and
/* application of the ProjectMembersDefined Decision Rule
   > 1 team members */
   #team > 0and
/* application of the PIDefined Decision Rule
   has a pi */
   #pi> 0}

and

/* application of the NoSupsInPIandDC Decision Rule
   neither the pi nor the da supervise each other
   directly or indirectly */
   let
      sup = p.(njh.projectPI) -> p.(njh.projectDataCollector) | {
   no (sup + ~sup ) & ^(njh.supervisors) }

and

/* application of the SomePurposeNotDirectTreatment Decision Rule
    project purpose is not for direct treatment */
   p.(njh.projectPurpose) != DirectTreatment

and

/* application of the SomeQueriesDefined Decision Rule
    at least one project query */
   some p.(njh.projectQueries)

and

```

```
  /* application of the SomeSourcesDefined Decision Rule
      at least one project source */
     some p.(njh.projectSources) ) }

private pred invProjectDataCollector(njh: NJH) {
  all
     p: njh.projects |
  // ClinicalDB iff DataCollector
  (some p->ClinicalDB & njh.projectSources) implies
     (some p.(njh.projectDataCollector) ) }

private pred invProjectSources1 (njh: NJH) {
  // no self datasource for projects, directly or indirectly
  irreflexive[^(njh.projectSources :> njh.projects)] }

private pred invProjectSources2 (njh: NJH ) {
  all
     p: njh.projects |
  some p.(njh.projectAT) implies
     /* all data sources for a project that are projects themselves
        should be (already) approved when the project gets it's
        access ticket */
     all
        ps: (p.(njh.projectSources) & Project) |
     some ps.(njh.projectAT) }

private pred invProjectSources (njh: NJH) {
  invProjectSources1[njh] and
  invProjectSources2[njh] }

private pred invQryReturns1 (njh: NJH) {
  all
     q: njh.queries |
  some q.(njh.qryReturns) implies
     ran[q.(njh.qryReturns)] in q.(njh.qryWorksOn) }

private pred invQryReturns2 (njh: NJH) {
  all
     q: njh.queries |
  some q.(njh.qryReturns) implies
     some njh.projectQueries.q.(njh.projectAT) }

private pred invQryReturns (njh: NJH) {
  invQryReturns1[njh] and
  invQryReturns2[njh] }

private pred invQryWorksOn (njh: NJH) {
  all
     q: njh.queries,
     qi: njh.qryItems |
  let
     qSources = (njh.projectQueries).q.(njh.projectSources) |
  // constraints on what can be in QryWorksOn for a query
  some q -> qi & njh.qryWorksOn implies
  (qi in (njh.DISource).qSources and
  no qi -> Disallow & select23[njh.patientData]) }

private pred invRDType (njh: NJH) {
  all
     q: njh.queries,
     r: njh.retItems |
  let
```

```
      qrq = (r.(q.(njh.qryReturns))) {
   // these are the entries
   select12[njh.RDType] = select12[njh.qryReturns]

   // individual type
   some q -> r -> Individual & njh.RDType iff
      #qrq = 1

   // group type
    some q -> r -> Group & njh.RDType iff
      #qrq > 1 } }

private pred invResearcherL (njh: NJH) {
   // ********** for approve researcher licence
   all
      res: njh.researchers |
   some res.(njh.researcherL) implies
      // researcher is qualified
      some res.(njh.resQualifier) and
      // the licence granted required qualification
      (res.(njh.researcherL)).(njh.permRules) =
         QualifierPresent }

private pred invResQualifier (njh: NJH) {
      // ********** for qualify researcher this should always be true
   no iden & ^(njh.resQualifier) }

private pred invSupervisors (njh: NJH) {
   // no cycles in supervisor relations,
   irreflexive[^(njh.supervisors)]
   // all personnel are either supervisor or supervised
   all
      p: njh.personnel | {
    p in (dom[njh.supervisors] + ran[njh.supervisors])} and
   /* supervisor relation is a single tree, i.e. not a forest
      this means that one personel has no supervisor */
   one
      sup: njh.personnel |
   no (njh.supervisors).sup }

// this checks only for DeIDed access ticket
private pred invVDAllowedDeIDed(
   njh: NJH,
   qry: Query) {
   let
      at = (njh.projectQueries).qry.(njh.projectAT) |

   some at & DeIDed iff
   all
      d: ((Date & dom[qry.(njh.qryReturns)].(njh.dataValues)) +
         dom[qry.(njh.qryReturns)].(njh.enteredOn)) |
   not identifiedDate[d] }

// this checks only for Identified access ticket
private pred invVDAllowedIdentified(
   njh: NJH,
   qry: Query) {
   let
      at = (njh.projectQueries).qry.(njh.projectAT) |

   some at & Identified iff
   all
```

```
      d: ((Date & dom[qry.(njh.qryReturns)].(njh.dataValues)) +
         dom[qry.(njh.qryReturns)].(njh.enteredOn)) |
   identifiedDate[d] }

pred invVDAllowed1 (
   njh: NJH,
   q: Query) {

   (invVDAllowedDeIDed[njh, q] and
      invVDAllowedIdentified[njh, q] ) }

private pred invVDAllowed (njh: NJH) {
   all
      q: njh.queries | {
   // if a query has a a VD status then it has some return data
   some q.(njh.VDAllowed) implies
      some q.(njh.qryReturns)

   some q -> DownloadAllowed & njh.VDAllowed implies
      invVDAllowed1[njh, q]

   some q -> DownloadDisabled & njh.VDAllowed implies
      not invVDAllowed1[njh, q] }}

private pred setPredefinedConfigurations (njh: NJH) {
   // for sets
   njh.accessRules = // 5
      DeIDedTransformHDate +
      IdentifiedDoesNotTransformHDate +
      PatientConsent and
      //ProtectedChild +
      //ProtectedPregnantWomen and

   njh.decisionRules = //13
      CanUseTotallyDeIDed +
      DataAccessAgreementPresent +
      DataSourcePriorityOK +
      LicenedTeamAndPI +
      NoOverlapPITeamDC +
      NoSupsInPIandDC +
      PIDefined +
      ProjectMembersDefined +
      SomePurposeNotDirectTreatment +
      QualifierPresent +
      SomeQueriesDefined +
      SomeSourcesDefined and

   // access tickets (2)
   njh.accessTickets =
      DeIDed +
      Identified and

   // licences (1)
   njh.licences = Fishing and

   // statuses (2)
   njh.statuses =
      DownloadAllowed +
      DownloadDisabled and

   // transforms (2)
   njh.transforms =
```

```
    TotallyDeIDed +
  NotTotallyDeIDed and

  //sources (at least 1)
  some ClinicalDB & njh.sources and

  // types
  njh.types = ran[njh.ARAppliesTo] and

  // for relations
  // access ticket priority (1)
  njh.ATPriority = Identified -> DeIDed and

  //ARAppliesTo: accessRules -> some types (3)
  njh.ARAppliesTo =
    DeIDedTransformHDate -> Individual +
    IdentifiedDoesNotTransformHDate -> Individual +
    PatientConsent -> Individual and

  //ARTransforms: accessRules -> some hCats (2)
  njh.ARTransforms =
    DeIDedTransformHDate -> HDate +
    IdentifiedDoesNotTransformHDate -> HDate and

  //ARHides: accessRules -> some hCats (1)
  njh.ARHides =
    PatientConsent -> Disallow and

  //permRules: permissions -> some rules (26)
  njh.permRules =
    // access rules for DeIDed access ticket (2)
    DeIDed -> DeIDedTransformHDate +
    DeIDed -> PatientConsent +

    // access rules for Identified access ticket (2)
    Identified ->IdentifiedDoesNotTransformHDate +
    Identified -> PatientConsent +

    // decision rules for fishing licence (1)
    Fishing -> QualifierPresent +

    // decision rules for DeIDed access ticket (11)
    DeIDed -> CanUseTotallyDeIDed +
    DeIDed -> DataAccessAgreementPresent+
    DeIDed -> DataSourcePriorityOK +
    DeIDed -> LicenedTeamAndPI +
    DeIDed -> NoOverlapPITeamDC +
    DeIDed -> NoSupsInPIandDC +
    DeIDed -> PIDefined +
    DeIDed -> ProjectMembersDefined +
    DeIDed -> SomePurposeNotDirectTreatment +
    DeIDed -> SomeQueriesDefined +
    DeIDed -> SomeSourcesDefined +

    // decision rules for Identified access ticket (10)
    Identified -> DataAccessAgreementPresent+
    Identified -> DataSourcePriorityOK +
    Identified -> LicenedTeamAndPI +
    Identified -> NoOverlapPITeamDC +
    Identified -> NoSupsInPIandDC +
    Identified -> PIDefined +
    Identified -> ProjectMembersDefined +
```

```
      Identified -> SomePurposeNotDirectTreatment +
      Identified -> SomeQueriesDefined +
      Identified -> SomeSourcesDefined and

   /* Important to add these so that Alloy does not use a
      subset of the configuration!!!
      This is important when setting object configurations too */
   #njh.accessRules = 3and
   #njh.decisionRules = 12and
   #njh.accessTickets = 2and
   #njh.licences = 1and
   #njh.statuses = 2and
   #njh.sources > 0and
   #njh.transforms = 2and
   #njh.types = #ran[njh.ARAppliesTo] and
   #njh.ATPriority = 1and
   #njh.ARAppliesTo = 3and
   #njh.ARTransforms = 2and
   #njh.ARHides = 1and
   #njh.permRules = 26}

/********** ********** ********** ********** ********** ********** **********
      End Structural Model, NJHg
/********** ********** ********** ********** ********** ********** **********/


/********** ********** ********** ********** ********** ********** **********
   These are not a part of the model. The provide sanity
   checks before going on to write the operation specifications
/********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
    any instance of the model
********** ********** ********** ********** **********/
private pred show (njh: NJH) {}
run show for 7but 1NJH expect 1

/********** ********** ********** ********** **********
   We can get an instance of the model for all
   the relations?
********** **sets******* ********** ********** **********/
private pred someOfAllSets(njh: NJH) {
   some njh.accessRules and
   some njh.accessTickets and
   some consents and
   some njh.dataItems and
   some njh.dates and
   some njh.decisionRules and
   some njh.hCats and
   some njh.licences and
   some njh.patients and
   some njh.permissions and
   some njh.personnel and
   some njh.projects and
   some njh.purposes and
   some njh.qryItems and
   some njh.qualifiers and
   some njh.queries and
   some njh.researchers and
   some njh.retItems and
   some rules and
   some njh.sources and
```

```
   some njh.statuses and
   some njh.transforms and
   some njh.types and
   some njh.values  }
run someOfAllSets for 7but 1NJH expect 1

/********** ********** ********** ********** **********
   We can get an instance of the model for all
   the relations?
********** ********** ********** ********** **********/
private pred someOfAllRelations(njh: NJH) {
   some njh.ARAppliesTo and
   some njh.ARHides and
   some njh.ARTransforms and
   some njh.ATPriority and
   some njh.dataAccessAgreement and
   some njh.dataValues and
   some njh.enteredOn and
   some njh.DICat and
   some njh.DISource and
   some njh.patientData and
   some njh.permRules and
   some njh.projectAT and
   some njh.projectDataCollector and
   some njh.projectDataTransformRequired and
   some njh.projectPurpose and
   some njh.projectSources and
   some njh.projectPI and
   some njh.projectMembers and
   some njh.projectQueries and
   some njh.qryReturns and
   some njh.qryWorksOn and
   some njh.RDType and
   some njh.resQualifier and
   some njh.researcherL and
   some njh.supervisors and
   some VDAllowed }
run someOfAllRelations for 7but 1NJH expect 1

/********** ********** ********** ********** **********
   We can get an instance of the model for all
   the relations that satisfy generator[]?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingGenerator (
   njh: NJH) {
      someOfAllRelations[njh] and generator[njh] }
run someOfAllRelationsSatisfyingGenerator for 7
   but 15Rule, 1NJH expect 1

/********** ********** ********** ********** **********
   We can get an instance of the model for all
   the relations that satisfy generator[] and a
   project has an Identified access Ticket?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingGeneratorForIdentifiedAT(
   njh: NJH, at: Identified) {
   some njh.projectAT.at and
      someOfAllRelations[njh] and generator[njh] }
run someOfAllRelationsSatisfyingGeneratorForIdentifiedAT
   for 7 but 15Rule, 1NJH expect 1

/********** ********** ********** ********** **********
```

```
   We can get an instance of the model for all
   the relations that satisfy generator[] and a
   project has a DeIDed access Ticket?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingGeneratorForDeIDedAT (
   njh: NJH, at: DeIDed) {
   some njh.projectAT.at and
      someOfAllRelations[njh] and generator[njh] }
run someOfAllRelationsSatisfyingGeneratorForDeIDedAT
   for 7 but 15 Rule, 1 NJH expect 1

/********** ********** ********** ********** **********
   all sets that are defined are used!
   using IFF instead of IMPLIES is not applicable
   because lone on some sides of the relations.
********** ********** ********** ********** **********/
assert TestIfAllSetsAreApplicableToTheModel {
   all
      njh: NJH |
   someOfAllRelationsSatisfyingGenerator[njh] implies
      someOfAllSets[njh] }
check TestIfAllSetsAreApplicableToTheModel for 7
   but 15 Rule, 1 NJH expect 0
```

Listing A.4: *Full NJH structural model: adding operation specifications. Imports Listing A.3 on line 9.*

```
1 /********** ********** ********** ********** ********** ********** **********
2      Begin Process Model, NJHgPM
3 /********** ********** ********** ********** ********** ********** **********/
4 module NJHgPM
5
6 /********** ********** ********** ********** **********
7 IMPORTS
8 ********** ********** ********** ********** **********/
9 open NJHg
10 open util/ordering[NJH] as ord
11
12 /********** ********** ********** ********** **********
13    SOME NOTES ON OPERATION TRACES
14
15    Since inv is not a fact, every instance on NJH
16    will not satisfy the invariants, just the ones
17    that have an operation applied on them.
18    This means that saying:
19
20      all nhj: NHJ | inv[njh]
21
22    in an assertion will always return a
23    counterexample.
24
25    However we know that:
26
27      all
28        njh, njh': NJH |
29      (inv[njh] and op[njh, njh'])
30        implies inv[njh']
31
32    should not return counterexamples.
33
34    The fact called traces enforces this -
35    the initial state satisfies inv[] and all next
36    states should satisfy inv[] as well.
37
38 ********** ********** ********** ********** **********/
39
40 // used in Traces for the first state in ord
41 private pred init (
42    njh: NJH) {
43    // operations work on these so initial none of them
44
45    // for sets
46    // NONE
47
48    // for relations
49    no njh.resQualifier and
50    no njh.researcherL and
51    no njh.projectAT and
52    no njh.qryReturns and
53    no njh.qryWorksOn and
54    no njh.RDType and
55    no njh.VDAllowed}
56 run init for 7but 15Rule, 1NJH expect 1
57
58 fact Traces {
59    // get the initial state, i.e. the first state in sequence ord
```

```
  init[ord/first]
  // the first state fulfils the generator constraints
  generator[ord/first]
  all
    /* since last does not have a next state, do not use it here.
       used later in njh.next */
    njh: NJH - ord/last |
  some
    res: Researcher,
    per: Personnel,
    lic: Licence,
    proj: Project,
    at: AccessTicket,
    qry: Query |
  let
    /* set the next state */
    njh' = njh.next |
  /* possible operations on the state */
  qualifyResearcher[njh, njh', res, per] or
  approveResearcherL[njh, njh', res, lic] or
  approveProjectAT[njh, njh', proj, at] or
  runQuery[njh, njh', res, proj, qry, at] or
  updateConformance[njh, njh', qry] or
  skip[njh, njh'] }

/********** ********** ********** ********** **********
  REUSE - predicates and functions
********** ********** ********** ********** **********/
// the sets do not change
private pred noChangeSets (njh, njh': NJH) {
  njh.accessRules = njh'.accessRules and
  njh.accessTickets = njh'.accessTickets and
  njh.categories = njh'.categories and
  njh.consents = njh'.consents and
  njh.dataItems = njh'.dataItems and
  njh.dates = njh'.dates and
  njh.decisionRules = njh'.decisionRules and
  njh.hCats = njh'.hCats and
  njh.licences = njh'.licences and
  njh.patients = njh'.patients and
  njh.permissions = njh'.permissions and
  njh.personnel = njh'.personnel and
  njh.projects = njh'.projects and
  njh.purposes = njh'.purposes and
  njh.qryItems = njh'.qryItems and
  njh.qualifiers = njh'.qualifiers and
  njh.queries = njh'.queries and
  njh.researchers = njh'.researchers and
  njh.retItems = njh'.retItems and
  njh.rules = njh'.rules and
  njh.sources = njh'.sources and
  njh.statuses = njh'.statuses and
  njh.transforms = njh'.transforms and
  njh.types = njh'.types and
  njh.values = njh'.values }

// the relations do not change
private pred noChangeRelations (njh, njh': NJH) {
  njh.ARAppliesTo = njh'.ARAppliesTo and
  njh.ARHides = njh'.ARHides and
  njh.ARTransforms = njh'.ARTransforms and
  njh.ATPriority = njh'.ATPriority and
```

```
   njh.dataAccessAgreement = njh'.dataAccessAgreement and
   njh.dataValues = njh'.dataValues and
   njh.enteredOn = njh'.enteredOn and
   njh.DICat= njh'.DICat and
   njh.DISource = njh'.DISource and
   njh.patientData = njh'.patientData and
   njh.permRules = njh'.permRules and
   njh.projectAT = njh'.projectAT and
   njh.projectDataCollector = njh'.projectDataCollector and
   njh.projectDataTransformRequired = njh'.projectDataTransformRequired and
   njh.projectPurpose = njh'.projectPurpose and
   njh.projectSources = njh'.projectSources and
   njh.projectPI = njh'.projectPI and
   njh.projectMembers = njh'.projectMembers and
   njh.projectQueries = njh'.projectQueries and
   njh.qryReturns = njh'.qryReturns and
   njh.qryWorksOn = njh'.qryWorksOn and
   njh.RDType = njh'.RDType and
   njh.resQualifier = njh'.resQualifier and
   njh.researcherL = njh'.researcherL and
   njh.supervisors = njh'.supervisors and
   njh.VDAllowed = njh'.VDAllowed }

private pred applyDecisionRules (
   njh: NJH,
   perm: Permission,
   rp: (Researcher + Project) ) {

   let
      team = rp.(njh.projectMembers),
      pi = rp.(njh.projectPI),
      dc = rp.(njh.projectDataCollector) ,
      sup = pi-> dc,
      dr = perm.(njh.permRules) & njh.decisionRules,
      pss = rp.(njh.projectSources) & Project |

      // 0. CanUseTotallyDeIDed decision rule is applicable
      (some dr & CanUseTotallyDeIDed implies (
         (some perm & DeIDed implies
            rp.(njh.projectDataTransformRequired) = TotallyDeIDed)
         and
         (some perm & Identified implies
            rp.(njh.projectDataTransformRequired) = NotTotallyDeIDed )))

   and

      // 1. DataAccessAgreementPresent decision rule is applicable
      (some dr & DataAccessAgreementPresent implies
      /* p1->p2 in njh.dataAccessAgreement means
         p1 gives p2 access to data produced by p1
      all the project's sources that are projects have a
         corresponding data agreement */
      all
         ps: pss | {
      // data access agreement is in place
      some ps -> rp & njh.dataAccessAgreement })

   and

      // 2. DataSourcePriorityOK decision rule is applicable
      (some dr & DataSourcePriorityOK implies
      all
```

```
      ps: pss |
    /* all the project's sources that are projects themselves
      each have an approved access ticket */
    some ps.(njh.projectAT) and
    /* if access ticket being considered has priority over
    the access tickets of any of its project sources
    (i.e. other projects) }then we cannot approve the
    project because the data returned would not be at
    the level required */
    no perm -> ps.(njh.projectAT) & njh.ATPriority )

  and

    // 3. NoSupsInPIandDC decision rule is applicable
    (some dr & NoSupsInPIandDC implies
      /* neither the pi nor the da supervise each other
      directly or indirectly */
      no (sup + ~sup ) & ^(njh.supervisors) )

  and

    // 4. PIDefined decision rule is applicable
    (some dr & PIDefined implies #pi > 0)

  and

    // 5. ProjectMembersDefined decision rule is applicable
    (some dr & ProjectMembersDefined implies #team > 0)

  and

    // 6. LicenedTeamAndPI decision rule is applicable
    (some dr & LicenedTeamAndPI implies (
      // each team member and pi has a Licence
    all
      r: (team + pi) | {
    some r.(njh.researcherL) }) )

  and

    // 7. NoOverlapPITeamDC decision rule is applicable
    (some dr & NoOverlapPITeamDC implies (
      // neither pi nor dc are a part of project team
      no (pi + dc) & team and

      // pi and da are not the same
      no pi & dc ) )

  and

    // 8. SomePurposeNotDirectTreatment decision rule is applicable
    (some dr & SomePurposeNotDirectTreatment implies (

      // purpose defined for project
      some rp.(njh.projectPurpose) and

      // purpose is not direct treatment
      rp.(njh.projectPurpose) != DirectTreatment ) )

  and

    // 9. QualifierPresent decision rule is applicable
```

```
      (some dr & QualifierPresent implies
        some rp.(njh.resQualifier) )

   and

      // 10. SomeQueriesDefined decision rule is applicable
      (some dr & SomeQueriesDefined implies
        some rp.(njh.projectQueries) )

   and

      // 11. SomeSourcesDefined decision rule is applicable
      (some dr & SomeSourcesDefined implies
        some rp.(njh.projectSources) ) }

/********** ********** ********** ********** **********
   OPERATION - skip
********** ********** ********** ********** **********/
pred skip (
      njh, njh': NJH) {

   noChangeSets[njh, njh'] and
      noChangeRelations[njh, njh'] }
run skip for 7but 15Rule, 1NJH expect 1

/********** ********** ********** ********** **********
   OPERATION - qualifyResearcher
********** ********** ********** ********** **********/
pred qualifyResearcher (
   njh, njh': NJH,
   res: Researcher,
   per: Personnel) {

   // preconditions */
   res in njh.researchers and
   per in njh.qualifiers and
   no res->per & njh.resQualifier and
   // adding this mapping does not make resQualifier reflexive
   irreflexive[^(res->per + njh.resQualifier)] and

   // set the qualifier for the reaearcher, postcondition */
   njh'.resQualifier = njh.resQualifier + res->per and

   // these do not change */
   noChangeSets[njh, njh'] and

   njh.ARAppliesTo = njh'.ARAppliesTo and
   njh.ARHides = njh'.ARHides and
   njh.ARTransforms = njh'.ARTransforms and
   njh.ATPriority = njh'.ATPriority and
   njh.dataAccessAgreement = njh'.dataAccessAgreement and
   njh.dataValues = njh'.dataValues and
   njh.enteredOn = njh'.enteredOn and
   njh.DICat= njh'.DICat and
   njh.DISource = njh'.DISource and
   njh.patientData = njh'.patientData and
   njh.permRules = njh'.permRules and
   njh.projectAT = njh'.projectAT and
   njh.projectDataCollector = njh'.projectDataCollector and
   njh.projectDataTransformRequired =
      njh'.projectDataTransformRequired and
   njh.projectPurpose = njh'.projectPurpose and
```

```
  njh.projectSources = njh'.projectSources and
  njh.projectPI = njh'.projectPI and
  njh.projectMembers = njh'.projectMembers and
  njh.projectQueries = njh'.projectQueries and
  njh.qryReturns = njh'.qryReturns and
  njh.qryWorksOn = njh'.qryWorksOn and
  njh.RDType = njh'.RDType and
  njh.researcherL = njh'.researcherL and
  njh.supervisors = njh'.supervisors and
  njh.VDAllowed = njh'.VDAllowed}
run qualifyResearcher for 7but 15Rule, 2NJH expect 1
run qualifyResearcher for 7but 15Rule, 1NJH expect 0

/********** ********** ********** ********** **********
 OPERATION - Approve Researcher's Licence
********** ********** ********** ********** **********/
pred approveResearcherL (
  njh, njh': NJH,
  res: Researcher,
  lic: Licence) {

  // preconditions
  res in njh.researchers and
  lic in njh.permissions and
  res->lic not in njh.researcherL and
  applyDecisionRules[njh, lic, res] and

  // set the access ticket for the reaearcher, postcondition
  njh'.researcherL = njh.researcherL + res->lic and

  //these do not change
  njh.ARAppliesTo = njh'.ARAppliesTo and
  njh.ARHides = njh'.ARHides and
  njh.ARTransforms = njh'.ARTransforms and
  njh.ATPriority = njh'.ATPriority and
  njh.dataAccessAgreement = njh'.dataAccessAgreement and
  njh.dataValues = njh'.dataValues and
  njh.enteredOn = njh'.enteredOn and
  njh.DICat= njh'.DICat and
  njh.DISource = njh'.DISource and
  njh.patientData = njh'.patientData and
  njh.permRules = njh'.permRules and
  njh.projectAT = njh'.projectAT and
  njh.projectDataCollector = njh'.projectDataCollector and
  njh.projectDataTransformRequired =
    njh'.projectDataTransformRequired and
  njh.projectPurpose = njh'.projectPurpose and
  njh.projectSources = njh'.projectSources and
  njh.projectPI = njh'.projectPI and
  njh.projectMembers = njh'.projectMembers and
  njh.projectQueries = njh'.projectQueries and
  njh.qryReturns = njh'.qryReturns and
  njh.qryWorksOn = njh'.qryWorksOn and
  njh.RDType = njh'.RDType and
  njh.resQualifier = njh'.resQualifier and
  njh.supervisors = njh'.supervisors and
  njh.VDAllowed = njh'.VDAllowed }
run approveResearcherL for 7but 15Rule, 3NJH expect 1
run approveResearcherL for 7but 15Rule, 2NJH expect 0

/********** ********** ********** ********** **********
  OPERATION - approve project's AT
```

```
********** ********** ********** ********** **********/
pred approveProjectAT(
   njh, njh': NJH,
   proj: Project, at: AccessTicket) {

   // preconditions
   proj in njh.projects and
   at in njh.permissions and
   no proj.(njh.projectAT) and

   applyDecisionRules[njh, at, proj ] and

   // set the access ticket for the project
   njh'.projectAT = njh.projectAT + proj -> at and

   //these do not change
   noChangeSets[njh, njh'] and

   njh.ARAppliesTo = njh'.ARAppliesTo and
   njh.ARHides = njh'.ARHides and
   njh.ARTransforms = njh'.ARTransforms and
   njh.ATPriority = njh'.ATPriority and
   njh.dataAccessAgreement = njh'.dataAccessAgreement and
   njh.dataValues = njh'.dataValues and
   njh.enteredOn = njh'.enteredOn and
   njh.DICat= njh'.DICat and
   njh.DISource = njh'.DISource and
   njh.patientData = njh'.patientData and
   njh.permRules = njh'.permRules and
   njh.projectDataCollector = njh'.projectDataCollector and
   njh.projectDataTransformRequired =
      njh'.projectDataTransformRequired and
   njh.projectPurpose = njh'.projectPurpose and
   njh.projectSources = njh'.projectSources and
   njh.projectPI = njh'.projectPI and
   njh.projectMembers = njh'.projectMembers and
   njh.projectQueries = njh'.projectQueries and
   njh.qryReturns = njh'.qryReturns and
   njh.qryWorksOn = njh'.qryWorksOn and
   njh.RDType = njh'.RDType and
   njh.resQualifier = njh'.resQualifier and
   njh.researcherL = njh'.researcherL and
   njh.supervisors = njh'.supervisors and
   njh.VDAllowed = njh'.VDAllowed }
/* since a project needs at least two researchers, i.e. PI and
   at one team member we need at least 5previous states
   to qualify the researchers and to approve their
   licences */
run approveProjectAT for 7but 15Rule, 6NJH expect 1
run approveProjectAT for 7but 15Rule, 5NJH expect 0

/********** ********** ********** ********** **********
   OPERATION - runQuery,
      researcher executes query
********** ********** ********** ********** **********/
private pred researcherAuthorisedForProject
      ( njh: NJH, res: Researcher, p: Project ) {
   // researcher is in the projectMembers, or is project's PI
   some (p.(njh.projectMembers) + p.(njh.projectPI)) & res }

private pred runQueryPre[
   njh: NJH, r: Researcher,
```

```
  p: Project, q: Query,
  at: AccessTicket] {

  // query is a part of the project's queries for the project
  q in p.(njh.projectQueries) and

  // at is the access ticket for the project
  some at & p.(njh.projectAT) and

  // researcher is authorised for the project
  researcherAuthorisedForProject[njh, r, p] and

  // since (we assume) Query has not yet been run
  no q.(njh.qryWorksOn) }

// Frame Conditions are post conditions
private pred runQueryPost[njh, njh':NJH, q: Query] {

  // operation does not change these sets
    njh.accessRules = njh'.accessRules and
    njh.accessTickets = njh'.accessTickets and
    njh.categories = njh'.categories and
    njh.consents = njh'.consents and
    njh.decisionRules = njh'.decisionRules and
    njh.hCats = njh'.hCats and
    njh.licences = njh'.licences and
    njh.patients = njh'.patients and
    njh.permissions = njh'.permissions and
    njh.personnel = njh'.personnel and
    njh.projects = njh'.projects and
    njh.purposes = njh'.purposes and
    njh.qualifiers = njh'.qualifiers and
    njh.queries = njh'.queries and
    njh.researchers = njh'.researchers and
    njh.rules = njh'.rules and
    njh.sources = njh'.sources and
    njh.statuses = njh'.statuses and
    njh.transforms = njh'.transforms and
    njh.types = njh'.types
  and
  // these relations do not change
    njh.ARAppliesTo = njh'.ARAppliesTo and
    njh.ARHides = njh'.ARHides and
    njh.ARTransforms = njh'.ARTransforms and
    njh.ATPriority = njh'.ATPriority and
    njh.dataAccessAgreement = njh'.dataAccessAgreement and
    njh.DICat= njh'.DICat and
    njh.DISource = njh'.DISource and
    njh.patientData = njh'.patientData and
    njh.permRules = njh'.permRules and
    njh.projectAT = njh'.projectAT and
    njh.projectDataCollector = njh'.projectDataCollector and
    njh.projectDataTransformRequired =
      njh'.projectDataTransformRequired and
    njh.projectPurpose = njh'.projectPurpose and
    njh.projectSources = njh'.projectSources and
    njh.projectPI = njh'.projectPI and
    njh.projectMembers = njh'.projectMembers and
    njh.projectQueries = njh'.projectQueries and
    njh.resQualifier = njh'.resQualifier and
    njh.researcherL = njh'.researcherL and
    njh.supervisors = njh'.supervisors and
```

```
      njh.VDAllowed = njh'.VDAllowed
   and

   /* operation changes these sets and relations
      these changes relate to changes in qryItems, and retItems
      and all that relate to them */

      let
         qItems = q.(njh'.qryWorksOn),
         qRetItems = dom[q.(njh'.qryReturns)],
         qDataItems = qRetItems+ qItems,
         qValues = qDataItems.(njh'.dataValues),
         qDates = Date & qDataItems.(njh'.dataValues) |

      // ********** for sets **********
      /* since we could be reusing dataitems in qDataItems using addition
         to specify the constraintis correct, njh' (post state) on the LHS */
      njh'.dataItems = njh.dataItems + qDataItems and
      njh'.qryItems = njh.qryItems + qItems and
      njh'.retItems = njh.retItems + qRetItems and
      njh'.dates = njh.dates + qDates and
      njh'.values = njh.values + qValues and

      // ********** for relations **********
      /* since qDataItems mappings to qry are new, using subtraction
         to specify the constraintis correct, njh' (post state) on the RHS */
      njh.qryReturns = njh'.qryReturns - q <: (njh'.qryReturns) and
      njh.qryWorksOn = njh'.qryWorksOn - q <: (njh'.qryWorksOn) and

      /* these could be reused from njh (pre state), so using addition
          to specify the constraint is correct, njh' (post state) on the LHS */
      njh'.dataValues = njh.dataValues + qDataItems <: njh'.dataValues and
      njh'.enteredOn = njh.enteredOn + qDataItems <: (njh'.enteredOn) and
      njh'.RDType = njh.RDType + q <: njh'.RDType }

private pred applyHidesAccessRules (
    njh: NJH, q: Query, qItems: set QryData,
   p: Project, at: AccessTicket, rules: set Rule) {

   // apply PatientConsent Rule
   ( some PatientConsent & rules implies
      qItems in (
      // dataitems from projectsources
         (njh.DISource).((njh.projectQueries.q).(njh.projectSources)) -
         // excluding dataItems where patients do not give consent
         dom[select23[njh.patientData] :>
            PatientConsent.(njh.ARHides)] )
   else
      qItems in
      // dataitems from projectsources
         (njh.DISource).((njh.projectQueries.q).(njh.projectSources)) ) }

private pred applyTransformAccessRules (
    njh: NJH, q: Query, qItems: set QryData,
   p: Project, at: AccessTicket, rules: set Rule) {

   let
      rItems = q.(njh.qryReturns).qItems |

   // apply DeIDedTransformHDate Rule
   (some rules & DeIDedTransformHDate implies (
   all
```

```
    ri: rItems | {
  let
    qis = ri.(q.(njh.qryReturns)) | {
  all
    qi: qis | {
   (some qi.(njh.DICat) & HDate implies
    (ri.(njh.dataValues) = DeIDedDateTransform[qi.(njh.dataValues)] and
    ri.(njh.enteredOn) = DeIDedDateTransform[qi.(njh.enteredOn)] )
   else
   // ri is not a date but the enteredOn needs de-identifying
    (ri.(njh.dataValues) = qi.(njh.dataValues) and
    ri.(njh.enteredOn) = DeIDedDateTransform[qi.(njh.enteredOn)] ) )
  and
  (#qis = 0iff no ri.(q.(njh.RDType)))
  and
  (#qis = 1iff ri.(q.(njh.RDType)) = Individual)
  and
  (#qis = 1iff ri.(q.(njh.RDType)) = Group) }}}
  ))

  and

  // apply IdentifiedDoesNotTransformHDate Rule
  (some rules & IdentifiedDoesNotTransformHDate implies (
  all
    ri: rItems | {
  let
    qis = ri.(q.(njh.qryReturns)) | {
  all
    qi: qis | {
   (ri.(njh.dataValues) = qi.(njh.dataValues) and
    ri.(njh.enteredOn) = qi.(njh.enteredOn) )
  and
  (#qis = 0iff no ri.(q.(njh.RDType)))
  and
  (#qis = 1iff ri.(q.(njh.RDType)) = Individual)
  and
  (#qis > 1iff ri.(q.(njh.RDType)) = Group) }}}
  )) }

private pred applyAccessRules (
  njh:NJH, p:Project,
  q: Query, at: AccessTicket ) {
  let
    qItems = q.(njh.qryWorksOn),
    rules = at.(njh.permRules) & njh.accessRules |

  applyHidesAccessRules[njh, q, qItems, p, at, rules] and
  applyTransformAccessRules[njh, q, qItems, p, at, rules] }

pred runQuery(
  njh, njh':NJH,
  r: Researcher, p: Project,
  q: Query, at: AccessTicket ) {

  // preconditions
  runQueryPre[njh, r, p, q, at] and
  // postconditions
  runQueryPost[njh, njh', q] and
  //  how changes are done, i.e. construct the return data
  applyAccessRules[njh', p, q, at] }
run runQuery for 7but 15Rule expect 1
```

```
run runQuery for 7but 15Rule, 6NJH expect 1// when qry works on no data
run runQuery for 7but 15Rule, 5NJH expect 0

private pred runQueryWithReturnData (
   njh, njh':NJH,
   r: Researcher, p: Project,
   q: Query, at: AccessTicket ) {

   runQuery[njh, njh', r, p, q, at] and
   some q.(njh'.qryReturns) }
run runQueryWithReturnData for 7but 15Rule, 7NJH expect 1


/********** ********** ********** ********** **********
   UpdateConformance
********** ********** ********** ********** **********/
pred updateConformance [
   njh, njh': NJH,
   qry: Query ] {

   // preconditions
   no qry.(njh.VDAllowed) and
   qry in njh.queries and

   // sequencing condition
   some qry.(njh.qryReturns) and

   // VDAllowed changes
   (invVDAllowed1[njh, qry] iff
      njh.VDAllowed = njh'.VDAllowed - qry -> DownloadAllowed) and
   (not invVDAllowed1[njh, qry] iff
      njh.VDAllowed = njh'.VDAllowed - qry -> DownloadDisabled )

   and

   noChangeSets[njh, njh'] and

   njh.ARAppliesTo = njh'.ARAppliesTo and
   njh.ARHides = njh'.ARHides and
   njh.ARTransforms = njh'.ARTransforms and
   njh.ATPriority = njh'.ATPriority and
   njh.dataAccessAgreement = njh'.dataAccessAgreement and
   njh.dataValues = njh'.dataValues and
   njh.enteredOn = njh'.enteredOn and
   njh.DICat= njh'.DICat and
   njh.DISource = njh'.DISource and
   njh.patientData = njh'.patientData and
   njh.permRules = njh'.permRules and
   njh.projectAT = njh'.projectAT and
   njh.projectDataCollector = njh'.projectDataCollector and
   njh.projectDataTransformRequired =
      njh'.projectDataTransformRequired and
   njh.projectPurpose = njh'.projectPurpose and
   njh.projectSources = njh'.projectSources and
   njh.projectPI = njh'.projectPI and
   njh.projectMembers = njh'.projectMembers and
   njh.projectQueries = njh'.projectQueries and
   njh.qryReturns = njh'.qryReturns and
   njh.qryWorksOn = njh'.qryWorksOn and
   njh.RDType = njh'.RDType and
   njh.resQualifier = njh'.resQualifier and
   njh.researcherL = njh'.researcherL and
```

```
    njh.supervisors = njh'.supervisors }
run updateConformance for 8but 15Rule expect 1
run updateConformance for 8but 15Rule, 7NJH expect 0
```

Listing A.5: *Full NJH structural model: adding LTL rules. imports Listing A.4 on line 11.*

```
1  /********** ********** ********** ********** ********** ********** **********
2   Sone note and to dos:
3    1.
4  ********** ********** ********** ********** ********** ********** **********/
5  module NJHgLTL
6
7  /********** ********** ********** ********** ********** ********** **********
8    IMPORTS
9  /********** ********** ********** ********** ********** ********** **********/
10 open util/ordering[NJH] as ord
11 open NJHgPM
12
13 /********** ********** ********** ********** ********** ********** **********
14   Simulating LTL and never claims -
15   These should follow from the model
16 /********** ********** ********** ********** ********** ********** **********/
17
18 /********** ********** ********** ********** **********
19   Check that we can both qualify and approve a
20   licence for a researcher
21
22   Verify that a Researcher always is qualified
23   before licence is approved
24 ********** ********** ********** ********** **********/
25 private pred
26   ltl_ApproveResLicenceAfterQualifyRes_ViableOnDifferentStates (
27   njh, njh', njh'', njh''': NJH,
28   res: Researcher,
29   lic: Licence, per: Personnel) {
30   let
31     first = ord/first |
32   some res & first.researchers and
33   some lic & first.permissions and
34   some per & first.personnel and
35   qualifyResearcher[njh, njh', res, per] and
36   approveResearcherL[njh'', njh''', res, lic] and
37   inv[njh] and
38   inv[njh'] and
39   inv[njh''] and
40   inv[njh'''] }
41
42 /* Is this the correct formulation for writing the LTL? */
43 assert ltl_ApproveResLicenceAfterQualifyRes {
44   some
45     njh, njh', njh'', njh''': NJH,
46     res: Researcher,
47     lic: Licence, per: Personnel |
48   (qualifyResearcher[njh, njh', res, per] and
49     approveResearcherL[njh'', njh''', res, lic] ) implies
50   ((njh + njh') in njh'''.prevs and
51   inv[njh] and
52   inv[njh'] and
53   inv[njh''] and
54   inv[njh''']) }
55
56 /********** ********** ********** ********** **********
57   Check that we can qualify a researcher,
58   approve a researcher's licence, approve
59   an access ticket for a project, and query that
```

```
  project

  Verify that if approving project access ticket
  and project members licence are successful,
  project members and pi licence are approved
  before the project's accessticket is approved.
********** ********** ********** ********** **********/
private pred
  ltl_ProjectApproveAfterTeamAndPILicenceApprove_viableOnDiffNJHStates (
  njh, njh', njh'', njh''', njh'''', njh''''': NJH,
  res: Researcher, lic: Licence, per: Personnel,
  proj: Project, at: AccessTicket) {
  let
    first = ord/first |
  some res & first.researchers and
  some lic & first.permissions and
  some per & first.personnel and
  some proj & first.projects and
  some at & first.permissions and

  qualifyResearcher[njh, njh', res, per] and
  approveResearcherL[njh'', njh''', res, lic] and
  approveProjectAT[njh'''', njh''''', proj, at]  and
  inv[njh] and
  inv[njh'] and
  inv[njh''] and
  inv[njh'''] and
  inv[njh''''] and
  inv[njh'''''] }

/* If both approveResearchL() for any researcher + PI and
  ApproveProjectAT() suceed for the same project we know that
  approveResearchL() suceeded in states previous to the final
  state for ApproveProjectAT(). */
assert ltl_ProjectApproveAfterTeamAndPILicenceApprove1 {
  some
    njh, njh', njh'', njh''': NJH,
    res: Researcher, lic: Licence,
    proj: Project, at: AccessTicket |
  ( res in (proj.(njh''.projectMembers) +
      proj.(njh''.projectPI)) and
    approveResearcherL[njh, njh', res, lic] and
    approveProjectAT[njh'', njh''', proj, at]) implies
  ( (njh + njh') in prevs[njh'''] and
  inv[njh] and
  inv[njh'] and
  inv[njh''] and
  inv[njh'''] ) }

/********** ********** ********** ********** **********
  Check that we can qualify a researcher,
  approve a researcher's licence, approve
  an access ticket for a project, and execute a
  query from the approved project.

  Verify that if qunning a query is successful
  then project's access ticket was approved in a
  state before the query was executable.
********** ********** ********** ********** **********/
private pred
  ltl_RunQueryWithOutQryReturnsAfterProjectApprove_viableOnDiffNJHStates (
  njh, njh', njh'', njh''', njh'''', njh''''', njh6, njh7: NJH,
```

```
  res: Researcher, lic: Licence, per: Personnel,
  proj: Project, at: AccessTicket,
  qry: Query) {
  let
    first = ord/first |
  some res & first.researchers and
  some lic & first.permissions and
  some per & first.personnel and
  some proj & first.projects and
  some at & first.permissions and
  some qry & first.queries and

  qualifyResearcher[njh, njh', res, per] and
  approveResearcherL[njh'', njh''', res, lic] and
  approveProjectAT[njh'''', njh''''', proj, at] and
  runQuery[njh6, njh7, res, proj, qry, at] and
  inv[njh] and
  inv[njh'] and
  inv[njh''] and
  inv[njh'''] and
  inv[njh''''] and
  inv[njh'''''] and
  inv[njh6] and
  inv[njh7] }

private pred
  ltl_RunQueryWithQryReturnsAfterProjectApprove_viableOnDiffNJHStates (
  njh, njh', njh'', njh''', njh'''', njh''''', njh6, njh7: NJH,
  res: Researcher, lic: Licence, per: Personnel,
  proj: Project, at: AccessTicket,
  qry: Query) {
  let
    first = ord/first |
  some res & first.researchers and
  some lic & first.permissions and
  some per & first.personnel and
  some proj & first.projects and
  some at & first.permissions and
  some qry & first.queries and

  // execute operations
  qualifyResearcher[njh, njh', res, per] and
  approveResearcherL[njh'', njh''', res, lic] and
  approveProjectAT[njh'''', njh''''', proj, at] and
  runQuery[njh6, njh7, res, proj, qry, at] and

  // we have some return data
  some qry.(njh7.qryReturns) and
  inv[njh] and
  inv[njh'] and
  inv[njh''] and
  inv[njh'''] and
  inv[njh''''] and
  inv[njh'''''] and
  inv[njh6] and
  inv[njh7] }

/* If both ApproveProjectAT() and RunQuery() succeed for the same
   project we know that ApproveProjectAT() suceeded in states
  previous to the final state for RunQuery(). */
assert ltl_RunQueryAfterProjectApprove1 {
  some
```

```
      njh, njh', njh'', njh''': NJH,
      res: Researcher, qry: Query,
      proj: Project, at: AccessTicket |
    ( res in (proj.(njh''.projectMembers) +
        proj.(njh''.projectPI)) and
    qry in proj.(njh'''.projectQueries) and
    approveProjectAT[njh, njh', proj, at] and
      runQuery[njh'', njh''', res, proj, qry, at] ) implies
    ((njh + njh') in prevs[njh'''] and
    inv[njh] and
    inv[njh'] and
    inv[njh''] and
    inv[njh'''] ) }


/********** ********** ********** ********** **********
   Check that we can qualify a researcher,
   approve a researcher's licence, approve
   an access ticket for a project, execute a
   query from the approved project, and update
   the query conformance.

   Verify that if updating a query conformance
   is successful then the corresponding running
   of the query to get the results was successful
   in a state before the update was executable.
********** ********** ********** ********** **********/
private pred
   ltl_UpdateConformanceAfterRunQuery_viableOnDiffNJHStates (
   njh, njh', njh'', njh''', njh'''', njh''''', njh6, njh7, njh8, njh9: NJH,
   res: Researcher, lic: Licence, per: Personnel,
   proj: Project, at: AccessTicket,
   qry: Query) {
   let
      first = ord/first |
   some res & first.researchers and
   some lic & first.permissions and
   some per & first.personnel and
   some proj & first.projects and
   some at & first.permissions and
   some qry & first.queries and

   qualifyResearcher[njh, njh', res, per] and
   approveResearcherL[njh'', njh''', res, lic] and
   approveProjectAT[njh'''', njh''''', proj, at] and
   runQuery[njh6, njh7, res, proj, qry, at] and
   updateConformance[njh8, njh9, qry] and
   inv[njh] and
   inv[njh'] and
   inv[njh''] and
   inv[njh'''] and
   inv[njh''''] and
   inv[njh'''''] and
   inv[njh6] and
   inv[njh7] and
   inv[njh8] and
   inv[njh9] }

assert ltl_UpdateConformanceAfterRunQuery {
   some
      njh, njh', njh'', njh''': NJH,
      res: Researcher, qry: Query,
```

```
      proj: Project, at: AccessTicket |
   ( res in (proj.(njh''.projectMembers) +
        proj.(njh''.projectPI)) and
   qry in proj.(njh'''.projectQueries) and
      runQuery[njh'', njh''', res, proj, qry, at] and
   updateConformance[njh'', njh''', qry] ) implies
   ((njh + njh') in prevs[njh'''] and
   inv[njh] and
   inv[njh'] and
   inv[njh''] and
   inv[njh'''] ) }


/********** ********** ********** ********** **********
   INV - predicates and functions
********** ********** ********** ********** **********/
// eventually will rename generator to inv
pred inv (njh: NJH) {
  // original generator predicate is true
   generator[njh] }

/********** ********** ********** ********** ********** ********** **********
   Checks to prove that each operation preserves the invariants
/********** ********** ********** ********** ********** ********** **********/
assert qualifyResearcherPreservesInv {
   all
      njh, njh': NJH,
      res: Researcher, per: Personnel |
   (inv[njh] and qualifyResearcher [njh, njh', res, per]) implies inv[njh'] }

assert approveResearcherLPreservesInv {
   all
      njh, njh': NJH ,
      res: Researcher, lic: Licence |
   (inv[njh] and approveResearcherL [njh, njh', res, lic]) implies inv[njh'] }

assert approveprojectATPreservesInv {
   all
      njh, njh': NJH,
      p: Project, at: AccessTicket |
   (inv[njh] and approveProjectAT [njh, njh', p, at]) implies inv[njh'] }

assert runQueryPreservesInv {
   all
      njh, njh': NJH,
      r: Researcher, q: Query, p: Project, at: AccessTicket |
   (inv[njh] and runQuery [njh, njh', r, p, q, at]) implies inv[njh'] }

assert skipPreservesInv {
   all
      njh, njh': NJH |
   (inv[njh] and skip [njh, njh']) implies inv[njh'] }

assert updateConformancePreservesInv {
   all
      njh, njh': NJH,
      q: Query |
   (inv[njh] and updateConformance [njh, njh', q]) implies inv[njh'] }


/********** ********** ********** ********** ********** ********** **********
   Conformance
```

```
/********** ********** ********** ********** ********** ********** **********/
/* an error occurs on this one, the problem may be because of the
   DStr data type dataitem */
assert Conformance {
   all
      njh: NJH,
      qry: Query,
       d: (Date & dom[qry.(njh.qryReturns)].(njh.dataValues)) +
             dom[qry.(njh.qryReturns)].(njh.enteredOn) |
   let
      at = (njh.projectQueries).qry.(njh.projectAT) |

   ((some qry -> DownloadAllowed & njh.VDAllowed and
      some qry.(njh.qryReturns) and
         some at & DeIDed) iff not identifiedDate[d] )
   or

   ((some qry -> DownloadAllowed & njh.VDAllowed and
      some qry.(njh.qryReturns) and
         some at & Identified ) iff identifiedDate[d] ) }

/********** ********** ********** ********** ********** ********** **********
   Executing the Predicates and Assertions
/********** ********** ********** ********** ********** ********** **********/

run
   ltl_ApproveResLicenceAfterQualifyRes_ViableOnDifferentStates
   for 8 but 15Rule, 3NJH expect 1
//should not be viable on < 3instances,
// i.e. need three distinct instances for both operations to succeed.
run
   ltl_ApproveResLicenceAfterQualifyRes_ViableOnDifferentStates
   for 8 but 15Rule, 2NJH expect 0

run
   ltl_ProjectApproveAfterTeamAndPILicenceApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 6NJH expect 1
// not viable on < 4instances,
// i.e. need four distinct instances for both operations to succeed.
run
   ltl_ProjectApproveAfterTeamAndPILicenceApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 5NJH  expect 0

// viable on four (4) states because query could return no results
run
   ltl_RunQueryWithOutQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 7NJH expect 1
// not viable on < 4instances,
// i.e. need three distince instances for both operations to succeed.
run
   ltl_RunQueryWithOutQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 6NJH expect 1
run
   ltl_RunQueryWithOutQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 5NJH expect 0
run
   ltl_RunQueryWithQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15Rule expect 1
run
   ltl_RunQueryWithQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15Rule, 7NJH expect 1
// not viable on < 4instances,
```

```
// i.e. need three distinct instances for both operations to succeed.
run
   ltl_RunQueryWithQryReturnsAfterProjectApprove_viableOnDiffNJHStates
   for 8 but 15 Rule, 6 NJH expect 0

run
   ltl_UpdateConformanceAfterRunQuery_viableOnDiffNJHStates
   for 8 but 15 Rule expect 1
run
   ltl_UpdateConformanceAfterRunQuery_viableOnDiffNJHStates
   for 8 but 15 Rule, 7 NJH expect 0

check ltl_ApproveResLicenceAfterQualifyRes for 8but 15Rule expect 0
check ltl_ProjectApproveAfterTeamAndPILicenceApprove1 for 8but 15Rule expect 0
check ltl_RunQueryAfterProjectApprove1 for 8but 15Rule expect 0
check ltl_UpdateConformanceAfterRunQuery for 8but 15Rule expect 0

//check qualifyResearcherPreservesInv for 8but 15Rule expect 0
//check approveResearcherLPreservesInv for 8but 15Rule expect 0
//check approveprojectATPreservesInv for 8but 15Rule expect 0
//check runQueryPreservesInv for 8but 15Rule expect 0
check skipPreservesInv for 8but 15Rule expect 0
//check updateConformancePreservesInv for 8but 15Rule expect 0

check Conformance for 8but 15Rule expect 0
```

B.1 Alloy Model Slice for the *Query* Operation

Listing B.1: *Slice 4: runQuery*Alloy Specifications

```
module NJH

/*
   ALLOY RELATION MODELLING REMINDER:
   the relation,
      AC: A some -> lone C
   means that in AC
      each A is linked to at most 1(lone) C, and
      each C is linked to at least one (some) A

    IMPORTANT Assumptions:
   1. access ticket for a project has already been granted;
   2. system ONLY issues DeIDed accesst tickets;
   3. we enforce in the CD and the Alloy model that a project has only can have
       one access ticket

    INDICATON of additional constraints:
    we use "// **" to identify constraints added to or removed from the Alloy
      model that are not currently in the CD.

   INTERPRETATION of the main assertions:
      OpPreserves and AlwaysDeIDedConformance
   A result of no counterexample found for OpPreserves and
      AlwaysDeIDedConformance is the result we require. However a no
      counterexample for both do not tell us the same things.
   OpPreserves tells us that operations pre- and post condition do not
      violate any of the constraints set.
   AlwaysDeIDedConformance tells us that the system constraints ensure
      conformance to the rules.
   So, the results could show that OpPreserves has no counterexample but
      AlwaysDeIDedConformance has a counterexample. This can be observed
      when AllDatesCorrectlyCategorised[...] is disabled in the inv[...] predicate.
*/

open util/relation
open util/ordering[NJH] as ord

sig DataSource, Day, Month, Name, Patient, Project, Query, Researcher, Year {}

abstract sig Type {}
lone sig Individual extends Type {}
// include when checking TransFormHDateAppliesToIndividual[njh]
//lone sig Group extends Type {}

abstract sig AccessTicket {}
lone sig DeIDed extends AccessTicket{}
// include when checking TransformHDateIsDeIDedRule[njh]
//lone sig LDS extends AccessTicket{}

sig DataItem {name: Name}
sig QryData, RetData extends DataItem {}
```

```

abstract sig Data{}
sig Date extends Data {
   day: lone Day,
   month: lone Month,
   year: Year
} {
   //  day iff month also exists
   some day implies some month
   some month implies some day }

abstract sig Rule {}
abstract sig AccessRule extends Rule {}
lone sig DeIDedTransformHDate extends AccessRule {}

abstract sig HIPAACat {}
lone sig HDate extends HIPAACat {}

sig NJH {
   // style is to alphabetise for easy finding :)

   // sets, creating a closed system
   accessRules: set AccessRule,
   accessTickets: set AccessTicket,
   dataItems: set DataItem,
   values: set Data,
   dates: set Date,
   hCats: set HIPAACat,
   patients: set Patient,
   projects: set Project,
   qryItems: set QryData,
   queries: set Query,
   researchers: set Researcher,
   retItems: set RetData,
   sources: set DataSource,
   types: set Type,

   // relations
   ARAppliesTo: accessRules -> some types,
   ARTransforms: accessRules -> some hCats,
   ATRules: accessTickets -> some accessRules,
   DataValues: dataItems -> one values,
   DICat: dataItems -> hCats,
   // ** no direct link between retItems and sources,
   // data sources of retItems are found through the RDFromQD relation
   DISource: (dataItems - retItems) -> one sources,
   EnteredOn: dataItems -> lone dates,
   // ** no direct link between retItems and patients,
   // patients associated with retItems are found through the RDFromQD relation
   PatientData: patients one -> some (dataItems - retItems),
   ProjAT: projects -> one accessTickets,
   ProjMembers: projects -> some researchers,
   ProjQueries: projects some -> some queries,
   ProjSources: projects -> some sources,
   // RunQuery specs require that a query have neither RetData nor QryData
   // before exexution, so we relax the multiplicity on the queries side
   QryReturns: queries -> retItems,
   QryWorksOn: queries -> qryItems,
   RDFromQD: retItems -> some qryItems,
   RDType: retItems -> one types
} {
   // CONSTRAINTS, comment out to check operation specifications
```

```
   // when commented out, it is enforced in the traces fact
   //inv[this]
}


////////////////////////////////////////////////////////////////////////////////
// INSTANCES
////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// - These predicates are not a part of the model and may be removed
// //////////

//////////////////////////////////////////////////////////////////////
// This predicate is a part of the model, used in init[...] to initialise the
//  system
// //////////
private pred ShowSomeOfEverything[njh: NJH] {
   some accessRules and
   some accessTickets and
   some dataItems and
   some values and
   some dates and
   some hCats and
   some patients and
   some projects and
   some qryItems and
   some queries and
   some researchers and
   some retItems and
   some sources and
   some types }
// important to run this with exactly 1NJH because the relations have the NJH
//   instance as their first element
//run ShowSomeOfEverything for 3but exactly 1NJH expect 1

////////////////////////////////////////////////////////////////////////////////
//  CONSTRAINTS as predicates
// //////////
private fun DeIDedDateTransform(d: Date): Date {
   {ri: Date |
      no ri.day and
      no ri.month and
      ri.year = d.year }}

 private pred QryRetDataDeIDed[njh: NJH, q: Query] {
   all qi: q.(njh.QryWorksOn) |
      some qi.(njh.DICat) & HDate
         implies ( // imp4
            // RetData
            (njh.RDFromQD).qi.(njh.DataValues) =
               DeIDedDateTransform[qi.(njh.DataValues)] and
               // if RetData EnteredOn exists
               (some (njh.RDFromQD).qi.(njh.EnteredOn)
                  implies ( // imp5
                     njh.RDFromQD).qi.(njh.EnteredOn) =
                        DeIDedDateTransform[qi.(njh.EnteredOn)]
                  ) //imp5
      ) // imp4
}

private pred DeIDedTransformHDateIndividual[njh: NJH, p: Project, q: Query] {
```

```
  // When a Query has RetData, this is how we construct it's return data and
  // its EntereOn Value
  (some q.(njh.QryReturns) and
    // query is a part of project
    some p.(njh.ProjQueries) & q and
      //  uses the DeIDed access ticket
      some p.(njh.ProjAT) & DeIDed and
        // DeIDed access ticket is associated with the TransformHDate rule
        TransformHDateIsDeIDedRule[njh] and
          // TransformHDate should be applied to individuals
          TransFormHDateAppliesToIndividual[njh])
    implies ( QryRetDataDeIDed[njh, q] )}


pred DeIDedTransformHDateIndividual[njh: NJH] {
  // When a Query has RetData, this is how we construct it's return data and
  // its EntereOn Value
  // this formulation works ONLY because the DeIDed is the ONLY access ticket
  //  in the system.
  all q: njh.queries |
    // if query returns values
    (some q.(njh.QryReturns) and
      //  uses the DeIDed access ticket
      some njh.ProjQueries.q.(njh.ProjAT) & DeIDed and
        // DeIDed access ticket is associated with the TransformHDate rule
        TransformHDateIsDeIDedRule[njh] and
          // TransformHDate should be applied to individuals
          TransFormHDateAppliesToIndividual[njh])
    implies ( QryRetDataDeIDed[njh, q] )}

private pred AllDatesCorrectlyCategorised [njh: NJH] {
  // correct formulation,
  // all dataItems in PatientData that are dates are identified as a HIPAACat
  all di: ran[njh.PatientData] |
    some di.(njh.DataValues) & Date implies some di.(njh.DICat) & HDate }

private pred TransformHDateIsDeIDedRule[njh: NJH] {
  some njh.ATRules & DeIDed -> DeIDedTransformHDate }

private pred TransFormHDateAppliesToIndividual[njh: NJH] {
  some njh.ARAppliesTo & DeIDedTransformHDate-> Individual }

// ** Defines additional constraints not in the UML CD
 pred inv [njh: NJH] {
  // all dataItems are mapped
  njh.dataItems =
    ran[njh.QryWorksOn] + ran[njh.QryReturns] + ran[njh.PatientData]

  // closed system constraint - any date is a part of the set of dates
  (njh.values & Date + ran[njh.EnteredOn]) = njh.dates

  // dataItems in Patient data
  all di: ran[njh.PatientData] | {
    // each has a date entered, we don't care if retItems are not in EnteredOn
    some di.(njh.EnteredOn)

    // each EnteredOn data has a day and month (constraint in Date signature
    //    ensures that month is non-empty iff day is non-empty)
    some di.(njh.EnteredOn.day)

    // each dataItem in PateintData has at most one HIPAACat
    #(di.(njh.DICat)) < 2}
```

```

   // queryData is patient data
   njh.qryItems & ran[njh.PatientData] = njh.qryItems

   // construct RDFromQD
   (~(njh.QryReturns)).(njh.QryWorksOn) = njh.RDFromQD

   all ri: dom[njh.RDFromQD] |
      // return data linked to the Individual type is only linked to one query data
      // in RDFromQD
      (some Individual & ri.(njh.RDType)) implies
         #(ri.(njh.RDFromQD)) = 1

   // a query's data source is contained in its project's sources
   all p: njh.projects, q: njh.queries | q in p.(njh.ProjQueries) implies
      q.(njh.QryWorksOn).(njh.DISource) in p.(njh.ProjSources)

   // Areas to seed for non-conformance
   // 1. TransformHDate rule for Individual Type,
   //    this is important when there are other access tickets other than DeIDed
   //    in the system
   TransformHDateIsDeIDedRule[njh]

   // 2. DeIDed access ticket has associated TransformHDate rule for Individuals,
   //    this is important when there are other types other than Individual in
   //    the system
   TransFormHDateAppliesToIndividual[njh]

   // 3. Ensure that all dataItems in PatientData that are dates are identified
   //    as a HIPAACat in DICat
   AllDatesCorrectlyCategorised[njh]
}


///////////////////////////////////////////////////////////////////////////////
// INSTANCES
///////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// - These predicates are not a part of the model and may be removed
// //////////
private pred ShowAny [ njh: NJH]{
   inv[njh]}
//run ShowAny for 3expect 1

private pred ShowProjQueryWithData [njh: NJH, q: Query]{
   inv[njh] and
      q in njh.queries and
         some q.(njh.QryWorksOn)}
//run ShowProjQueryWithData for 3but 1NJH expect 1

private pred ShowCheckingMultiplicities [njh: NJH, ar: AccessRule]{
   inv[njh] and
      ar in njh.accessRules and no ar.(njh.ARAppliesTo)}
//run ShowCheckingMultiplicities for 3but 1NJH expect 0

private pred ShowSomeOfEverythingWithHDateUnsetAndInv[njh: NJH, q: Query, qi: QryData] {
   q in njh.queries and
       qi in q.(njh.QryWorksOn) and
         no qi.(njh.DICat) and
            ShowSomeOfEverything[njh]
               and inv[njh] }
```

```
//  gives an instance only when
//    AllDatesCorrectlyCategorised[...] is disabled in inv[...]
//run ShowSomeOfEverythingWithHDateUnsetAndInv for 3but exactly 1NJH expect 0

private pred ShowSomeOfEverythingWithInv[njh: NJH] {
   ShowSomeOfEverything[njh] and inv[njh] }
//run ShowSomeOfEverythingWithInv for 3but exactly 1NJH expect 1

//////////////////////////////////////////////////////////////////////////////
// QUERY OPERATION SPECIFICATION
//////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// HELPER/USEFUL Predicates and Functions
// //////////
// not checking this predicate is a hidden path into executing RunQuery
private pred ResearcherAuthorisedToRunQuery
      [njh: NJH, res: Researcher, p: Project, qry: Query] {
   // query is associated with a project that the researcher is a member of
   some p.(njh.ProjMembers) & res and some p.(njh.ProjQueries) & qry }

// Helps the model to progress in traces
private pred NoChangeOp [njh, njh': NJH] {
   njh = njh'
   or ( //they both have the same sets and relations
      njh.accessRules = njh'.accessRules and
      njh.accessTickets =  njh'.accessTickets and
      njh.dataItems = njh'.dataItems and
      njh.values =  njh'.values and
      njh.dates = njh'.dates and
      njh.hCats = njh'.hCats and
      njh.patients = njh'.patients and
      njh.projects = njh'.projects and
      njh.qryItems = njh'.qryItems and
      njh.queries = njh'.queries and
      njh.researchers = njh'.researchers and
      njh.retItems = njh'.retItems and
      njh.sources = njh'.sources and
      njh.types = njh'.types and

      // relations
      njh.ARAppliesTo = njh'.ARAppliesTo and
      njh.ARTransforms = njh'.ARTransforms and
      njh.ATRules = njh'.ATRules and
      njh.DataValues = njh'.DataValues and
      njh.EnteredOn = njh'.EnteredOn and
      njh.DICat = njh'.DICat and
      njh.DISource = njh'.DISource and
      njh.PatientData = njh'.PatientData and
      njh.ProjAT =njh'.ProjAT and
      njh.ProjSources = njh'.ProjSources and
      njh.ProjMembers = njh'.ProjMembers and
      njh.ProjQueries = njh'.ProjQueries and
      njh.QryReturns = njh'.QryReturns and
      njh.QryWorksOn = njh'.QryWorksOn and
      njh.RDFromQD = njh'.RDFromQD and
      njh.RDType = njh'.RDType) }

private pred RunQueryPre[njh: NJH, r: Researcher, p: Project, q: Query] {
      // in sets
      q in njh.queries and
      r in njh.researchers and
```

```

    // in relations
    ResearcherAuthorisedToRunQuery[njh, r, p, q] and
    // since (we assume) Query has not yet been run
    no q.(njh.QryWorksOn) }

private pred RunQueryPost[njh, njh':NJH, q: Query] {
  // Frame Conditions are post conditions
  // frame conditions - no change
  {
    // sets
    njh.accessRules = njh'.accessRules and
    njh.accessTickets =  njh'.accessTickets and
    njh.hCats = njh'.hCats and
    njh.patients = njh'.patients and
    njh.projects = njh'.projects and
    njh.queries = njh'.queries and
    njh.researchers = njh'.researchers and
    njh.sources = njh'.sources and
    njh.types = njh'.types and

    // relations
    njh.ARAppliesTo = njh'.ARAppliesTo and
    njh.ARTransforms = njh'.ARTransforms and
    njh.ATRules = njh'.ATRules and
    njh.DICat = njh'.DICat and
    njh.DISource = njh'.DISource and
    njh.PatientData = njh'.PatientData and
    njh.ProjAT =njh'.ProjAT and
    njh.ProjSources = njh'.ProjSources and
    njh.ProjMembers = njh'.ProjMembers and
    njh.ProjQueries = njh'.ProjQueries }

  and

  // frame conditions - changes
  {
    // to sets
    njh.dataItems = njh'.dataItems - q.(njh'.QryReturns) and
    njh.values in njh'.values
    njh.dates in njh'.dates and
    njh.qryItems in njh'.qryItems and
    njh.retItems = njh'.retItems and

    // to relations
    // these changes relate to changes in qryItems, and retItems
    njh.DataValues = njh'.DataValues - q.(njh'.QryReturns) <: njh'.DataValues //and
    njh.EnteredOn = njh'.EnteredOn - q.(njh'.QryReturns) <: (njh'.EnteredOn) and
    njh.QryReturns = njh'.QryReturns - q <: (njh'.QryReturns) and
    njh.QryWorksOn = njh'.QryWorksOn - q <: (njh'.QryWorksOn) and
    njh.RDFromQD in njh'.RDFromQD and
    njh.RDType = njh'.RDType - q.(njh'.QryReturns) <: njh'.RDType}
}

private pred RunQueryOutput[ njh, njh':NJH, p:Project, q: Query] {
  // frame postconditions
  RunQueryPost[njh, njh', q] and
  // currently these are a part of the invariants
  //   (see call to ConstructDeIDedReturnData[...] in inv[...] )
  //- enforced in the traces fact but could be extracted to here
  DeIDedTransformHDateIndividual[njh', p, q] }

```

```
// formulation is where a query has one access ticket through the project and
// project has exactly one access ticket
// preconditions and (All?) frame conditions can be automatically generated!
private pred runQuery[njh, njh':NJH, r: Researcher, p: Project, q: Query] {
   // preconditions
   RunQueryPre[njh, r, p, q] and
   //  how changes are done, i.e. construct the return data
   RunQueryOutput[njh, njh', p, q] }

/////////////////////////////////////////////////////////////////////
// Operation Specifications
// Operation specifications does not ensure Conformance!!!
// //////////

// this is how we initialise the system
pred init[njh: NJH] {
   some q: Query |
      q in njh.queries and
         // all the sets except qryItems and retItems are are non-empty
         ShowSomeOfEverything[njh] and
            // instance does not violate constraints
            inv[njh] and
               //the query in question is the one we want to check the operation specifications for
               no q.(njh.QryWorksOn)}
//run init for 3but exactly 1NJH expect 1

// this is how we move from instance to instance
fact traces {
   init[ord/first]
   all njh: NJH - ord/last, r: Researcher, q: Query, p: Project |
      let njh' = njh.next |
         runQuery[njh, njh', r, p, q] or NoChangeOp[njh, njh'] }


//  END OF THE MODEL and RunQuery specification
////////////////////////////////////////////////////////////////////-

////////////////////////////////////////////////////////////////////-
// SOME OPERATION SPECIFICATIONS CHECKS
// //////////
// verify that operations preserve the invariants
// also a way for possible hidden paths to exist
assert OpPreserves {
   all njh, njh': NJH |
      all r: Researcher, q: Query, p: Project |
         (inv[njh] and runQuery [njh, njh', r, p, q]) implies inv[njh'] }
// after a scope of 4, the checking takes too long, i.e. > 170secs
check OpPreserves for 4expect 0

// run only when opPreserves returns a counterexample
pred OpDoesNotPreserve[njh, njh': NJH, r: Researcher, p: Project, q: Query ]{
   inv[njh] and runQuery[njh, njh', r,p, q] and not inv[njh'] }
run OpDoesNotPreserve for 3but exactly 2NJH expect 0


///////////////////////////////////////////////////////////////////////////////
// CHECKING THE MODEL FOR CONFORMANCE
///////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// HELPER/USEFUL Predicates and Functions to check conformance
```

```
// these are not used in the model
// //////////
private pred ConformanceDeIDedHDateUnSet
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
   not HDateSet[njh, qi] and
   not IdentifiedDate[ri.(njh.DataValues)] }

private pred ConformanceDeIDedHDateUnSetFullDate
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
   FullDateConditions[njh, qi] and
   not HDateSet[njh, qi] and
   not IdentifiedDate[ri.(njh.DataValues)] }

// since there should be no instance where qi's datavalue that is a date is not
// marked as a HDate, we expect to see no instances from running these
// two predicates when there is system conformance
//run ConformanceDeIDedHDateUnSet for 3but 1NJH expect 0
//run ConformanceDeIDedHDateUnSetFullDate for 3but 1NJH expect 0

// useful to check if Data Deided properly
private pred NonConformanceDeIDedFullDateHDateSet
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
   FullDateConditions[njh, qi] and
   HDateSet[njh, qi] and
   IdentifiedDate[ri.(njh.DataValues)] }

// expect no instances from this predicate when there is system conformance
//run NonConformanceDeIDedFullDateHDateSet for 3but 1NJH expect 0

//////////////////////////////////////////////////////////////////
// HELPER/USEFUL Predicates and Functions to check conformance
// these are needed in the model
// //////////

// these predicates help to check conformance
// //////////
private pred IdentifiedDate[d: Date] {some d.day }

private pred BasicDeIdentifiedDateConditions
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   // constraints hold
   inv[njh] and

   // qry is in the NJH system of interest
   qry in njh.queries and

   // query has DeIDed access as a part of a project
   some (njh.ProjQueries).qry.(njh.ProjAT) & DeIDed and

   // query has some data
   qi in qry.(njh.QryWorksOn) and

   // QryData qi is a Date
   some qi.(njh.DataValues) & Date and

   // query returns some Data
   ri in qry.(njh.QryReturns) and

   // Date data for QryWorksOn is identified data
```

```
   IdentifiedDate[qi.(njh.DataValues)] and

   // the RetDdata we are interested in is for the QryData qi
   ri = njh.RDFromQD.qi and

   // When a Query has RetData, this is how we construct it's return data for
   //    the DeIDed access ticket for the individual category
   DeIDedTransformHDateIndividual[njh]
}

private pred FullDateConditions [njh: NJH, qi: QryData ] {
   some qi.(njh.DataValues).day }

private pred HDateSet[njh: NJH, qi: QryData] {some qi.(njh.DICat) & HDate }

// these predicates check conformance under certain conditions
// //////////

pred CanGetConformanceDeIDed
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri]
         and not IdentifiedDate[ri.(njh.DataValues)] }
// give me a system where some return data is de-identified
run CanGetConformanceDeIDed for 3but 1NJH expect 1

private pred ConformanceDeIDed
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri]
         implies not IdentifiedDate[ri.(njh.DataValues)] }
// give me a system where all the return data is de-identified
//run ConformanceDeIDed for 3but 1NJH expect 1

private pred ConformanceDeIDedHDateSet
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   (BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
      HDateSet[njh, qi])
         implies not IdentifiedDate[ri.(njh.DataValues)] }

private pred ConformanceDeIDedHDateSetFullDate
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
   FullDateConditions[njh, qi] and
   HDateSet[njh, qi] and
   not IdentifiedDate[ri.(njh.DataValues)] }

// We can get instances from this predicate even when there is non-conformance
//run ConformanceDeIDedHDateSet for 3but 1NJH expect 1
//run ConformanceDeIDedHDateSetFullDate for 3but 1NJH expect 1

private pred NonConformanceDeIDedFullDateHDateUnSet
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   BasicDeIdentifiedDateConditions[njh, qry, qi, ri] and
   FullDateConditions[njh, qi] and
   not HDateSet[njh, qi] and
   IdentifiedDate[ri.(njh.DataValues)] }

// expect no instances from this predicate when there is system conformance
//  NonConformanceDeIDedFullDateHDateUnSet[..] gives an instance only when
//    AllDatesCorrectlyCategorised[...] is disabled in inv[...]
//run NonConformanceDeIDedFullDateHDateUnSet for 3but 1NJH expect 0

//////////////////////////////////////////////////////////////////////
```

```
// ACTUAL CONformance verification, predicate here is public,
// run predicate DeIDedNonConformanceFullDateWhenHDateUnSet only
// when AlwaysDeIDedConformanceWhenHDateUnSet[..] returns a
// counterexample
// //////////

// Verifies that in all instances the return data is always de-identified
// a counterexample may mean partial conformance
assert AlwaysDeIDedConformance{
   all njh: NJH, q: njh.queries |
      all qi: q.(njh.QryWorksOn), ri: q.(njh.QryReturns) |
         ConformanceDeIDed[njh, q, qi, ri] }
check AlwaysDeIDedConformance for 3expect 0

// if all a system's return data is not de-identified, we check the reason,
// Reason: HDate is set fo ra dataitem that is a date so it means the Date
//    was not deidentified properly
// a counterexample may mean partial conformance
assert AlwaysDeIDedConformanceWhenHDateSet {
   all njh: NJH, q: njh.queries |
      all qi: q.(njh.QryWorksOn), ri: q.(njh.QryReturns) |
          ConformanceDeIDedHDateSet[njh, q, qi, ri] }
check AlwaysDeIDedConformanceWhenHDateSet for 3expect 0

// if all a system's return data is not de-identified, we check the reason,
//  Reason: a dataitem that is a date was not categorised as a HDate
// a counterexample may mean partial conformance
assert AlwaysDeIDedConformanceWhenHDateUnSet{
   all njh: NJH, q: njh.queries |
      all qi: q.(njh.QryWorksOn), ri: q.(njh.QryReturns) |
         not NonConformanceDeIDedFullDateHDateUnSet[njh, q, qi, ri] }
check AlwaysDeIDedConformanceWhenHDateUnSet for 3expect 0

// show example where a system return data is not de-identified because a
//    dataitem that is a date id not categorised as a HDate
// an instance means this could be one of the reasons for the non-conformance
pred DeIDedNonConformanceFullDateWhenHDateUnSet
      [njh: NJH, qry: Query, qi: QryData, ri: RetData ] {
   NonConformanceDeIDedFullDateHDateUnSet [njh, qry, qi, ri]}
run DeIDedNonConformanceFullDateWhenHDateUnSet for 3but 1NJH expect 0
```

B.2 Important Model Checks

Table B.1 describes the predicates and assertions we added to the *runQuery* Alloy model to extract model properties of interest. The most important results come from *OpPreserves*, *CanGetConformanceDeIDed* and *AlwaysDeIDedConformance*. A point worth mentioning is that *CanGetConformanceDeIDed* can give instances whether or not *OpPreserves* or *AlwaysDeIDedConformance* find counterexamples. We include both the *OpDoesNotPreserve* and *DeIDedNonConformanceFullDateWhenHDateUnSet* predicates as alternates to finding instances where the main assertions find counterexamples, because the assertions have much longer running times that probing the model for an instance when the assertions already produced counterexamples.

Table B.1: Important Model Checks for the *runQuery* method

Name	Type	Explanation	Result	State
OpPreserves	Assertion	Asserts that the *Query* operation specifications never cause the constraints we set up in *inv[...]* predicate to be violated	No counterexample expected	N/A
OpDoesNotPreserve	Predicate	Gives an instance under which the *runQuery* operation violates the constraints	No instance expected when *OpPreserves* gives no counterexamples	N/A
CanGetConformanceDeIDed	Predicate	Gives an instance to show that we can generate an instance in which data returned by a query is de-identified as expected	An instance is expected	instance means *De-identified* state present
AlwaysDeIDedConformance	Assertion	Asserts that under all circumstances, all query results using a de-identified access ticket are always de-identified	No counterexample expected	counterexample means *Identified* state present
AlwaysDeIDedConformanceWhenHDateSet	Assertion	Asserts that when we identify patient data with a HIPAA date category query results using a de-identified access ticket are never identified. Used to further probe the model when *AlwaysDeIDedConformance* gives a counterexample.	No counterexample expected	counterexample means *Identified* state present
AlwaysDeIDedConformanceWhenHDateUnSet	Assertion	Asserts that when we do not identify patient data with a HIPAA date category query results using a de-identified access ticket are never identified. Used to further probe the model when *AlwaysDeIDedConformance* gives a counterexample.	No counterexample expected	counterexample means *Identified* state present
DeIDedNonConformanceFullDateWhenHDateUnSet	Predicate	Gives and instance where query results using a de-identified access ticket are identified on patient data with the HIPAA date category not set but should have been set	No instance expected; no instance if state is aways *De-identified*	instance means *Identified* state present

C.1 Counterexample in the *CheckConformance* Operation

C.1.1 Slice 5: Alloy Specifications

The specifications are included in Section D.1.2.

C.1.2 Slice 5: Alloy Counterexample XML representation

See Figure 7.15 for a graphical representation of the Alloy Analyzer counterexample. Source in xml file removed as Alloy model is given in another appendix.

Listing C.1: *Slice 5: CheckConformance* XML Counterexample

```
<alloy builddate="2014-05-16 16:44 EDT">

<instance bitwidth="0" maxseq="0" command="Run showDeIDedDD for 7but 1NJH expect 1"
    filename="slice_5_g_inst.als">

<sig label="seq/Int" ID="0" parentID="1" builtin="yes">
</sig>

<sig label="Int" ID="1" parentID="2" builtin="yes">
</sig>

<sig label="String" ID="3" parentID="2" builtin="yes">
</sig>

<sig label="this/Date" ID="4" parentID="5">
   <atom label="Date$0"/>
   <atom label="Date$1"/>
</sig>

<field label="day" ID="6" parentID="4">
   <tuple> <atom label="Date$1"/> <atom label="Day$2"/> </tuple>
   <types> <type ID="5"/> <type ID="7"/> </types>
</field>

<field label="month" ID="8" parentID="4">
   <tuple> <atom label="Date$1"/> <atom label="Month$0"/> </tuple>
   <types> <type ID="5"/> <type ID="9"/> </types>
</field>

<field label="year" ID="10" parentID="4">
   <tuple> <atom label="Date$0"/> <atom label="Year$0"/> </tuple>
   <tuple> <atom label="Date$1"/> <atom label="Year$0"/> </tuple>
   <types> <type ID="5"/> <type ID="11"/> </types>
</field>

<sig label="this/Data" ID="5" parentID="2" abstract="yes">
</sig>

<sig label="this/Project" ID="12" parentID="13">
   <atom label="Project$0"/>
</sig>

<sig label="this/DataSource" ID="13" parentID="2" abstract="yes">
</sig>

<sig label="this/AllowDeIDed" ID="14" parentID="15" one="yes">
   <atom label="AllowDeIDed$0"/>
</sig>

<sig label="this/TotallyDeIDed" ID="16" parentID="15" one="yes">
   <atom label="TotallyDeIDed$0"/>
```

```
</sig>

<sig label="this/TotallyIDed" ID="17" parentID="15" one="yes">
   <atom label="TotallyIDed$0"/>
</sig>

<sig label="this/DataTransform" ID="15" parentID="2" abstract="yes">
</sig>

<sig label="this/Age" ID="18" parentID="19">
   <atom label="Age$0"/>
</sig>

<sig label="this/Other" ID="20" parentID="19">
   <atom label="Other$0"/>
</sig>

<sig label="this/Name" ID="19" parentID="2" abstract="yes">
</sig>

<sig label="this/DeIDed" ID="21" parentID="22" lone="yes">
   <atom label="DeIDed$0"/>
</sig>

<sig label="this/Identified" ID="23" parentID="22" lone="yes">
   <atom label="Identified$0"/>
</sig>

<sig label="this/AccessTicket" ID="22" parentID="24" abstract="yes">
</sig>

<sig label="this/Permission" ID="24" parentID="2" abstract="yes">
</sig>

<sig label="this/DownloadAllowed" ID="25" parentID="26" lone="yes">
   <atom label="DownloadAllowed$0"/>
</sig>

<sig label="this/DownloadDisabled" ID="27" parentID="26" lone="yes">
   <atom label="DownloadDisabled$0"/>
</sig>

<sig label="this/Status" ID="26" parentID="2" abstract="yes">
</sig>

<sig label="this/Day" ID="7" parentID="2">
   <atom label="Day$0"/>
   <atom label="Day$1"/>
   <atom label="Day$2"/>
</sig>

<sig label="this/Month" ID="9" parentID="2">
   <atom label="Month$0"/>
</sig>

<sig label="this/Query" ID="28" parentID="2">
   <atom label="Query$0"/>
   <atom label="Query$1"/>
   <atom label="Query$2"/>
</sig>

<sig label="this/Year" ID="11" parentID="2">
```

```
    <atom label="Year$0"/>
</sig>

<sig label="this/DataItem" ID="29" parentID="2">
    <atom label="DataItem$0"/>
    <atom label="DataItem$1"/>
    <atom label="DataItem$2"/>
    <atom label="DataItem$3"/>
    <atom label="DataItem$4"/>
    <atom label="DataItem$5"/>
    <atom label="DataItem$6"/>
</sig>

<field label="name" ID="30" parentID="29">
    <tuple> <atom label="DataItem$0"/> <atom label="Other$0"/> </tuple>
    <tuple> <atom label="DataItem$1"/> <atom label="Age$0"/> </tuple>
    <tuple> <atom label="DataItem$2"/> <atom label="Other$0"/> </tuple>
    <tuple> <atom label="DataItem$3"/> <atom label="Age$0"/> </tuple>
    <tuple> <atom label="DataItem$4"/> <atom label="Age$0"/> </tuple>
    <tuple> <atom label="DataItem$5"/> <atom label="Age$0"/> </tuple>
    <tuple> <atom label="DataItem$6"/> <atom label="Other$0"/> </tuple>
    <types> <type ID="29"/> <type ID="19"/> </types>
</field>

<sig label="this/NJH" ID="31" parentID="2">
    <atom label="NJH$0"/>
</sig>

<field label="accessTickets" ID="32" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Identified$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DeIDed$0"/> </tuple>
    <types> <type ID="31"/> <type ID="24"/> </types>
</field>

<field label="dataItems" ID="33" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$2"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$3"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$4"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$5"/> </tuple>
    <types> <type ID="31"/> <type ID="29"/> </types>
</field>

<field label="dates" ID="34" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Date$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="Date$1"/> </tuple>
    <types> <type ID="31"/> <type ID="5"/> </types>
</field>

<field label="permissions" ID="35" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Identified$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DeIDed$0"/> </tuple>
    <types> <type ID="31"/> <type ID="24"/> </types>
</field>

<field label="projects" ID="36" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> </tuple>
    <types> <type ID="31"/> <type ID="13"/> </types>
</field>

<field label="qryItems" ID="37" parentID="31">
```

```
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$2"/> </tuple>
    <types> <type ID="31"/> <type ID="29"/> </types>
</field>

<field label="queries" ID="38" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Query$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="Query$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="Query$2"/> </tuple>
    <types> <type ID="31"/> <type ID="28"/> </types>
</field>

<field label="retItems" ID="39" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$3"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$4"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$5"/> </tuple>
    <types> <type ID="31"/> <type ID="29"/> </types>
</field>

<field label="statuses" ID="40" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="DownloadDisabled$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DownloadAllowed$0"/> </tuple>
    <types> <type ID="31"/> <type ID="26"/> </types>
</field>

<field label="transforms" ID="41" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="AllowDeIDed$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="TotallyDeIDed$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="TotallyIDed$0"/> </tuple>
    <types> <type ID="31"/> <type ID="15"/> </types>
</field>

<field label="values" ID="42" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Date$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="Date$1"/> </tuple>
    <types> <type ID="31"/> <type ID="5"/> </types>
</field>

<field label="dataValues" ID="43" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$0"/> <atom label="Date$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$1"/> <atom label="Date$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$2"/> <atom label="Date$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$3"/> <atom label="Date$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$4"/> <atom label="Date$1"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$5"/> <atom label="Date$1"/> </tuple>
    <types> <type ID="31"/> <type ID="29"/> <type ID="5"/> </types>
</field>

<field label="enteredOn" ID="44" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$3"/> <atom label="Date$0"/> </tuple>
    <tuple> <atom label="NJH$0"/> <atom label="DataItem$4"/> <atom label="Date$0"/> </tuple>
    <types> <type ID="31"/> <type ID="29"/> <type ID="5"/> </types>
</field>

<field label="projectAT" ID="45" parentID="31">
    <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> <atom label="DeIDed$0"/> </tuple>
    <types> <type ID="31"/> <type ID="13"/> <type ID="24"/> </types>
</field>

<field label="projectDataTransformRequired" ID="46" parentID="31">
```

```
  <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> <atom label="TotallyDeIDed$0"/> </tuple>
  <types> <type ID="31"/> <type ID="13"/> <type ID="15"/> </types>
</field>

<field label="projectQueries" ID="47" parentID="31">
  <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> <atom label="Query$0"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> <atom label="Query$1"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Project$0"/> <atom label="Query$2"/> </tuple>
  <types> <type ID="31"/> <type ID="13"/> <type ID="28"/> </types>
</field>

<field label="qryReturns" ID="48" parentID="31">
  <tuple> <atom label="NJH$0"/> <atom label="Query$0"/> <atom label="DataItem$0"/> <atom label="DataItem$2"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Query$0"/> <atom label="DataItem$3"/> <atom label="DataItem$1"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Query$1"/> <atom label="DataItem$4"/> <atom label="DataItem$1"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Query$2"/> <atom label="DataItem$5"/> <atom label="DataItem$1"/> </tuple>
  <types> <type ID="31"/> <type ID="28"/> <type ID="29"/> <type ID="29"/> </types>
</field>

<field label="VDAllowed" ID="49" parentID="31">
  <tuple> <atom label="NJH$0"/> <atom label="Query$0"/> <atom label="DownloadDisabled$0"/> </tuple>
  <tuple> <atom label="NJH$0"/> <atom label="Query$2"/> <atom label="DownloadDisabled$0"/> </tuple>
  <types> <type ID="31"/> <type ID="28"/> <type ID="26"/> </types>
</field>

<sig label="ord/Ord" ID="50" parentID="2" one="yes" private="yes">
  <atom label="ord/Ord$0"/>
</sig>

<field label="First" ID="51" parentID="50" private="yes">
  <tuple> <atom label="ord/Ord$0"/> <atom label="NJH$0"/> </tuple>
  <types> <type ID="50"/> <type ID="31"/> </types>
</field>

<field label="Next" ID="52" parentID="50" private="yes">
  <types> <type ID="50"/> <type ID="31"/> <type ID="31"/> </types>
</field>

<sig label="univ" ID="2" builtin="yes">
</sig>

<sig label="this/QryData" ID="53">
  <atom label="DataItem$1"/>
  <atom label="DataItem$2"/>
  <atom label="DataItem$3"/>
  <atom label="DataItem$4"/>
  <atom label="DataItem$5"/>
  <atom label="DataItem$6"/>
  <type ID="29"/>
</sig>

<sig label="this/RetData" ID="54">
  <atom label="DataItem$0"/>
  <atom label="DataItem$3"/>
  <atom label="DataItem$4"/>
```

```
    <atom label="DataItem$5"/>
    <atom label="DataItem$6"/>
    <type ID="29"/>
</sig>

<skolem label="$init_q" ID="55">
    <tuple> <atom label="Query$1"/> </tuple>
    <types> <type ID="28"/> </types>
</skolem>

<skolem label="$showDeIDedDD_njh" ID="56">
    <tuple> <atom label="NJH$0"/> </tuple>
    <types> <type ID="31"/> </types>
</skolem>

<skolem label="$showDeIDedDD_p" ID="57">
    <tuple> <atom label="Project$0"/> </tuple>
    <types> <type ID="13"/> </types>
</skolem>

<skolem label="$showDeIDedDD_q" ID="58">
    <tuple> <atom label="Query$2"/> </tuple>
    <types> <type ID="28"/> </types>
</skolem>

<skolem label="$common_inst_p" ID="59">
    <tuple> <atom label="Project$0"/> </tuple>
    <types> <type ID="13"/> </types>
</skolem>

<skolem label="$common_inst_q" ID="60">
    <tuple> <atom label="Query$2"/> </tuple>
    <types> <type ID="28"/> </types>
</skolem>

<skolem label="$totallyDeIDedTransform_d" ID="61">
    <tuple> <atom label="Date$1"/> </tuple>
    <types> <type ID="5"/> </types>
</skolem>

</instance>

</alloy>
```

C.1.3 Slice 5: Alloy Counterexample USE representation (see Figure 7.17 for a graphical representation of the object model)

Listing C.2: *Slice 5: CheckConformance* USE Counterexample

```
-- Script generated by USE 4.2.0

!new DownloadDisabled('DownloadDisabled_0')
!new DeIDed('DeIDed_0')

!new QryData('DataItem_4')
!new QryData('DataItem_5')

!DataItem_4.name := 'Age'
!DataItem_5.name := 'Other'

!new Date('Date_1')
!Date_1.day := 9
!Date_1.month := 8
!Date_1.year := 1931

!insert (DataItem_5,Date_1) into DataValues
!insert (DataItem_4,Date_1) into DataValues

!new Project('Project_1')
!new Query('Query_0')
!insert (Project_1,DeIDed_0) into ProjectAT
!insert (Project_1,Query_0) into ProjectQueries

!new RetData('DataItem_0')
!new RetData('DataItem_1')
!new RetData('DataItem_2')
!new RetData('DataItem_3')

!new Date('Date_0')
!Date_0.day := 0
!Date_0.month := 0
!Date_0.year := 1931

!DataItem_0.name := 'Age'
!insert (Query_0,DataItem_0,DataItem_4) into QryReturns
!insert (DataItem_0,Date_0) into DataValues

!DataItem_3.name := 'Other'
!insert (Query_0,DataItem_3,DataItem_5) into QryReturns
!insert (DataItem_3,Date_1) into DataValues

!DataItem_2.name := 'Age'
!insert (Query_0,DataItem_2,DataItem_4) into QryReturns
!insert (DataItem_2,Date_0) into DataValues

!DataItem_1.name := 'Age'
!insert (Query_0,DataItem_1,DataItem_4) into QryReturns
!insert (DataItem_1,Date_0) into DataValues

!insert (Query_0,DownloadDisabled_0) into VDAllowed
```

C.2 USE Commands for Generating On-Demand Object Models in the NJH System

The listings in sections C.2.1 through C.2.4 are used in the listings in Section C.2.5.

C.2.1 USE Class Models

Listing C.3: *USE Class Model for Slice 5 to Check Conformance*

```
/*
 Model slice for NJH to
5. Check Conformance

Written by Phillipa Bennett
Date Sept 20, 2016
Version 4
 */

model NJHg_slice_5

/* Abstract CLASSES */

abstract class Data end
abstract class Permission end

/* Extended abstract classes */
abstract class AccessTicket < Permission end

/* Unextended concrete classes */
class DataItem
attributes
    name: String
end

class Query
attributes
operations
    download()
    view()
end

abstract class Status end

/*  Extended concrete classes */

class Date < Data
attributes
    day: Integer
    month: Integer
    year: Integer
operations
    isIdentified(): Boolean
    isNotIdentified(): Boolean
end

class DStr < Data
attributes
    sVal: String
end
```

```

class Project end

class QryData < DataItem end
class RetData < DataItem end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class DownloadDisabled < Status end
class DownloadAllowed < Status end

/* ASSOCIATIONS */
association DataValues between
    DataItem[*]
    Data[1]
end

association EnteredOn between
    DataItem[*] role item
    Date[0..1] role date
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation program to work, enforced as 1in a
        constraint */
    Query[*]
end

association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association VDAllowed between
    Query[*]
    Status[0..1]
end
```

Listing C.4: *USE Class Model for Slice 4 to Execute Query*

```
/*
 Model slice for NJH to
4. execute query

Written by Phillipa Bennett
Date Sept 1, 2016
Version 4
 */

model NJHg_slice_4

/* Abstract CLASSES */
abstract class Category end
abstract class Data end
abstract class DataSource end
abstract class Permission end
abstract class Rule
attributes
operations
    applyRule()
end

/* Extended abstract classes */
abstract class AccessTicket < Permission end
abstract class AccessRule < Rule end

abstract class HIPAACat < Category end
abstract class Consent < Category end

abstract class Type end

/* Unextended concrete classes */
class DataItem
attributes
    name: String
end

class Patient end
class Personnel end
class Query
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end

/*  Extended concrete classes */
class Allow < Consent end
class Disallow < Consent end

class Date < Data
attributes
    day: Integer
    month: Integer
    year: Integer
operations
    isIdentified(): Boolean
    isNotIdentified(): Boolean
```

```
end

/*class DStr < Data
attributes
    sVal: String
end */
class HDate < HIPAACat end

class Project < DataSource end
class ClinicalDB < DataSource end

class Researcher < Personnel end
class Qualifier < Personnel end

class QryData < DataItem end
class RetData < DataItem end

class Individual < Type end
class Group < Type end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class TransformHDate < AccessRule end
class PatientConsent < AccessRule end

/* ASSOCIATIONS */
association ARAppliesTo between
    AccessRule[*] role accessrule
    Type[1..*] role type
end
association ARHides between
    AccessRule[*]
    Category[*]
end

association ARTransforms between
    AccessRule[*] role hAccessRules
    HIPAACat[*]
end

association DataValues between
    DataItem[*]
    Data[1]
end

association DICat between
    DataItem[*]
    HIPAACat[*]
end

association DISource between
    DataSource[0..1]
    DataItem[*]
end

association EnteredOn between
    DataItem[*] role item
    Date[0..1] role date
end

association PatientData between
```

```
    Patient[0..1]
    DataItem[*]
    Consent[0..1]
end

association PermRules between
    Permission[*]
    Rule[1..*]
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectDataCollector between
    Project[*]
    Personnel[0..1] role dc
end

association ProjectMembers between
    Project[*] role proj
    Researcher[*] role members
end

association ProjectPI between
    Project[*] role pi_proj
    Researcher[0..1] role pi
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation program to work, enforced as 1in a
        constraint */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
end

association QryWorksOn between
    Query[*]
    QryData[*]
end

association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association RDType between
    Query[*] role rd_qry
    RetData[*] role rd_data
    Type[0..1]
end
```

Listing C.5: *USE Class Model for Slice 3 to Approve Access Ticket*

```
/*
 Model slice for NJH to
3. approve project access ticket,

Written by Phillipa Bennett
Date August 18, 2016
Version 4
 */

model NJHg_slice_1

/* Abstract CLASSES */
abstract class DataSource end
abstract class DataTransform end
abstract class Permission end
abstract class Rule
attributes
operations
    applyRule()
end
abstract class Purpose end

/* Extended abstract classes */
abstract class AccessTicket < Permission end

class TotallyDeIDed < DataTransform end
class NotTotallyDeIDed < DataTransform end

abstract class Licence < Permission end
abstract class DecisionRule < Rule end

/* Unextended concrete classes */
class Personnel end
class Query
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end


/*  Extended concrete classes */
class Project < DataSource end
class ClinicalDB < DataSource end


class Fishing < Licence end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class CanUseTotallyDeIDed < DecisionRule end
class ClinicalDBNeedsDataCollector < DecisionRule end
class DataAccessAgreementPresent < DecisionRule end
class DataSourcePriorityOK < DecisionRule end
class LicenedTeamAndPI < DecisionRule end
class NoOverlapPITeamDC < DecisionRule end
class NoSupsInPIandDC < DecisionRule end
```

```
class PIDefined < DecisionRule end
class ProjectMembersDefined < DecisionRule end
class QualifierPresent < DecisionRule end
class SomePurposeNotDirectTreatment < DecisionRule end
class SomeQueriesDefined < DecisionRule end
class SomeSourcesDefined < DecisionRule end

class DirectTreatment < Purpose end
class Research < Purpose end

/* These classes are defined using the 'in' keyword in the Alloy model.
   How will we achieve this in OCL? */
class Qualifier < Personnel
attributes
operations
   QualifyResearcher(res: Researcher)
end
class Researcher < Personnel end


/* ASSOCIATIONS */

association ATPriority between
   AccessTicket[*] role ant
   AccessTicket[*] role desc
end

association DataAccessAgreement between
   Project[*] role owner
   Project[*] role user
end

association PermRules between
   Permission[*]
   Rule[1..*]
end

association ProjectAT between
   Project[*]
   AccessTicket[0..1]
end

association ProjectDataCollector between
   Project[*]
   Personnel[0..1] role dc
end

association ProjectDataTransformRequired between
   Project[*]
   DataTransform[0..1]
end

association ProjectMembers between
   Project[*] role proj
   Researcher[*] role members
end

association ProjectPI between
   Project[*] role pi_proj
   Researcher[0..1] role pi
end

```

```
association ProjectPurpose between
    Project[*]
    Purpose[0..1]
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
end

association ResearcherL between
    Researcher[*]
    Licence[0..1]
end

association Supervisors between
    Personnel[*] role supervisor
    Personnel[*] role supervised
end
```

C.2.2 OCLConstraints

Listing C.6: *USE Constraints applicable only to Slices 2, and 3 to Approve Researcher's Licence and Approve Access Ticket respectively - filename reference for listings in Section C.2.5 is slice_23g.cnsts*

```
context Fishing

inv singletonFishing:
    Fishing.allInstances->size()<=1

inv FishingDesicionRules:
    rule->forAll(r | r.oclIsTypeOf(QualifierPresent)=true)

context QualifierPresent

inv QualifierPresentOnlyForFishing:
    permission->forAll(p | p.oclIsTypeOf(Fishing)=true)

context DecisionRule

inv singletonEachDecisionRule:
    DecisionRule.allInstances.select(
        oclIsTypeOf(CanUseTotallyDeIDed)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(DataSourcePriorityOK)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(LicenedTeamAndPI)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(NoOverlapPITeamDC)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(NoSupsInPIandDC)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(PIDefined)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(ProjectMembersDefined)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(QualifierPresent)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(SomePurposeNotDirectTreatment)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(SomeQueriesDefined)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(SomeSourcesDefined)=true)->size<=1
and
    DecisionRule.allInstances.select(
        oclIsTypeOf(DataAccessAgreementPresent)=true)->size<=1
```

Listing C.7: *USE Constraints applicable only to Slices 2, 3, and 4 to Approve Researcher's Licence, Approve Access Ticket, and Execute Query respectively - filename reference for listings in Section C.2.5 is slice_234g.cnsts*

```

/* This was weakened in the CD for slice 5and 4,
    so we add it as a constraint here */
context Permission
inv invEachPermHasAtLeastOneRule:
    rule->size()>=1
```

Listing C.8: *USE Constraints applicable only to Slices 3 and 4 to Approve Access Ticket and Execute Query respectively - filename reference for listings in Section C.2.5 is slice_34g.cnsts*

```
context AccessTicket

inv singletonEachAT:
    AccessTicket.allInstances.select(
        oclIsTypeOf(Identified)=true)->size()<=1
    and
    AccessTicket.allInstances.select(
        oclIsTypeOf(DeIDed)=true)->size()<=1

context ClinicalDB
inv singletonClinicalDB:
    ClinicalDB.allInstances.select(oclIsTypeOf(ClinicalDB)=true)->size()<=1

context Project
inv invProjectNeedsDataCollectorForClinicalDB:
    dataSource->select(oclIsTypeOf(ClinicalDB)=true)->size()=1 implies
        dc->size()=1
/* this not really required because executing the query should check it */
inv invProjectSources2:
    dataSource->select(oclIsTypeOf(Project)=true)->forAll(
        p | p.oclAsType(Project).accessTicket->size()=1 )

context DataSource
inv invProjectSources1: /* easier to write this in the contex of DataSource */
    project.closure(project)->excludes(self)
```

Listing C.9: *USE Constraints applicable only to Slices 3, 4 and 5 to Approve Researcher's Licence, Approve Access Ticket, and Execute Query respectively - filename reference for listings in Section C.2.5 is slice_345g.cnsts*

```
context Query

inv invEachQueryAssociatedWithOnlyOneProject:
    project->size()=1
```

Listing C.10: *USE Constraints applicable only to Slices 4 and 5 to Execute Query and Check Conformance respectively - filename reference for listings in Section C.2.5 is slice_45g.cnsts*

```
context Date
inv attValues1:
    day >= 0and day <= 31
    and
    month >= 0and month <=12
    and
    year >= 1900

inv attValues2:
    day>29 implies
        Sequence{1,3..12}->includes(month)

inv attValues3:
    (month=2 and day=29) implies
        year.mod(4)=0

inv attValues4:
    (month=2 and day=29 and year.mod(100)=0) implies
        year.mod(400)=0

context Type
inv singletonEachType:
    Type.allInstances.select(
        oclIsTypeOf(Group)=true)->size<=1
    and
    Type.allInstances.select(
        oclIsTypeOf(Individual)=true)->size<=1

context RetData
inv retDataInOneQuery:
    qry->size()<=1 /* should be =1? */

inv retDataType:
    type->size()=1

context Query
inv invRDType:
    rData->forAll(
        (qData->size()=1 implies
            type->select(oclIsTypeOf(Individual)=true)->size=1)
        and
        (qData->size()>1 implies
            type->select(oclIsTypeOf(Group)=true)->size=1)
        )


inv invQryReturns1:
    qryData->includesAll(qData)

inv invQryReturns2:
    qData->size()>0 implies project.accessTicket->size()=1
```

Listing C.11: *USE Constraints applicable only to Slice 5 to Check Conformance, filename reference for listings in Section C.2.5 is slice_5g_1.cnsts*

```
context Status
inv singletonEachStatus:
    Status.allInstances.select(
        oclIsTypeOf(DownloadDisabled)=true)->size<=1
    and
    Status.allInstances.select(
        oclIsTypeOf(DownloadAllowed)=true)->size<=1
```

Listing C.12: *USE Constraints applicable only to Slice 5 to Check Conformance, filename reference for listings in Section C.2.5 is slice_5g_2.cnsts*

```
context Query
inv invVDAllowed:
    let
    cond1: Boolean =
        rData->size()>0,
    cond2: Boolean =
        status->size()=1
    in
    cond1 implies cond2
    and
    cond2 implies cond1

inv invDownloadAllowedDeIDed:
    let
    cond1: Boolean =
        project.accessTicket->select(oclIsTypeOf(DeIDed)=true)->size()=1,
    cond2: Boolean = rData.data->select(
            oclIsTypeOf(Date)=true)->forAll(d |d.oclAsType(Date).day=0),
    cond3: Boolean =
        status.oclIsTypeOf(DownloadAllowed)=true
    in
    cond1 implies (
        (cond2 implies cond3)
        and
        (cond3 implies cond2))

inv invDownloadDisabledDeIDed:
    let
    cond1: Boolean =
        project.accessTicket->select(oclIsTypeOf(DeIDed)=true)->size()=1,
    cond2: Boolean =
        rData.data->select(
            oclIsTypeOf(Date)=true)->exists(d |d.oclAsType(Date).day<>0),
    cond3: Boolean =
        status.oclIsTypeOf(DownloadDisabled)=true
    in
    cond1 implies (
        (cond2 implies cond3)
        and
        (cond3 implies cond2))
```

Listing C.13: *USE Constraints applicable only to Slice 4 to Execute Query - filename reference for listings in Section C.2.5 is slice_4g.cnsts*

```
context AccessRule
```

```
inv invARHides:
    category->excludesAll(hIPAACat) and
        hIPAACat->excludesAll(category)

inv singletonEachAccessRule:
    AccessRule.allInstances.select(
        oclIsTypeOf(TransformHDate)=true)->size<=1
    and
    AccessRule.allInstances.select(
        oclIsTypeOf(PatientConsent)=true)->size<=1

context Category
inv singletonEachCategory:
    Category.allInstances.select(
        oclIsTypeOf(HDate)=true)->size<=1
    and
    Category.allInstances.select(
        oclIsTypeOf(Allow)=true)->size<=1
    and
    Category.allInstances.select(
        oclIsTypeOf(Disallow)=true)->size<=1

context DataItem
inv invDISourceAndEnteredOn:
    dataSource.oclIsTypeOf(ClinicalDB)=true implies
        (date->size()=1 and
            date.day >=1 and date.month >=1 )

/* Correctly categoriises ClinicalCB dates as HDate
 this relaxed for non-conformance */
/*inv invPatientDataAndDICat:
    (data.oclIsTypeOf(Date)=true and
        self.oclIsTypeOf(RetData)=false)
    implies
        hIPAACat->select(oclIsTypeOf(HDate)=true)->size()=1 */

inv invEnteredOn:
    patient->size()>0 implies (date->size()=1 and date.day>=1)

context DataSource
inv invDISource1:
    if oclIsTypeOf(Project)=true then
        self.dataItem->forAll(oclIsTypeOf(RetData)=true )
    else
        self.dataItem->forAll(oclIsTypeOf(RetData)=false)
    endif

context Query
inv invDISource2:
    rData.qData->forAll(qd | qd.dataSource.oclIsTypeOf(ClinicalDB)=true)
```

Listing C.14: *OCL Constraints applicable only to Slice 3 to Approve Access Ticket - filename reference for listings in Section C.2.5 is slice_3g.cnsts*

```
/*
Constraints for approve project licence

Written by Phillipa Bennett
Date August 18, 2016
```

```
Version 4
 */

context AccessTicket

inv invATPriority: /* no cycles */
    desc.closure(desc)->excludes(self) or
        ant.closure(ant)->excludes(self)

inv QualifierPresentNotAnATDecisionRule:
    rule.select(
        oclIsTypeOf(QualifierPresent)=true)->size()=0

context DataTransform

inv singletonEachDT:
    DataTransform.allInstances.select(
        oclIsTypeOf(TotallyDeIDed)=true)->size()<=1
    and
    DataTransform.allInstances.select(
        oclIsTypeOf(NotTotallyDeIDed)=true)->size()<=1

context Purpose

inv singletonEachPurpose:
    Purpose.allInstances.select(
        oclIsTypeOf(DirectTreatment)=true)->size()<=1
    and
    Purpose.allInstances.select(
        oclIsTypeOf(Research)=true)->size()<=1

context Project
inv invDataAccessAggreement1: /* no cycles */
    owner->closure(owner)->excludes(self) or
        user->closure(user)->excludes(self)

/* inv invDataAccessAggreement2: - see invDataAccessAgreementPresent below */

context Personnel
inv invSupervisors: /* no cycles */
    supervised->closure(supervised)->excludes(self) or
        supervisor->closure(supervisor)->excludes(self)
```

C.2.3 ASSL Procedures

Listing C.15: *ASSL Procedures for Slice 4 to Execute Query*

```
/* ********** ********** ********** ********** ********** ********** **********
   PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
 procedure add_4g_singleton_objects(
   max: Integer )
var
   /* misc */
   n: Integer;

begin
   /* a. create singleton objects */
   Create(ClinicalDB);
   Create(HDate);
   Create(Allow);
   Create(Disallow);
   Create(Group);
   Create(Individual);

   /* Personnel, choose da, pi, and team pool */
   //n := Any([Sequence{1..max}]);
   CreateN(Personnel, [2]); /* if n>2 generation fails */

   //n := Any([Sequence{1..max}]);
   CreateN(Qualifier, [2]); /* if n>2 generation fails */

   n := Any([Sequence{3..max}]);
   CreateN(Researcher, [n]);
end;

/* ********** ********** ********** ********** ********** ********** **********
 PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
procedure configure_AT_AccessRules()
var
   /* Permissions */
   iat: Identified,
   dat: DeIDed,
   g: Group,
   i: Individual,

   /* Hippa categories */
   hipaad: HDate,

   /* abstract DecisionRule object */
   ar: AccessRule;

begin
   dat := Any([DeIDed.allInstances->asSequence()]);
   iat := Any([Identified.allInstances->asSequence()]);
   g := Any([Group.allInstances->asSequence()]);
   i := Any([Individual.allInstances->asSequence()]);
   hipaad := Any([HDate.allInstances->asSequence()]);

   /* Access ticket AccessRules,
    Create PermRules and ARAppliesTo associations */
   ar := Create(TransformHDate);
```

```
    Insert(PermRules, [dat], [ar]);
    /* Insert(ARAppliesTo, [ar], [g]); */
    Insert(ARAppliesTo, [ar], [i]);
    Insert(ARTransforms, [ar], [hipaad]);

    ar := Create(PatientConsent);
    Insert(PermRules, [dat], [ar]);
    Insert(PermRules, [iat], [ar]);
    Insert(ARAppliesTo, [ar], [g]);
    Insert(ARAppliesTo, [ar], [i]);
end;

/* ********** ********** ********** ********** ********** ********** **********
 PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
procedure generate_patient_data(
    max: Integer,
    maxMonth: Integer,
    currentYear: Integer)
var
    di: Sequence(DataItem),
    di_5: Sequence(DataItem),
    pdi: Sequence(DataItem),

    da: Sequence(Date),
    nda: Sequence(Date),

    patients: Sequence(Patient),
    cnsts: Sequence(Consent),

    cDB: ClinicalDB,
    date: Date,
    cnst: Consent,
    hipaad: HDate,

    allow: Boolean,
    first: Boolean,
    maxYears: Integer,
    m: Integer,
    nbr: Integer,
    n: Integer;

begin
    allow := [false];
    nbr := [1];
    maxYears :=[95];

    /* Categories */
    cnsts := [Consent.allInstances->asSequence()];
    hipaad := Any([HDate.allInstances->asSequence()]);

    /* DataSources */
    cDB := Any([ClinicalDB.allInstances->asSequence()]);

    /* Patients */
    n := Any([Sequence{1..max}]);
    patients := CreateN(Patient, [n]);

    /* DataItems */
    di_5 := [DataItem.allInstances()->asSequence()];
    di := CreateN(DataItem, [nbr*n]); /* nbr DataItem for each patient */
    di := [DataItem.allInstances()->asSequence()]; /* includes dataitems created in slice 5*/
```

```

/* Date Data */
da := CreateN(Date, [di->size()]);
for d: Date in [da] begin
    // day
    [d].day := Any([Sequence{1..31}]);

    // month
    if [d.day>28] then begin
        m:= Any([Sequence{1, 3..12}]);
    end
    else begin
        m:= Any([Sequence{1..12}]); //leave out month=2 & day=29 for now
    end;
    [d].month := [m];

    // year
    if [d.month>maxMonth] then begin
        m:= Any([Sequence{currentYear-maxYears..currentYear-1}]);
    end
    else begin
        m:= Any([Sequence{currentYear-maxYears..currentYear}]);
    end;
    [d].year := [m];
end;
da := [Date.allInstances()->
    select(d | d.day<>0)->asSequence()]; /* includes identified dataitems created in slice
        5*/

/* Association Links */
Try(DataValues, [di], [da]);

/* PatientData */
for p: Patient in [patients] begin
    first := [false];
    n := Any([Sequence{1..nbr}]);
    pdi := Sub([di->select(patient->size()=0)->asSequence()], [n]);
    for d: DataItem in [di_5] begin
        /*if [first=false] then begin
            [d].name := Any([Sequence{'Age'}]);
            first := [true];
        end; */
        /* ensure at lease one DataItem has Allow in PatientData */
        if [allow=false] then begin
            cnst := Any([cnsts->select(oclIsTypeOf(Allow)=true)]);
            allow := [true];
        end;
        /* else begin
            cnst := Any([cnsts]);
        end; */
        Insert(PatientData, [p], [d], [cnst]);
    end;
end;

/* Delete DataItems not assigned to patient */
for d: DataItem in [di->select(patient->size()=0)->asSequence()] begin
    Delete([d]);
end;

/* do we need to update di? */
di := [DataItem.allInstances->asSequence()];

```

```
    /* DISource for data linked to a patient */
    for d: DataItem in [di] begin
        date := Any([da]);
        Insert(EnteredOn, [d], [date]);
        Insert(DISource, [cDB], [d]);
    end;

    /* Delete Date not assigned to DataItem in DataValues or EnteredOn */
    for d: Date in [
        da->select(dataItem->size()=0)->asSequence()] begin
        if [d.item->size()=0] then begin Delete([d]); end;
    end;
    /* do we need to update da? */
    da := [Date.allInstances->asSequence()];

    /* Set HDate for Dates */
    for d: DataItem in [di] begin
        if [d.data.oclIsTypeOf(Date)=true] then begin
            //date := [d.data.oclAsType(Date)];
            if [d.name='Age'] then begin
                Insert(DICat, [d], [hipaad]);
            end;
        end;
    end;
end;

/* ********** ********** ********** ********** ********** ********** **********
 PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
procedure setup_project(
    proj: Project,
    qry: Query,
    at: AccessTicket,
    pdss: Sequence(Project),
    max: Integer)

var
    /* for objects already created */
    cDB: ClinicalDB,

    projs: Sequence(Project),

    pers: Sequence(Personnel),
    res: Sequence(Researcher),

    /* for setting up links */
    rs: Sequence(Researcher),
    pss: Sequence(Project),
    da: Personnel,
    pi: Researcher,
    team: Sequence(Researcher),

    /* misc */
    m: Integer,
    n: Integer;

begin
    cDB := Any([ClinicalDB.allInstances->asSequence()]);

    /* Personnel, Researchers */
    pers := [Personnel.allInstances->asSequence()];
    res := [Researcher.allInstances->asSequence()];
```

```

    /* choose da */
    da := Any([pers]);

    /* set pi and update team */
    if [da.oclIsTypeOf(Researcher)] then begin
       pi := Any([res->excluding(da.oclAsType(Researcher))]);
       team := [res->excluding(da.oclAsType(Researcher))->excluding(pi)];
    end
    else begin
       pi := Any([res]);
       team := [res->excluding(pi)];
    end;

    /* Projects, put proj in projs */
    projs := [Project.allInstances->excluding(proj)->asSequence()];

    /* Generate applicable association links */

    /* SomeSourcesDefined, Clinical DB ProjectSource for proj */
    Insert(ProjectSources, [proj], [cDB]);

     /* Since pi, team and da do not overlap, NoOverlapPITeamDC=true
       Insert datacollector is applicable */
    if [proj.dataSource->select(oclIsTypeOf(ClinicalDB)=true)->size=1]
    then begin
       Insert(ProjectDataCollector, [proj], [da]);
    end;

    /* Add other ProjectSources */
    for p:Project in [pdss] begin
       if [p.accessTicket->size()>0] then begin
          Insert(ProjectSources, [proj], [p]);
       end;
    end;

    /*  Insert Project PI */
    Insert(ProjectPI, [proj], [pi]);

    /*  Insert Project Members */
    m := [team->size()];
    n := Any([Sequence{1..m}]);
    rs := Sub([team], [n]);
    for r: Researcher in [rs] begin
       Insert(ProjectMembers, [proj], [r]);
    end;

    /* Insert Link between proj and qry in ProjectQueries */
    Insert(ProjectQueries, [proj], [qry]);

    /* Insert link between proj and at */
    Insert(ProjectAT, [proj], [at]);
end;

/* ********** ********** ********** ********** ********** ********** **********
 PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
procedure add_query_works_on(
    proj: Project,
    qry: Query,
    res: Researcher,
    at: AccessTicket,
```

```
    max: Integer)
var
    qd: Sequence(QryData),
    qd2: Sequence(QryData),
    di: Sequence(DataItem),

    p: Patient,
    c: Consent,
    d: DataItem,

    n: Integer;

begin

    /* check prerequisites */
    if [proj.query->includes(qry) and
            proj.pi->union(proj.members)->includes(res) and
                proj.accessTicket->size() = 1] then begin

        /* Apply Patient Consent AccessRule */
        if [at.rule->select(oclIsTypeOf(PatientConsent)=true)->size()=1]
        then begin
            di := [Allow.allInstances.dataItem->asSequence()];
        end
        else begin
            di := [DataItem.allInstances->asSequence()];
        end;

        /* add qd as a subset of di and set up related associations*/
        n := Any([Sequence{1..di->size()}]);
        qd2 := [QryData.allInstances()->asSequence()]; /* set before qd */
        qd := CreateN(QryData, [n]);
        n := [1];

        for q: QryData in [qd] begin
            d := [di->at(n)];
            [q].name := [d.name];

            Insert(DataValues, [q], [d.data]);
        end;

        for q: QryData in [qd->union(qd2)] begin
            /*for h: HIPAACat in [d.hIPAACat->asSequence()] begin
                if [q.name='Age'] then begin Insert(DICat, [q], [h]); end;
            end;

            Insert(DISource, [d.dataSource], [q]);

            p:= Any([d.patient->asSequence]);
            c:= Any([d.consent->asSequence]);
            Insert(PatientData, [p], [q], [c]);

            Insert(EnteredOn, [q], [d.date]); */
            Insert(QryWorksOn, [qry], [q]);

            n := [n + 1];
        end; /* end for qryData, qd and qd2 */

    end; /* end prerequisites */
    /* else do nothing */
end;
```

```

/* ********** ********** ********** ********** ********** ********** **********
 PROCEDURE
 * ********** ********** ********** ********** ********** ********** **********/
procedure add_query_returns(
    qry: Query,
    at: AccessTicket
    )
var
    qd: Sequence(QryData),
    rd: Sequence(RetData),
    rd2: Sequence(RetData),
    di: Sequence(DataItem),

    ind: Individual,
    grp: Group,
    da: Data,
    p: Patient,
    c: Consent,
    d: DataItem,

    n: Integer;

begin
    qd := [qry.qryData->asSequence()];
    ind := Any([Individual.allInstances->asSequence()]);
    grp := Any([Group.allInstances->asSequence()]);

    /* add RetData based on access ticket */
    n := Any([Sequence{1..qd->size()}]);
    rd2 := [RetData.allInstances()->asSequence()]; /* set before rd */
    rd := CreateN(RetData, [n]);
    n:= [1];
    for r: RetData in [rd2] begin
        d := [qd->at(n)];
        [r].name := [d.name];
        Insert(QryReturns, [qry], [r], [d.oclAsType(QryData)]);

        /* Apply TransformHDate AccessRule */
        if [at.rule->select(oclIsTypeOf(TransformHDate)=true)->size()=1 and
            at.rule->select(
                oclIsTypeOf(TransformHDate)=true).oclAsType(AccessRule).type->
                    select(oclIsTypeOf(Individual)=true)->size()=1 and
            at.rule->select(oclIsTypeOf(TransformHDate)=true).oclAsType(
                AccessRule).hIPAACat.dataItem->includes(d)
            ]
        then begin
            da := Create(Date);
            [da.oclAsType(Date)].day := [0];
            [da.oclAsType(Date)].month := [0];
            [da.oclAsType(Date)].year := [d.data.oclAsType(Date).year];
            Insert(DataValues, [r], [da]);
        end
        else begin
            Insert(DataValues, [r], [d.data]);
        end; /* end Apply TransformHDate AccessRule */

        /* setup RDType */
        if [r.qData->size()=1] then begin
            Insert(RDType, [qry], [r], [ind]);
        end
        else begin
```

```
            Insert(RDType, [qry], [r], [grp]);
        end;
    end; /* end for each RetData */
end;

procedure complete_query_returns(
    qry: Query,
    at: AccessTicket
    )
var

    rd: Sequence(RetData),
    ind: Individual,
grp: Group;

begin
    ind := Any([Individual.allInstances->asSequence()]);
    grp := Any([Group.allInstances->asSequence()]);

    rd := [RetData.allInstances()->asSequence()];

    for r: RetData in [rd] begin
        /* setup RDType */
        if [r.qData->size()=1] then begin
            Insert(RDType, [qry], [r], [ind]);
        end
        else begin
            Insert(RDType, [qry], [r], [grp]);
        end;
    end; /* end for each RetData */
end;
```

Listing C.16: *ASSL Procedures for Slice 3 to Approve Access ticket*

```
procedure generate_objects(
    max: Integer )
var
    /* misc */
    n: Integer;

begin
    /* a. create singleton objects */
    Create(DirectTreatment);
    Create(Research);
    Create(TotallyDeIDed);
    Create(NotTotallyDeIDed);

    /* Personnel, choose da, pi, and team pool */
    n := Any([Sequence{1..max}]);
    CreateN(Personnel, [2]); /* if n>2 generation fails */

    n := Any([Sequence{1..max}]);
    CreateN(Qualifier, [1]); /* if n>2 generation fails */

    //n := Any([Sequence{2..max}]);
    //CreateN(Researcher, [n]);
end;

procedure configure_PermRules_and_ATPriority()
var
    /* Permissions */
    fl: Fishing,
    iat: Identified,
    dat: DeIDed,

    /* abstract DecisionRule object */
    dr: DecisionRule;

begin
    fl := Any([Fishing.allInstances->asSequence()]);
    dat := Any([DeIDed.allInstances->asSequence()]);
    iat := Any([Identified.allInstances->asSequence()]);

    /* ATPriority */
    Insert(ATPriority, [iat], [dat]);

    /* Access ticket DecisionRules and Create PermRules Associations */
    dr := Create(CanUseTotallyDeIDed);
    Insert(PermRules, [dat], [dr]);

    dr := Create(ClinicalDBNeedsDataCollector);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(DataAccessAgreementPresent);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(DataSourcePriorityOK);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(LicenedTeamAndPI);
```

```
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(NoOverlapPITeamDC);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    /*dr := Create(NoSupsInPIandDC);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]); */

    dr := Create(PIDefined);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(ProjectMembersDefined);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(SomePurposeNotDirectTreatment);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(SomeQueriesDefined);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(SomeSourcesDefined);
    Insert(PermRules, [dat], [dr]);
    Insert(PermRules, [iat], [dr]);

    dr := Create(QualifierPresent);
    Insert(PermRules, [fl], [dr]);
end;


procedure generate_approved_project(
    proj: Project,
    at: AccessTicket,
    max: Integer )
var
    /* for objects already created */
    td: TotallyDeIDed,
    ntd: NotTotallyDeIDed,
    research: Research,
    fl: Fishing,
    cDB: ClinicalDB,

    projs: Sequence(Project),
    ps: Sequence(Project),

    pers: Sequence(Personnel),
    res: Sequence(Researcher),

    //qrys: Sequence(Query),

    /* for setting up links */
    rs: Sequence(Researcher),
    qs: Sequence(Query),
    pss: Sequence(Project),
    da: Personnel,
    pi: Researcher,
```

```
    //team: Sequence(Researcher),

    /* misc */
    m: Integer,
    n: Integer;

begin
    td := Any([TotallyDeIDed.allInstances->asSequence()]);
    ntd := Any([NotTotallyDeIDed.allInstances->asSequence()]);

    research := Any([Research.allInstances->asSequence()]);
    fl := Any([Fishing.allInstances->asSequence()]);
    cDB := Any([ClinicalDB.allInstances->asSequence()]);

    /* Personnel, Researchers */
    pers := [Personnel.allInstances->asSequence()];
    res := [Researcher.allInstances->asSequence()];

    da := [proj.dc];
    pi := [proj.pi];
    //team := [proj.members->asSequence()];

    /* Projects, put proj in projs */
    projs := [Project.allInstances->excluding(proj)->asSequence()];
    ps := [Sequence{proj}];

    /* Queries */
    n := Any([Sequence{1..max}]);
    //qrys := CreateN(Query, [n]);

    /* Generate association links to fulfil each rule */

    /* 1. CanUseTotallyDeIdentified */
    if [at.oclIsTypeOf(DeIDed)=true]
    then begin
        Insert(ProjectDataTransformRequired, [proj], [td]);
    end
    else begin
        Insert(ProjectDataTransformRequired, [proj], [ntd]);
    end;

    /* 13. SomeSourcesDefined, Clinical DB ProjectSource for proj
     - from slice 4*/

    /* 2. 6. ClinicalDBNeedsDataCollector,
     Since pi, team and da do not overlap, NoOverlapPITeamDC=true
     get from slice 4*/


    /* 3. 4. DataAccessAgreementPresent, DataDourcePriorityOK */
    m := [projs->size()];
    if [m>0] then begin
        n := Any([Sequence{1..m}]);
        pss := Sub([projs], [n]);
        for p:Project in [pss] begin
            if [p.accessTicket->size()>0] then begin
                if [p.accessTicket=at or at.ant->includes(p.accessTicket)]
                then begin
                    Insert(ProjectSources, [proj], [p]);
                    Insert(DataAccessAgreement, [p], [proj]);
                end;
            end;
```

```
        end;
    end;

    /* 4. See 3above */

    /* 5. See after 8. and 9. below (as 5depends on 8& 9)*/

    /* 6. See 2. above */

    /* 7. NoSupsInPIandDC */
    /*Try(Supervisors, [pers], [pers]); */

    /* 8. PIDefined - get from slice 4*/

    /* 9. ProjectMembersDefined and LicencedTeamAndPI - from slice 4*/

    /* 5. LicencedTeamAndPI */
    if [at.rule.select(oclIsTypeOf(LicenedTeamAndPI)=true)->size()=1]
    then begin
        rs := [proj.members->including(proj.pi)->asSequence()];
        for r: Researcher in [rs] begin
            if [r.licence->size()=0] then begin
                Insert(ResearcherL, [r], [fl]);
            end;
        end;
    end;

    /* 10. QualifierPresent does not apply to access tickets */

    /* 11. SomePurposeNotDirectTreatment */
    if [at.rule.select(
        oclIsTypeOf(SomePurposeNotDirectTreatment)=true)->size()=1]
    then begin
            Insert(ProjectPurpose, [proj], [research]);
    end;

    /* 12. /* Some Queries Defined from slice 4*/

    /* 13. SomeSourcesDefined. inserted before 2(as 2depends on it) */

    /* Finally insert link between proj and at - from slice 4*/

end;
```

C.2.4 SOIL Commands

Listing C.17: *SOIL Commands used to re-create objects from slice 5 needed in other slices - filename reference for listings in Section C.2.5 is slice_5_overlap\overlapping_objects_1.soil*

```
!create iat: Identified
!new DeIDed('DeIDed_0')
```

Listing C.18: *SOIL Commands used to re-create objects from slice 5 needed in other slices - filename reference for listings in Section C.2.5 is slice_5_overlap\overlapping_objects_2.soil*

```
-- Script generated by USE 4.2.0
!new QryData('DataItem_4')
!new QryData('DataItem_5')

!DataItem_4.name := 'Age'
!DataItem_5.name := 'Other'

!new Date('Date_1')
!Date_1.day := 9
!Date_1.month := 8
!Date_1.year := 1931

!insert (DataItem_5,Date_1) into DataValues
!insert (DataItem_4,Date_1) into DataValues
```

Listing C.19: *SOIL Commands used to re-create objects from slice 5 needed in other slices - filename reference for listings in Section C.2.5 is slice_5_overlap\overlapping_objects_3.soil*

```
!new Project('Project_1')
!new Query('Query_0')
```

Listing C.20: *SOIL Commands used to re-create objects from slice 5 needed in other slices - filename reference for listings in Section C.2.5 is slice_5_overlap\overlapping_objects_4.soil*

```
-- Script generated by USE 4.2.0
!new Project('Project_1')
!new Query('Query_0')
!insert (Project_1,DeIDed_0) into ProjectAT
!insert (Project_1,Query_0) into ProjectQueries
```

Listing C.21: *SOIL Commands used to re-create objects from slice 5 needed in other slices - filename reference for listings in Section C.2.5 is slice_5_overlap\overlapping_objects_5.soil*

```
-- Script generated by USE 4.2.0
!new RetData('DataItem_0')
!new RetData('DataItem_1')
!new RetData('DataItem_2')
```

```
!new RetData('DataItem_3')

!new Date('Date_0')
!Date_0.day := 0
!Date_0.month := 0
!Date_0.year := 1931

!DataItem_0.name := 'Age'
!insert (Query_0,DataItem_0,DataItem_4) into QryReturns
!insert (DataItem_0,Date_0) into DataValues

!DataItem_3.name := 'Other'
!insert (Query_0,DataItem_3,DataItem_5) into QryReturns
!insert (DataItem_3,Date_1) into DataValues

!DataItem_2.name := 'Age'
!insert (Query_0,DataItem_2,DataItem_4) into QryReturns
!insert (DataItem_2,Date_0) into DataValues

!DataItem_1.name := 'Age'
!insert (Query_0,DataItem_1,DataItem_4) into QryReturns
!insert (DataItem_1,Date_0) into DataValues

!insert (Query_0,DownloadDisabled_0) into VDAllowed
```

Listing C.22: *SOIL Commands used to re-create objects from slice 4 needed in slice 3 - filename reference for listings in Section C.2.5 is slice_4_overlap\overlapping_objects_1.soil*

```
-- Script generated by USE 4.2.0

!new DeIDed('DeIDed_0')
!new ClinicalDB('ClinicalDB1')
```

Listing C.23: *SOIL Commands used to re-create objects from slice 4 needed in slice 3 - filename reference for listings in Section C.2.5 is slice_4_overlap\overlapping_objects_2.soil*

```
-- Script generated by USE 4.2.0

!new Project('Project_0')
```

Listing C.24: *SOIL Commands used to re-create objects from slice 4 needed in slice 3 - filename reference for listings in Section C.2.5 is slice_4_overlap\overlapping_objects_3.soil*

```
-- Script generated by USE 4.2.0

!new Researcher('Researcher1')
!new Researcher('Researcher2')
!new Researcher('Researcher3')
!new Researcher('res1')
!new Project('Project_1')
!new Query('Query_0')
!insert (Project_1,ClinicalDB1) into ProjectSources
!insert (Project_1,Researcher1) into ProjectDataCollector
```

```
!insert (Project_1,Researcher2) into ProjectMembers
!insert (Project_1,res1) into ProjectMembers
!insert (Project_1,Researcher3) into ProjectPI
!insert (Project_1,Query_0) into ProjectQueries
!insert (Project_1,DeIDed_0) into ProjectAT
```

C.2.5 USE Commands to Generate On-Demand Feedback

Listing C.25: *USE Commands to Generate Object Model for Slice 4 to Execute Query*

```
/* 1. Initialisation - remove all the elements in the object diagram */
reset

/* 2. unload constraints */
constraints -unload

/* 3. Load the class diagram specification */
open /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.use

/* 4. Load some of the invariants */
constraints -load /Users/Philly/Desktop/overlap/slice_345g.cnsts

/* 5. load flags, -d enables invariants, -n does not negate the invariants */
constraints -flags -d -n

/* 6. Generate an object model that satisfies invariants in the class diagram
/* a. generate singleton objects */
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl
    add_4g_singleton_objects(3)
gen result accept

/* b. Also, since I want to pass in an access ticket explicitly,
 I create them here */
open /Users/Philly/Desktop/slice_seq_nc/slice_5_overlap/overlapping_objects_1.soil

/* c. generate PermRules and ATPriority links, load appropriate constraints
 here as well */
constraints -load /Users/Philly/Desktop/overlap/slice_234g.cnsts
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl
    configure_AT_AccessRules()
gen result accept

/* d. generate Data for project sources */
open /Users/Philly/Desktop/slice_seq_nc/slice_5_overlap/overlapping_objects_2.soil
gen start /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl generate_patient_data(1, 8
    , 2016)
gen result accept

/* e. since I want to pass in the project and query explicitly,
 I create them here */
open /Users/Philly/Desktop/slice_seq_nc/slice_5_overlap/overlapping_objects_3.soil

/* f. Load the rest of the invariants */
constraints -load /Users/Philly/Desktop/overlap/slice_45g.cnsts
constraints -load /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.cnsts

/* g. setup project links */
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl
    setup_project(Project_1, Query_0, DeIDed_0, Sequence{}, 3)
gen result accept

/* h. since I want to pass in the researcher who is running the query,
 I create it here, I also explicity ass the researcher as a ProjectMember for
 the project that the query belongs to, to ensure successful query execution */
!create res1: Researcher
!insert (Project_1, res1) into ProjectMembers
```

```
/* i. generate query works on data */
open /Users/Philly/Desktop/slice_seq_nc/slice_5_overlap/overlapping_objects_4.soil
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl
    add_query_works_on(Project_1, Query_0, res1, DeIDed_0, 3)
gen result accept

/* j. generate query returns data */
open /Users/Philly/Desktop/slice_seq_nc/slice_5_overlap/overlapping_objects_5.soil
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_4/slice_4g.assl
    complete_query_returns(Query_0, DeIDed_0)
gen result accept

/* 7. Check */
check
```

Listing C.26: *USE Commands to Generate Object Model for Slice 3 to Approve Access Ticket*

```
/* 1. remove all the elements in the object diagram */
reset

/* 2. unload constraints */
constraints -unload

/* 3. Load the class diagram specification */
open /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g.use

/* 4a. Load some of the invariants and flags */
constraints -load /Users/Philly/Desktop/overlap/slice_23g.cnsts
constraints -load /Users/Philly/Desktop/overlap/slice_34g.cnsts

/* load flags, -d enables invariants, -n does not negate the invariants */
constraints -flags -d -n

/* 5. generate an object diagram that satisfys the class diagram

/* a. generate objects */
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g.assl generate_objects(3)
gen result accept

!create fl: Fishing
!create iat: Identified
open /Users/Philly/Desktop/slice_seq_nc/slice_4_overlap/overlapping_objects_1.soil

/* b. generate PermRules and ATPriority links */
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g.assl
    configure_PermRules_and_ATPriority()
gen result accept

/* c. Load some more of the invariants */
constraints -load /Users/Philly/Desktop/overlap/slice_345g.cnsts

/* d. generate projects that are approved */
open /Users/Philly/Desktop/slice_seq_nc/slice_4_overlap/overlapping_objects_2.soil
open /Users/Philly/Desktop/slice_seq_nc/slice_4_overlap/overlapping_objects_3.soil
gen start -b -d /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g.assl
    generate_approved_project(Project_1, DeIDed_0, 2)
gen result accept

/* f. Load the rest of the invariants */
constraints -load /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g.cnsts
constraints -load /Users/Philly/Desktop/slice_seq_nc/slice_3/slice_3g_at.cnsts

/* 6. check that none of the invariants have been violated */
check
```

D.1 Updated Alloy Specifications

D.1.1 Alloy Specifications for Slice 3 to Approve Access Ticket

Listing D.1: *Updated Alloy Specifications for Slice 3 to Approve Access Ticket*

```
/********** ********** ********** ********** ********** ********** **********
   Begin Structural Model, NJH, slice 3

   Written By: Phillipa Bennett
   Version 5
   Date: Version 5completed Nov 28, 2016

   Notes:
   Predicates and Assertions are executed with
      exactly 11Rule
   when the NSIPIDC Rule is excluded from the model.

   Places in the specification that are impacted by excluding of the
   NSIPIDC Rule are labeled with
      *** DA_COI ***
   either just before or at the end of the line.

   Also other notes throughout the specification.

/********** ********** ********** ********** ********** ********** **********/
module slice_3_g_inst

/********** ********** ********** ********** **********
   imports
/********** ********** ********** ********** **********/
open util/relation
open util/ternary
open util/ordering[NJH] as ord

/********** ********** ********** ********** **********
   base abstract signatures
********** ********** ********** ********** **********/
abstract sig
   DataSource,
   DataTransform,
   Permission,
   Purpose,
   Rule{}

/********** ********** ********** ********** **********
   extended abstract signatures
********** ********** ********** ********** **********/
abstract sig
   AccessTicket,
   Licence
extends Permission{}

abstract sig
   DecisionRule
```

```
extends Rule {}

/********** ********** ********** ********** **********
   unextended concrete signatures
********** ********** ********** ********** **********/
sig
   /* Personnel cannot be abstract,
      because of supervisors and data collectors */
   Personnel,
   Query {}

/********** ********** ********** ********** **********
   extended concrete signatures
********** ********** ********** ********** **********/
one sig
   CUTD,      /* CanUseTotallyDeIDed */
   DAAP,      /* DataAccessAgreementPresent */
   DSPOK,        /* DataSourcePriorityOK */
   LTAPI,        /* LicenedTeamAndPI */
   NOPITDC,   /* NoOverlapPITeamDC */
   NSIPIDC,   /* NoSupsInPIandDC */ /*** DA_COI ***/
   PID,          /* PIDefined */
   PMD,       /* ProjectMembersDefined */
   QP,           /* QualifierPresent */
   SPNDT,        /* SomePurposeNotDirectTreatment */
   SQD,          /* SomeQueriesDefined */
   SSD           /* SomeSourcesDefined */
extends DecisionRule {}

one sig
   AllowDeIDed,
   TotallyDeIDed,
   TotallyIDed
extends DataTransform {}

sig Project extends DataSource{}
one sig ClinicalDB extends DataSource{}

one sig Fishing extends Licence {}

one sig DeIDed,
   Identified
extends AccessTicket {}

one sig
   DirectTreatment,
   Research
extends Purpose{}

/********** ********** ********** ********** **********
   subset concrete signatures
********** ********** ********** ********** **********/
sig
   Researcher
in Personnel{}

/********** ********** ********** ********** **********
   NJH Closed System
********** ********** ********** ********** **********/
sig NJH {
   accessTickets: set AccessTicket,
   decisionRules: set DecisionRule,
```

```
  licences: set Licence,
  permissions: set Permission,
  personnel: set Personnel,
  projects: set Project,
  purposes: set Purpose,
  queries: set Query,
  researchers : set Researcher,
  rules: set Rule,
  sources: set DataSource,
  transforms: set DataTransform,

  /* helps to determine
     1. if data from a project can be used as a data source */
  ATPriority : accessTickets -> accessTickets,

  // p1->p2 means p1 gives p2 access to data produced by p1
  dataAccessAgreement: projects -> projects,

  /* permission has applicable decision and access rules that must be
     applied to approve the licence or to access the data. */
  permRules: permissions -> some rules,

  /* project access tickets, each one has at most one */
  projectAT: projects -> lone accessTickets,

  /* project data collector, each project has at most one */
  projectDataCollector: projects -> lone personnel,

  projectDataTransformRequired: projects -> one transforms,

  /* project team members */
  projectMembers: projects -> researchers,

  /* project principal investigator */
  projectPI: projects -> lone researchers,

  /* project purpose */
  projectPurpose: projects -> lone purposes,

  /* project queries */
  projectQueries: projects one -> queries,

  /* project sources, could be other projects too */
  projectSources: projects -> sources,

  /* researcher licence */
  researcherL: researchers -> lone licences,

  /* supervisors, each personnel has at most one supervisor */
  supervisors: personnel lone -> personnel }

/********** ********** ********** ********** ********** ********** **********
     End Structural Model, NJHg
/********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
  Begin INVARIANTS
/********** ********** ********** ********** ********** ********** **********/

```

```
/********** ********** ********** ********** **********
  INVARIANTS
   separating the invariants for each set, relation,
   or related sets and relations allows for
   easier decomposition later on when slicing
********** ********** ********** ********** **********/
/* this predicate is exported from the model, to be used in inv[] */
pred inv (njh: NJH) {
   all
      njh: NJH |

   /** for sets */
   invPermissions[njh] and
   invPersonnel[njh] and
   invRules[njh] and
   invSources[njh] and

   /** for relations */
   invATRules[njh] and
   invATPriority[njh] and
   invDataAccessAggreement[njh] and
   invProjectAT[njh] and
   invProjectDataCollector[njh] and
   invProjectSources[njh] and
   invSupervisors[njh] }

/********** ********** ********** ********** **********
   Set invariants, ordered alphabetically by
   name of set used, as best as possible
********** ********** ********** ********** **********/

private pred invPermissions (njh: NJH) {
   njh.permissions = njh.accessTickets + njh.licences }

private pred invPersonnel (njh: NJH) {
   njh.researchers in njh.personnel}

private pred invRules (njh: NJH) {
   njh.rules = njh.decisionRules }

private pred invSources (njh: NJH) {
   njh.projects in njh.sources }

/********** ********** ********** ********** **********
   Relation invariants, ordered alphabetically by
   name of main relation used as best as
   possible
********** ********** ********** ********** **********/
private pred invATPriority (njh: NJH) {
      irreflexive[^(njh.ATPriority)] }

 /* p1->p2 means p1 gives p2 access to data produced by p1 */
private pred invDataAccessAggreement (njh: NJH) {
   /* no project has a data access agreement with itself */
   irreflexive[^(njh.dataAccessAgreement)]

   /* a project with a data access agreement with another
      project has that project as a data source */
   ~(njh.dataAccessAgreement) in njh.projectSources }

private pred invATRules (njh: NJH) {
   /* for approving of project access ticket */
```

```
   let
      dr =
         CUTD +
         DAAP+
         DSPOK +
         LTAPI +
         NOPITDC +
         NSIPIDC + /*** DA_COI ***/
         PID +
         PMD +
         SPNDT +
         SQD +
         SSD,
      di = dr - CUTD,
      d = DeIDed,
      i = Identified |

   /* specific for DeIDed access tickets */
   d.(njh.permRules) & njh.decisionRules = dr
   and
   /* specific for Identified access tickets */
   i.(njh.permRules) & njh.decisionRules = di }

private pred invCUTD(njh: NJH, p: Project, at: AccessTicket) {
   some at->CUTD & njh.permRules implies (
      (some at & Identified iff
         /* kind of Transformation access ticket allows,
            mixed, AllowDeIDed
               or
            TotallyIDed, no deidentification allowed */
         some p.(njh.projectDataTransformRequired) & (TotallyIDed + AllowDeIDed)) or
      (some at & DeIDed iff
         // kind of Transformation access ticket allows, totally deidentified
         some p.(njh.projectDataTransformRequired) & TotallyDeIDed) ) }

private pred inv_DAAP_DSPO(njh:NJH, p: Project, at: AccessTicket) {
   all
      ps: p.(njh.projectSources) & njh.projects | {
   (some at->DAAP & njh.permRules and some ps) implies
         some ps -> p & njh.dataAccessAgreement

   /* if access ticket being considered has priority over
   the access tickets of any of its project sources
   (i.e. other projects) }then we cannot approve the
   project because the data returned would not be at the level required */
   (some at->DSPOK & njh.permRules and some ps) implies
      some ps.(njh.projectAT) and
         no at-> ps.(njh.projectAT) & njh.ATPriority  }}

private pred inv_LTAPI_NOPITDC_PMD_PID(njh: NJH, p:Project, at: AccessTicket) {
   let
      team = p.(njh.projectMembers),
      pi = p.(njh.projectPI),
      dc = p.(njh.projectDataCollector) | {

   all
      r: (team + pi) | {
   /* application of the LTAPI Decision Rule
         each pi and team member has a licence */
      some at->LTAPI & njh.permRules implies
         some r.(njh.researcherL) }

```

```
  /* application of the NOPITDC Decision Rule */
    some at -> NOPITDC & njh.permRules implies (
      /* 1. neither pi nor dc are a part of project team */
      (no (pi + dc) & team and
      // 2. pi and da are not the same
      no pi & dc ) and ( /*** DA_COI ***/
      let
        ps = p.(^(njh.projectSources)) & Project |
      no pi & ps.(njh.projectDataCollector) and
      no team & ps.(njh.projectDataCollector) )
    )

  /* application of the PMD Decision Rule
    > 1 team members */
    some at -> PMD & njh.permRules implies #team > 0

  /* application of the PID Decision Rule has a pi */
    some at -> PID & njh.permRules implies #pi> 0}}

/*** DA_COI ***/
/* application of the NSIPIDC Decision Rule
    the pi does not supervise the dc directly or indirectly */
private pred invNSIPIDC (njh: NJH, p: Project, at: AccessTicket) {
  let
    ps = p.(^(njh.projectSources)) & Project |
  some at -> NSIPIDC & njh.permRules implies (
    no p.(njh.projectPI) -> p.(njh.projectDataCollector) &
      ^(njh.supervisors) and (
    some ps implies
      no p.(njh.projectPI) -> (p+ps).(njh.projectDataCollector) &
        ^(njh.supervisors) )
  ) }

private pred invSPNDT (njh: NJH, p: Project, at: AccessTicket ) {
  /* application of the SPNDT Decision Rule
     project purpose is not for direct treatment */
    some at -> SPNDT & njh.permRules implies
      p.(njh.projectPurpose) != DirectTreatment }

private pred invSQD (njh: NJH, p: Project, at: AccessTicket ) {
  /* application of the SQD Decision Rule
     at least one project query */
  some at -> SQD & njh.permRules implies
    some p.(njh.projectQueries) }

private pred invSSD (njh: NJH, p: Project, at: AccessTicket ) {
  /* application of the SSD Decision Rule
     at least one project source */
  some at -> SSD & njh.permRules implies
    some p.(njh.projectSources) }

private pred invProjectAT (njh: NJH) {
  all
    p: njh.projects |
  let
    pat = njh.projectAT,
    at = p.pat |

  some p.pat implies (
    invCUTD[njh, p, at] and
    inv_DAAP_DSPO[njh, p, at] and
    inv_LTAPI_NOPITDC_PMD_PID[njh, p, at] and
```

```
      invNSIPIDC[njh, p, at] and /*** DA_COI ***/
      invSPNDT[njh, p, at] and
      invSQD[njh, p, at] and
      invSSD[njh, p, at] ) }

private pred invProjectDataCollector(njh: NJH) {
   all
      p: njh.projects |
   /* ClinicalDB iff DataCollector */
   (some p->ClinicalDB & njh.projectSources) iff
      (some p.(njh.projectDataCollector) ) }

private pred invProjectSources1 (njh: NJH) {
   // no self datasource for projects, directly or indirectly
   irreflexive[^(njh.projectSources :> njh.projects)] }

private pred invProjectSources2 (njh: NJH ) {
   all
      p: njh.projects |
   some p.(njh.projectAT) implies (
      /* all data sources for a project that are projects themselves
         should be (already) approved when the project gets it's
         access ticket */
   some (p.(njh.projectSources) & Project) implies
      all
         ps: (p.(njh.projectSources) & Project) |
      some ps.(njh.projectAT)
   ) }

private pred invProjectSources (njh: NJH) {
   invProjectSources1[njh] and
   invProjectSources2[njh] }

private pred invSupervisors (njh: NJH) {
   /* no cycles in supervisor relations, */
   irreflexive[^(njh.supervisors)]
   /* all personnel are either supervisor or supervised */
   all
      p: njh.personnel | {
    p in (dom[njh.supervisors] + ran[njh.supervisors])} and
   /* supervisor relation is a single tree, i.e. not a forest
      this means that one personel has no supervisor */
   one
      sup: njh.personnel |
   no (njh.supervisors).sup }

/********** ********** ********** ********** ********** ********** **********
      End INVARIANTS
/********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
   Partial instance CONFIGURATION,
      these will be instantiated in every instance
********** ********** ********** ********** ********** ********** **********/
pred setPartialInstanceConfiguration (njh: NJH) {

   /*********** for sets */
   njh.decisionRules = /* (12) */
      CUTD +
```

```
      DAAP +
      DSPOK +
      LTAPI +
      NOPITDC +
      NSIPIDC + /*** DA_COI ***/
      PID +
      PMD +
      SPNDT +
      QP +
      SQD +
      SSD  and

   /* access tickets (2) */
   njh.accessTickets =
      DeIDed +
      Identified and

   /* licences (1) */
   njh.licences = Fishing and

   /* transforms (3) */
   njh.transforms =
      AllowDeIDed +
      TotallyDeIDed +
      TotallyIDed and

   /* sources (at least 1) */
   some ClinicalDB & njh.sources and

   /*********** for relations */
   /* access ticket priority (1) */
   njh.ATPriority = Identified -> DeIDed and

   /* permRules: permissions -> some rules (22) */
   njh.permRules =
      /* decision rules for fishing licence (1) */
      Fishing -> QP +

      /* decision rules for DeIDed access ticket (11) */
      DeIDed -> CUTD +
      DeIDed -> DAAP+
      DeIDed -> DSPOK +
      DeIDed -> LTAPI +
      DeIDed -> NOPITDC +
      DeIDed -> NSIPIDC + /*** DA_COI ***/
      DeIDed -> PID +
      DeIDed -> PMD +
      DeIDed -> SPNDT +
      DeIDed -> SQD +
      DeIDed -> SSD +

      /* decision rules for Identified access ticket (10) */
      Identified -> DAAP+
      Identified -> DSPOK +
      Identified -> LTAPI +
      Identified -> NOPITDC +
      Identified -> NSIPIDC + /*** DA_COI ***/
      Identified -> PID +
      Identified -> PMD +
      Identified -> SPNDT +
      Identified -> SQD +
      Identified -> SSD and
```

```

  /** Important to add these so that Alloy does not use a
     subset of the configuration !!!
     In general this is important when using Alloy to set
     object configurations */
  //#njh.decisionRules = 11and /*** DA_COI ***/
  #njh.decisionRules = 12and
  #njh.accessTickets = 2and
  #njh.licences = 1and
  #njh.sources > 0and
  #njh.transforms = 3and
  #njh.ATPriority = 1//and
  //eq[#njh.permRules, 22] /* This produces an error ! */
}

/********** ********** ********** ********** ********** ********** **********
  end partial instance configuration,
********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** ********** ********** **********
  MODEL Instances - These are required in the op specifications
     someOfAllRelationsSatisfyingInvAndConfiguration is used in init
********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
  Can we get an instance of the model for all
  the sets?
********** **sets******* ********** ********** **********/
private pred someOfAllSets(njh: NJH) {
  some njh.accessTickets and
  some njh.decisionRules and
  some njh.licences and
  some njh.permissions and
  some njh.personnel and
  some njh.projects and
  some njh.purposes and
  some njh.queries and
  some njh.researchers and
  some rules and
  some njh.sources and
  some njh.transforms  }
//run someOfAllSets for 7but 1NJH expect 1

/********** ********** ********** ********** **********
  Can we get an instance of the model for all
  the relations?
********** ********** ********** ********** **********/
private pred someOfAllRelations(njh: NJH) {
  some njh.ATPriority and
  some njh.dataAccessAgreement and
  some njh.permRules and
  some njh.projectAT and
  some njh.projectDataCollector and
  some njh.projectDataTransformRequired and
  some njh.projectMembers and
  some njh.projectPI and
  some njh.projectPurpose and
  some njh.projectQueries and
  some njh.projectSources and
  some njh.researcherL and
  some njh.supervisors }
//run someOfAllRelations for 7but 1NJH expect 1
```

```

/********** ********** ********** ********** **********
   Can we get an instance of the model for all
   the relations that satisfy generator[]?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingInvAndConfig (njh: NJH) {
      someOfAllRelations[njh] and
         someOfAllSets[njh] and
            inv[njh] and
               setPartialInstanceConfiguration[njh] }
run someOfAllRelationsSatisfyingInvAndConfig
   //for 7but exactly 11Rule, 1NJH expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule, 1NJH expect 1

/********** ********** ********** ********** ********** ********** **********
   End MODEL Instances
********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
   OPERATION Specs
********** ********** ********** ********** ********** ********** **********/
private pred noChangeSets (njh, njh': NJH) {
   njh.accessTickets = njh'.accessTickets and
   njh.decisionRules = njh'.decisionRules and
   njh.licences = njh'.licences and
   njh.permissions = njh'.permissions and
   njh.personnel = njh'.personnel and
   njh.projects = njh'.projects and
   njh.purposes = njh'.purposes and
   njh.queries = njh'.queries and
   njh.researchers = njh'.researchers and
   njh.rules = njh'.rules and
   njh.sources = njh'.sources and
   njh.transforms = njh'.transforms }

private pred noChangeRelations(njh, njh': NJH) {
   njh.ATPriority = njh'.ATPriority and
   njh.dataAccessAgreement = njh'.dataAccessAgreement and
   njh.permRules = njh'.permRules and
   njh.projectAT = njh'.projectAT and
   njh.projectDataCollector = njh'.projectDataCollector and
   njh.projectDataTransformRequired = njh'.projectDataTransformRequired and
   njh.projectMembers = njh'.projectMembers and
   njh.projectPI = njh'.projectPI and
   njh.projectPurpose = njh'.projectPurpose and
   njh.projectQueries = njh'.projectQueries and
   njh.projectSources = njh'.projectSources and
   njh.researcherL = njh'.researcherL and
   njh.supervisors = njh'.supervisors }

private pred skip(njh, njh': NJH){
   /** Sets */
   noChangeSets[njh, njh'] and

   /** Relations */
   noChangeRelations[njh, njh'] }

pred approveProjectAT (njh, njh': NJH, p: Project, at: AccessTicket) {
   /** Pre-conditions */
```

```
  p in njh.projects and
  at in njh.accessTickets and
  no p->at & njh.projectAT and

  /** Post-conditions */

  /* Applying Decision Rules */
  inv_DAAP_DSPO[njh, p, at] and
  inv_LTAPI_NOPITDC_PMD_PID[njh, p, at] and
  invNSIPIDC[njh, p, at] and /*** DA_COI ***/
  invSPNDT[njh, p, at] and
  invSQD[njh, p, at] and
  invSSD[njh, p, at] and

  /* No change to sets */
  noChangeSets[njh, njh'] and

  /* These relations do not change */
  njh.ATPriority = njh'.ATPriority and
  njh.dataAccessAgreement = njh'.dataAccessAgreement and
  njh.permRules = njh'.permRules and
  njh.projectDataCollector = njh'.projectDataCollector and
  njh.projectMembers = njh'.projectMembers and
  njh.projectPI = njh'.projectPI and
  njh.projectPurpose = njh'.projectPurpose and
  njh.projectQueries = njh'.projectQueries and
  njh.projectSources = njh'.projectSources and
  njh.researcherL = njh'.researcherL and
  njh.supervisors = njh'.supervisors and

  /* These relations change */
  njh'.projectAT = njh.projectAT + p->at and

  /* Changes ensures the correct Data Transform exists */
  (some at & DeIDed iff
     njh'.projectDataTransformRequired =
        njh.projectDataTransformRequired + p->TotallyDeIDed ) and

  (some at & Identified iff
     (njh'.projectDataTransformRequired =
        njh.projectDataTransformRequired + p->TotallyIDed or
     njh'.projectDataTransformRequired =
        njh.projectDataTransformRequired + p-> AllowDeIDed )
  ) }

private pred ProjectApprovePossible(
  njh, njh': NJH,
  proj: Project,
  at: AccessTicket) {
  let
     first = ord/first |
  someOfAllRelationsSatisfyingInvAndConfig[njh] and
  some proj & first.projects and
  some at & first.permissions and
  approveProjectAT[njh, njh', proj, at]  and
  inv[njh] and
  inv[njh'] }
run ProjectApprovePossible
  //for 7but exactly 11Rule, 2 NJH expect 1/*** DA_COI ***/
  for 7 but exactly 12Rule, 2 NJH expect 1

// this is how we initialise the system
```

```
pred init(njh: NJH) {
   some p: Project |
      p in njh.projects and
            someOfAllRelationsSatisfyingInvAndConfig[njh] and
               no p.(njh.projectAT) }
run init
   //for 7but exactly 11Rule, 1NJH expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule, 1NJH expect 1

/** this is how we move from instance to instance */
fact traces {
   init[ord/first]
   all
      njh: NJH - ord/last |
   some
      p: Project,
      at: AccessTicket |
   let
      njh' = njh.next |
   approveProjectAT[njh, njh', p, at] or
      skip[njh, njh'] }

assert OpPreserves {
   all njh, njh': NJH |
      all p: Project, at: AccessTicket |
         (inv[njh] and approveProjectAT [njh, njh', p, at]) implies inv[njh'] }
check OpPreserves
   //for 7but exactly 11Rule expect 0/*** DA_COI ***/
   for 7 but exactly 12Rule expect 0

/** run only when opPreserves returns a counterexample */
pred OpDoesNotPreserve[njh, njh': NJH, r: Researcher, p: Project, at: AccessTicket ]{
   inv[njh] and approveProjectAT[njh, njh', p, at] and not inv[njh'] }
run OpDoesNotPreserve
   //for 7but exactly 2NJH, 11Rule expect 0/*** DA_COI ***/
   for 7 but exactly 2NJH, 12Rule expect 0

/********** ********** ********** ********** ********** ********** **********
 END OPERATION Specs
********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
   Internal NJH Conformance Rules
********** ********** ********** ********** ********** ********** **********/
/** This predicates, generator1 and generator2 are used in this section */
private pred generator1 (njh: NJH, p: Project) {
   some p.(njh.projectAT) and
      inv[njh] }

private pred generator2 (njh: NJH, p: Project) {
   generator1[njh, p] and
      someOfAllRelations[njh] and
         setPartialInstanceConfiguration[njh] }

assert NoInProjectNSIPIDC_Sups{
   all
      njh: NJH, p: Project |
   generator1[njh, p] implies
      no p.(njh.projectPI) -> p.(njh.projectDataCollector) &
```

```
        ^(njh.supervisors) }
check NoInProjectNSIPIDC_Sups
   //for 7but exactly 11Rule expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule expect 0

assert NoInSourcesNSIPIDC_Sups{
   all
      njh: NJH, p: Project |
   let
      ps = p.(^(njh.projectSources)) & Project |
   (generator1[njh, p] and some ps) implies
      no p.(njh.projectPI) -> (p+ps).(njh.projectDataCollector) &
         ^(njh.supervisors) }
check NoInSourcesNSIPIDC_Sups
   //for 7but exactly 11Rule expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule expect 0

assert NoInSourcesNSIPIDC_PIandDC{
   all
      njh: NJH, p: Project |
   let
      ps = p.(^(njh.projectSources)) & Project |
   (generator1[njh, p] and some ps) implies
      no p.(njh.projectPI) & ps.(njh.projectDataCollector) }
check NoInSourcesNSIPIDC_PIandDC
   //for 7but exactly 11Rule expect 1/*** DA_COI ***/
    for 7 but exactly 12Rule expect 0

assert NoInSourcesNSIPIDC_MEMSandDC{
   all
      njh: NJH, p: Project |
   let
      ps = p.(^(njh.projectSources)) & Project |
   (generator1[njh, p] and some ps) implies
      no p.(njh.projectMembers) & ps.(njh.projectDataCollector) }
check NoInSourcesNSIPIDC_MEMSandDC
   //for 7but exactly 11Rule expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule expect 0

/********** ********** ********** ********** **********
   Can we get an instance of the model for all
   the relations that satisfy inv[] and a
   project has a DeIDed access Ticket
   and a project where there is some suspicious
   relationship with the dataColector?
********** ********** ********** ********** **********/

/** 1. PI directly supervises DataCollector */
private pred DataCollectorICOI11(njh: NJH, p: Project){
   generator2[njh, p] and
      some p.(njh.projectAT) and
         some p.(njh.projectPI) -> p.(njh.projectDataCollector) &
            (njh.supervisors) }
run DataCollectorICOI11 for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NSIPIDC rule is commented */
   //but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   but  exactly 12Rule, 1NJH expect 0

/** 1. PI indirectly supervises DataCollector */
private pred DataCollectorICOI12(njh: NJH, p: Project){
   generator2[njh, p] and
```

```
      some p.(njh.projectAT) and
         some p.(njh.projectPI) -> p.(njh.projectDataCollector) &
            ^(njh.supervisors) and
            no p.(njh.projectPI) -> p.(njh.projectDataCollector) &
               (njh.supervisors)}
run DataCollectorICOI12 for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NSIPIDC rule is commented */
   but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   //but exactly 12Rule, 1NJH expect 0

/** 2. PI supervises DataCollector on direct ProjectSource */
private pred DataCollectorCOI21(njh: NJH, p: Project){
   let
         ps = p.(^(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectPI) -> (p+ps).(njh.projectDataCollector) &
            ^(njh.supervisors)}
run DataCollectorCOI21 for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NSIPIDC rule is commented */
   but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   //but exactly 12Rule, 1NJH expect 0

/** 3. PI directly supervises DataCollector on indirect ProjectSource */
private pred DataCollectorCOI22Indirect(njh: NJH, p: Project){
   let
      ps = ((p.(njh.projectSources) & Project).(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectPI) -> ps.(njh.projectDataCollector) &
            (njh.supervisors)}
run DataCollectorCOI22Indirect for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NSIPIDC rule is commented */
   //but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   but  exactly 12Rule, 1NJH expect 0

/** 3. PI indirectly supervises DataCollector on indirect ProjectSource */
private pred DataCollectorCOI23Indirect(njh: NJH, p: Project){
   let
      ps = ((p.(njh.projectSources) & Project).(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectPI) -> ps.(njh.projectDataCollector) &
               ^(njh.supervisors) and
               no p.(njh.projectPI) -> ps.(njh.projectDataCollector) &
                  (njh.supervisors) }
run DataCollectorCOI23Indirect for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NSIPIDC rule is commented */
   //but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   but  exactly 12Rule, 1NJH expect 0


/**4. PI is Data Collector on ProjectSource */
private pred DataCollectorICOI31(njh: NJH, p: Project){
   let
```

```
      ps = p.(^(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectPI) & ps.(njh.projectDataCollector) }
run DataCollectorICOI31 for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NOPITDC rule is commented */
   //but exactly 12Rule, 4 Project, 1NJH expect 1/*** DA_COI ***/
   but exactly 12Rule, 1NJH expect 0

/**5. PI is Data Collector on ProjectSource */
private pred DataCollectorICOI32Indirect(njh: NJH, p: Project){
   let
      ps = ((p.(njh.projectSources) & Project).(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectPI) & ps.(njh.projectDataCollector) }
run DataCollectorICOI32Indirect for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NOPITDC rule is commented */
   //but exactly 12Rule, 4 Project, 1NJH expect 1 /*** DA_COI ***/
   but exactly 12Rule, 1NJH expect 0


/** 6. ProjectMember is Data Collector on ProjectSource */
private pred DataCollectorCOI41(njh: NJH, p: Project){
   let
      ps = p.(^(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectMembers) & ps.(njh.projectDataCollector) }
run DataCollectorCOI41 for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NOPITDC rule is commented */
   //but exactly 12Rule, 4Project, 1NJH expect 1/*** DA_COI ***/
   but exactly 12Rule, 1NJH expect 0

/** 7. ProjectMember is Data Collector on ProjectSource */
private pred DataCollectorCOI42Indirect(njh: NJH, p: Project){
   let
         ps = ((p.(njh.projectSources) & Project).(njh.projectSources)) & Project |
   some ps and
      generator2[njh, p] and
         some p.(njh.projectAT) and
            some p.(njh.projectMembers) & ps.(njh.projectDataCollector) }
run DataCollectorCOI42Indirect for 7
   //but exactly 11Rule, 3Project, 1NJH expect 1/*** DA_COI ***/
   /** use only when the applicable part of rule NOPITDC rule is commented */
   //but exactly 12Rule, 4 Project, 1NJH expect 1/*** DA_COI ***/
   but exactly 12Rule, 1NJH expect 0


/********** ********** ********** ********** ********** ********** **********
   End Internal NJH Conformance Rules
********** ********** ********** ********** ********** ********** **********/
```

```
/********** ********** ********** ********** ********** ********** **********
   These are not a part of the object configuration.
   They provide sanity checks
********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
    any instance of the model
********** ********** ********** ********** **********/
private pred showg (njh: NJH) {}
//run showg
   //for 7 but exactly 11Rule, 1NJH expect 1/*** DA_COI ***/
   //for 7 but exactly 12Rule, 1NJH expect 1

/********** ********** ********** ********** **********
   Can we get an instance of the model for all
   the relations that satisfy generator[] and a
   project has an Identified access Ticket?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingInvForIdentifiedAT(
   njh: NJH, at: Identified) {
   some njh.projectAT.at and
      someOfAllRelations[njh] and
         inv[njh] and
            setPartialInstanceConfiguration[njh] }
run someOfAllRelationsSatisfyingInvForIdentifiedAT
   //for 7 but exactly 11Rule, 1NJH expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule, 1NJH expect 1

/********** ********** ********** ********** **********
   Can we get an instance of the model for all
   the relations that satisfy generator[] and a
   project has a DeIDed access Ticket?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingInvForDeIDedAT (
   njh: NJH, at: DeIDed) {
   some njh.projectAT.at and
      someOfAllRelations[njh] and
         inv[njh] and
            setPartialInstanceConfiguration[njh]}
run someOfAllRelationsSatisfyingInvForDeIDedAT
   //for 7 but exactly 11Rule, 1NJH expect 1/*** DA_COI ***/
   for 7 but exactly 12Rule, 1NJH expect 1

/********** ********** ********** ********** **********
   all sets that are defined are used!
   using IFF instead of IMPLIES is not applicable
   because lone on some sides of the relations.
********** ********** ********** ********** **********/
assert TestIfAllSetsAreApplicableToTheModel {
   all
      njh: NJH |
   someOfAllRelationsSatisfyingInvAndConfig[njh] implies
      someOfAllSets[njh] }
check TestIfAllSetsAreApplicableToTheModel
   //for 7 but exactly 11Rule, 1NJH expect 0/*** DA_COI ***/
   for 7 but exactly 12Rule, 1NJH expect 0
```

D.1.2 Alloy Specifications for Slice 5 to Check Conformance

Listing D.2: *Updated Alloy Specifications for Slice 5 to CheckConformance*

```
/********** ********** ********** ********** ********** ********** **********
   Begin Structural Model, NJH slice 5

   Written By: Phillipa Bennett
   Version 5
   Date: Version 5completed Nov 28, 2016

   Notes:
   A lot of notes through out the specification!

/********** ********** ********** ********** ********** ********** **********/
module slice_5

/********** ********** ********** ********** **********
   imports
/********** ********** ********** ********** **********/
open util/relation
open util/ternary
open util/ordering[NJH] as ord

/********** ********** ********** ********** **********
   base abstract signatures
********** ********** ********** ********** **********/
abstract sig
   Data,
   DataSource,
   DataTransform,
   Name,
   Permission,
   Status {}

/********** ********** ********** ********** **********
   extended abstract signatures
********** ********** ********** ********** **********/
abstract sig
   AccessTicket
extends Permission {}

/********** ********** ********** ********** **********
   unextended concrete signatures
********** ********** ********** ********** **********/
sig Day,
   Month,
   Query,
   Year {}

sig DataItem {
   name: Name}

/********** ********** ********** ********** **********
   extended concrete signatures
********** ********** ********** ********** **********/
sig Age, Other extends Name {}

sig Project extends DataSource{}

```

```
lone sig DeIDed,
   Identified
extends AccessTicket {}

lone sig
   DownloadAllowed,
   DownloadDisabled
extends Status {}

sig Date extends Data {
   day: lone Day,
   month: lone Month,
   year: Year }{
   /*  day iff month also exists */
   some day iff some month }

one sig
   AllowDeIDed,
   TotallyDeIDed,
   TotallyIDed
extends DataTransform {}

/********** ********** ********** ********** **********
   subset concrete signatures
********** ********** ********** ********** **********/
sig
   QryData,
   RetData
in DataItem {}

/********** ********** ********** ********** **********
   NJH Closed System
********** ********** ********** ********** **********/
sig NJH {
   accessTickets: set AccessTicket,
   dataItems: set DataItem,
   dates: set Date,
   permissions: set Permission,
   projects: set Project,
   qryItems: set QryData,
   queries: set Query,
   retItems: set RetData,
   statuses: set Status,
   transforms: set DataTransform,
   values: set Data,

   /* data items must a value or not. */
   dataValues: dataItems -> one values,

   enteredOn: dataItems -> lone dates,

   /* project access tickets, each one has at most one */
   projectAT: projects -> lone accessTickets,

   // Transformation of the data required
   projectDataTransformRequired: projects -> lone transforms,

   /* project queries */
   projectQueries: projects one -> queries,

   /* a query can work on any kind of data item
      retData is in position 2*/
```

```
   qryReturns: queries -> retItems -> dataItems,

   /* determines is query results meets conformance and the next
      operation, i.e. view/download is allowed */
   VDAllowed: queries -> lone statuses }

/********** ********** ********** ********** ********** ********** **********
      End Structural Model, NJHg_slice_5
/********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** ********** ********** **********
  INVARIANTS
   separating the invariants for each set, relation,
   or related sets and relations allows for
   easier decomposition later on when slicing
/********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
   Some Functions and Predicates to be reused
   when writing invariants and generating
   instances/counterexamples
********** ********** ********** ********** **********/
private fun applicableDates(njh: NJH, q: Query): set Date {
   { Date &
      dom[q.(njh.qryReturns)].(njh.dataValues) +
         dom[q.(njh.qryReturns)].(njh.enteredOn) }}

private fun DeIDedDateTransform (d: Date): Date {
   {ri: Date |
      no ri.day and
      no ri.month and
      ri.year = d.year }}

private pred identifiedDate (d: Date) {
   some d.day }

private pred totallyIDedTransform (njh: NJH, q: Query) {
   all
      d: applicableDates[njh, q] |
   identifiedDate[d] }

private pred totallyDeIDedTransform (njh: NJH, q: Query) {
   all
      d: applicableDates[njh, q] |
   not identifiedDate[d] }

private pred allowDeIDedTransform (njh: NJH, q:Query) {
   all
      d: applicableDates[njh, q] |
   identifiedDate[d] or not identifiedDate[d]}

/********** ********** ********** ********** **********
   Set invariants, ordered alphabetically by
   name of set used, as best as possible
********** ********** ********** ********** **********/
private pred invDataItems (njh: NJH) {
   /* set up dataItems, keep out of inv because it is always true */
   (qryItems + retItems) = dataItems }

/* closed system constraint - any date is a part of the set of dates */
private pred invDates (njh: NJH) {
   njh.dates = (njh.values & Date) + ran[njh.enteredOn]
```

```
  all
    d: Date |
  (d in njh.dates and identifiedDate[d]) implies
    DeIDedDateTransform[d] in njh.dates}

private pred invPermissions (njh: NJH) {
  njh.permissions = njh.accessTickets }

/********** ********** ********** ********** **********
  Relation invariants, ordered alphabetically by
  name of main relation used as best as possible
********** ********** ********** ********** **********/
/** extracted from invCUTD in slice 3*/
private pred invProjectATDataTransform(njh: NJH) {
  all
    p: njh.projects |
  (some p.(njh.projectAT) & Identified iff
  /* kind of Transformation access ticket allows,
    mixed - AllowDeIDed or TotallyIDed */
    some p.(njh.projectDataTransformRequired) &
      (TotallyIDed + AllowDeIDed))
  and
  (some p.(njh.projectAT) & DeIDed iff
  /*kind of Transformation access ticket allows,
    totally deidentified */
    some p.(njh.projectDataTransformRequired) &
      TotallyDeIDed) }

private pred invQryReturnsAT (njh: NJH) {
  all
    q: njh.queries |
  some q.(njh.qryReturns) implies
    some njh.projectQueries.q.(njh.projectAT) }

/* if a query has a a VD status then it has some return data */
private pred invVDAllowedWithQueryResults (njh: NJH, q: Query) {
  (some q.(njh.VDAllowed) implies
    some q.(njh.qryReturns)) }

/* project with AllowDeIDed can never have a DownloadDisables
  status */
private pred invVDAllowedWithAllowDeIDed (
  njh: NJH, p: Project, q: Query) {
  some p.(njh.projectDataTransformRequired) & AllowDeIDed implies
    no  q->DownloadDisabled & njh.VDAllowed }

/**********
TotallyIDED
 **********/
/** using iff does not matter, i.e., all predicated/assertions
  give the expected results. */
private pred invDownloadAllowedTotallyIDed(
  njh: NJH, p: Project, q: Query) {
  some p.(njh.projectDataTransformRequired) & TotallyIDed implies
    totallyIDedTransform[njh, q] }

/** iff causes counterexample for HIPAADateConformanceDeIDed */
private pred invDownloadDisabledTotallyIDed(
  njh: NJH, p: Project, q: Query) {
  some p.(njh.projectDataTransformRequired) & TotallyIDed implies
    not totallyIDedTransform[njh, q] }
```

```
/**********
AllowDeIDED
 **********/
   /** Introducing a fault in invDownloadAllowedAllowIDed,
      We introcuce a fault in the connector for these clauses that
      allows the Identified access ticket with a TotallyIDeD transform
      to  give de-identified data.

   This fault causes the:
      1. showDeIDedNCDA, showIdentifiedNCTotallyIDedDA, and
         HIPAADateNonConformanceIdentified predicates to give
         instances, and
      2. HIPAADateConformanceIdentified and
         HIPAADateConformanceDeIDed assertions to produce
         counterexamples
   for the Identified access ticket. */

   /** for fault use implies instead of iff */

private pred invDownloadAllowedAllowIDed(
   njh: NJH, p: Project, q: Query) {
   some p.(njh.projectDataTransformRequired) & AllowDeIDed implies
      allowDeIDedTransform[njh, q] }

/**********
TotallyDeIDED
 **********/
/** using iff does not matter, i.e., all predicated/assertions
   give the expected results. */
private pred invDownloadAllowedTotallyDeIDed(
   njh: NJH, p: Project, q: Query) {
   some p.(njh.projectDataTransformRequired) & TotallyDeIDed implies
      totallyDeIDedTransform[njh, q] }

/** iff gives instances for
      showIdentifiedNCTotallyIDedDA and
      HIPAADateNonConformanceIdentified
   and counterexamples for
      HIPAADateConformanceIdentified
   all contrary to expectation*/
private pred invDownloadDisabledTotallyDeIDed(
   njh: NJH, p: Project, q: Query) {
   some p.(njh.projectDataTransformRequired) & TotallyDeIDed implies
      not totallyDeIDedTransform[njh, q] }

private pred invVDAllowedCondAllowed(
   njh: NJH, p: Project, q: Query) {
   let
      a = invDownloadAllowedTotallyIDed[njh, p, q],
      b = invDownloadDisabledTotallyIDed[njh, p, q],
      c = invDownloadAllowedAllowIDed[njh, p, q],
      d = invDownloadAllowedTotallyDeIDed[njh, p, q],
      e = invDownloadDisabledTotallyDeIDed[njh, p, q] | {

   some q->DownloadAllowed & njh.VDAllowed implies
      ((a and not b) or (d and not e) or c) }}

private pred invVDAllowedCondDisabled(
   njh: NJH, p: Project, q: Query) {
   let
      a = invDownloadAllowedTotallyIDed[njh, p, q],
      b = invDownloadDisabledTotallyIDed[njh, p, q],
```

```
      d = invDownloadAllowedTotallyDeIDed[njh, p, q],
      e = invDownloadDisabledTotallyDeIDed[njh, p, q] | {

   some q->DownloadDisabled & njh.VDAllowed implies
      ((not a and b) or (not d and e)) }}

/**********
VDAllowed for all queries
 **********/
/* this is how VDAllowed is well formed for all queries */
private pred invVDAllowed (njh: NJH) {
   all
      q: njh.queries |
   let
      p = njh.projectQueries.q | {

   invVDAllowedWithQueryResults[njh, q]

   invVDAllowedWithAllowDeIDed[njh, p, q]

   no  q.(njh.VDAllowed) or {
      invVDAllowedCondAllowed[njh, p, q]
      invVDAllowedCondDisabled[njh, p, q] }}}

/********** ********** ********** ********** **********
   the FACTS
********** ********** ********** ********** **********/
private pred inv (njh: NJH) {
   all
      njh: NJH |

   /** for sets */
   invDataItems[njh] and
   invDates[njh] and
   invPermissions[njh] and

   /** for relations */
   invProjectATDataTransform[njh] and
   invQryReturnsAT[njh] and
   invVDAllowed[njh] }
//run inv for 7expect 1

/*fact {all njh: NJH | inv[njh] }*/

/********** ********** ********** ********** ********** ********** **********
      End of INVARIANTS
/********** ********** ********** ********** ********** ********** **********/



/********** ********** ********** ********** ********** ********** **********
   Start of Predicates for MODEL Instances that are a part of the
   operation specifications
********** ********** ********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
   Can we get an instance of the model for all
   the relations?
********** ********** ********** ********** **********/
private pred someOfAllRelations(njh: NJH) {
   some njh.dataValues and
   some njh.enteredOn and
```

```
  some njh.projectAT and
  some projectDataTransformRequired and
  some njh.projectQueries and
  some njh.qryReturns /*and */
  /** comment some VDAllowed when using operation specs
    to allow CheckConformance to get and instance
    It may break TestIfAllSetsAreApplicableToTheModel
    assertion, but that's ok */
  /*some njh.VDAllowed */ }

/********** ********** ********** ********** **********
  Can we get an instance of the model for all
  the relations that satisfy generator[]?
********** ********** ********** ********** **********/
private pred someOfAllRelationsSatisfyingInvAndConfig_DeIDed (
  njh: NJH) {
  someOfAllRelations[njh] and
    inv[njh] and
      setPartialInstanceConfig_DeIDed[njh] }

private pred someOfAllRelationsSatisfyingInvAndConfig_Identified (
  njh: NJH) {
  someOfAllRelations[njh] and
    inv[njh] and
      setPartialInstanceConfig_Identified[njh] }

private pred someOfAllRelationsSatisfyingInvAndConfig (
  njh: NJH) {
  someOfAllRelations[njh] and
    inv[njh] and
      setPartialInstanceConfig [njh] }

/*run someOfAllRelations for
  7 but 1 NJH expect 1
run someOfAllRelationsSatisfyingInvAndConfig_DeIDed for
  7 but  1 NJH expect 1
run someOfAllRelationsSatisfyingInvAndConfig_Identified
  for 7 but 1 NJH expect 1
run someOfAllRelationsSatisfyingInvAndConfig for
  7 but 1 NJH expect 1*/

/********** ********** ********** ********** **********
  Just sanity check. These 2checks can be
  removed from the model-
    the TestIfAllSetsAreApplicableToTheModel
    assertion checks that in all instances where
    the relations are non-empty, the invariants
     and the partial configuration ensures that
    all the sets defined are used!

    using IFF instead of IMPLIES in
      TestIfAllSetsAreApplicableToTheModel
    is not applicable because lone on some sides
    of the relations.
********** ********** ********** ********** **********/

/********** ********** ********** ********** **********
  Can we get an instance of the model for all
  the sets?
********** **sets******* ********** ********** **********/
private pred someOfAllSets(njh: NJH) {
  (some njh.accessTickets or
```

```
      some njh.permissions ) and
   (some njh.dataItems or
      (some njh.qryItems and
         some njh.retItems )) and
   some njh.dates and
   some njh.projects and
   some njh.queries and
   some njh.statuses and
   some transforms and
   some njh.values }

assert TestIfAllSetsAreApplicableToTheModel {
   all
      njh: NJH |
   (someOfAllRelationsSatisfyingInvAndConfig[njh] and
      someOfAllRelations[njh]) implies
   someOfAllSets[njh] }

/*run someOfAllSets for 7but 1NJH expect 1
check TestIfAllSetsAreApplicableToTheModel for 7expect 0*/

/********** ********** ********** ********** ********** ********** **********
   End of Predicates for MODEL Instances that are a part of the
   operation specifications
********** ********** ********** ********** ********** ********** ***********/




/********** ********** ********** ********** ********** ********** **********
      Start of OPERATION Specifications
/********** ********** ********** ********** ********** ********** **********/
/** this is how we initialise the system */
private pred init(njh: NJH) {
   some
      q: Query |
   some q.(njh.qryReturns) and
      no q.(njh.VDAllowed) and
         someOfAllRelationsSatisfyingInvAndConfig[njh] }

private pred noChangeSets(njh, njh': NJH) {
   njh.accessTickets = njh'.accessTickets and
   njh.dataItems = njh'.dataItems and
   njh.dates = njh'.dates and
   njh.permissions = njh'.permissions and
   njh.projects = njh'.projects and
   njh.qryItems = njh'.qryItems and
   njh.queries = njh'.queries and
   njh.retItems = njh'.retItems and
   njh.statuses = njh'.statuses and
   njh.transforms = njh'.transforms and
   njh.values = njh'.values }

private pred noChangeRelations(njh, njh': NJH) {
   njh.dataValues = njh'.dataValues and
   njh.enteredOn = njh'.enteredOn and
   njh.projectAT = njh'.projectAT and
   njh.projectDataTransformRequired =
      njh'.projectDataTransformRequired and
   njh.projectQueries = njh'.projectQueries and
   njh.qryReturns = njh'.qryReturns and
   njh.VDAllowed = njh'.VDAllowed }
```

```
/** i.e., specification of no operation */
private pred skip (njh, njh': NJH) {
  noChangeSets[njh, njh'] and
    noChangeRelations[njh, njh'] }

private pred checkConformance (njh, njh': NJH, p: Project, q: Query) {
  /*let
    at = p.(njh.projectAT) |*/ /** at are implied by the transtorm */
  /** Pre-conditions */
  p in njh.projects and
  q in p.(njh.projectQueries) and
  no q.(njh.VDAllowed) and
  some q.(njh.qryReturns) and

  /** Post-conditions - Frame Conditions*/
  noChangeSets[njh, njh'] and

  njh.dataValues = njh'.dataValues and
  njh.enteredOn = njh'.enteredOn and
  njh.projectAT = njh'.projectAT and
  njh.projectDataTransformRequired =
    njh'.projectDataTransformRequired and
  njh.projectQueries = njh'.projectQueries and
  njh.qryReturns = njh'.qryReturns and

  /** Post-conditions - Changes*/
  njh.VDAllowed = njh'.VDAllowed - (q->DownloadAllowed +
    q->DownloadDisabled) and
  some q.(njh'.VDAllowed) and (
  let
    a = invDownloadAllowedTotallyIDed[njh, p, q],
    b = invDownloadDisabledTotallyIDed[njh, p, q],
    c = invDownloadAllowedAllowIDed[njh, p, q],
    d = invDownloadAllowedTotallyDeIDed[njh, p, q],
    e = invDownloadDisabledTotallyDeIDed[njh, p, q] | {

  some q->DownloadAllowed & njh'.VDAllowed implies
    ((a and not b) or (d and not e) or c)

  some q->DownloadDisabled & njh'.VDAllowed implies
    ((not a and b) or (not d and e)) }) }

private pred CheckConformancePossible(
  njh, njh': NJH,
  p: Project,
  q: Query) {
  someOfAllRelationsSatisfyingInvAndConfig[njh] and
    checkConformance[njh, njh', p, q] and
      inv[njh'] }

/** this is how we move from instance to instance */
fact traces {
  init[ord/first]
  all
    njh: NJH - ord/last,
    p: Project,
    q: Query |
  let
    njh' = njh.next |
  skip[njh, njh'] or
    checkConformance[njh, njh', p, q] }
```

```
assert OpPreserves {
   all
      njh, njh': NJH ,
      p: Project, q: Query |
   (inv[njh] and
      checkConformance [njh, njh', p, q]) implies
   inv[njh'] }

/** run only when opPreserves returns a counterexample */
pred OpDoesNotPreserve[njh, njh': NJH, p: Project, q: Query ]{
   inv[njh] and
      checkConformance[njh, njh', p, q] and
         not inv[njh'] }

/*run init for 7but 1NJH expect 1
run skip for 7but 3NJH expect 1*/
run checkConformance for 7but 2NJH expect 1/*
run CheckConformancePossible for 7but 2NJH expect 1*/
check OpPreserves for 7expect 0
run OpDoesNotPreserve for 7expect 0

/********** ********** ********** ********** ********** ********** **********
   End Operation Specification
********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
   Partial instance CONFIGURATION,
      these will be instantiated in every instance
********** ********** ********** ********** ********** ********** **********/

/** We want to generate a small model. It is mportant to add the
   the size of the set so that Alloy does not use a subset of the
   configuration. */
private pred config_overlap (njh: NJH) {
   /*********** for sets */
   njh.dataItems.name =
      Age + Other and

   /* transforms (3) */
   njh.transforms =
      AllowDeIDed +
      TotallyDeIDed +
      TotallyIDed and

   /* statuses (2) */
   njh.statuses =
      DownloadAllowed +
      DownloadDisabled and

   /*********** for relations */
   #dataItems >= 6and
   #njh.dataItems.name = 2and
   #njh.transforms = 3and
   #njh.projects > 0and
   #qryItems >= 1and
   #queries > 1and
   #retItems >= 3and
   #njh.statuses >1 and
```

```
  #njh.transforms = 3and

  /* all projects have an access ticket */
  all
    p: njh.projects |{
  some p.(njh.projectAT)} and

  /* qryItems and retItems are distinct data */
  no njh.qryItems & njh.retItems and

  /* all qryItems are used to construct the return data */
  ran[select13[njh.qryReturns]] = njh.qryItems and

  /* all retItems are returned */
  ran[select12[njh.qryReturns]] = njh.retItems and

  /* all qryItems are identified dates */
  all
    q: njh.qryItems | {
  identifiedDate[q.(njh.dataValues)] }and

  /* there is only one retItem that is de-identified */
  #{r: njh.retItems | not identifiedDate[r.(njh.dataValues)]}= 1and

  /* the identified retItem and its associated dataItem have
    name = Age */
  all
    r: njh.retItems | {
  identifiedDate[r.(njh.dataValues)] implies
    r.name = Age and
      r.(select23[njh.qryReturns]).name = Age }and

  /* the not identified retItems and their associated dataItem have
    name = Other */
  all
    r: njh.retItems | {
  not identifiedDate[r.(njh.dataValues)] implies
    r.name = Other and r.(select23[njh.qryReturns]).name = Other }}

private pred setPartialInstanceConfig_DeIDed (njh: NJH) {
  config_overlap[njh] and

  /* access tickets (1) */
  njh.accessTickets = DeIDed and
  #njh.accessTickets = 1}

private pred setPartialInstanceConfig_Identified (njh: NJH) {
  /* load the overlap */
  config_overlap[njh] and

  /* access tickets (1) */
  njh.accessTickets = Identified and
  #njh.accessTickets = 1}

private pred setPartialInstanceConfig (njh: NJH) {
  /* load the overlap */
  config_overlap[njh] and

  /* access tickets (2) */
  njh.accessTickets = Identified + DeIDed and
  #njh.accessTickets = 2}
```

```
/*run config_overlap for 7expect 1
run setPartialInstanceConfiguration_DeIDed for 7expect 1
run setPartialInstanceConfiguration_Identified for 7expect 1
run setPartialInstanceConfiguration for 7expect 1*/

/********** ********** ********** ********** ********** ********** **********
     End of Partial Configuration
/********** ********** ********** ********** ********** ********** **********/




/********** ********** ********** ********** ********** ********** **********
   Start of Predicates/Assertions for other MODEL Instances
********** ********** ********** ********** ********** ********** **********/
private pred common_inst(
   njh: NJH, proj: Project, qry: Query, at: AccessTicket) {
   inv[njh] and
   some
      p: njh.projects |
   p = proj and
      p in njh.projects and
         p->at in njh.projectAT and
            some q: Query |
               q = qry and
                  some p->q & njh.projectQueries and
                     some q.(njh.qryReturns) }

/********** ********** ********** ********** **********
   AT: DeIDED
   Transform: well formed instances imply it
   Query Status: DD
   Conformance: yes
********** ********** ********** ********** **********/
private pred showDeIDedDD (
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, DeIDed] and
         some q->DownloadDisabled & njh.VDAllowed and
            not totallyDeIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   AT: IDED
   Transform: TotallyIDed
   Query Status: DD
   Conformance: yes
********** ********** ********** ********** **********/
private pred showIdentifiedTotallyIDedDD(
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, Identified] and
         some p->TotallyIDed &
               njh.projectDataTransformRequired and
            some q->DownloadDisabled & njh.VDAllowed and
               not totallyIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   AT: IDED
   Transform: AllowDeIDed
   Query Status: DD
   Conformance: yes
********** ********** ********** ********** **********/
```

```
private pred showIdentifiedAllowDeIDedDD (njh: NJH) {
   setPartialInstanceConfig[njh] and
      inv[njh] and
   some
      p: njh.projects |
   p in njh.projects and
   p->Identified in njh.projectAT and
      some q: Query |
         some p->q & njh.projectQueries and
            some q.(njh.qryReturns)  and
               some q->DownloadDisabled & njh.VDAllowed and
                  some p->AllowDeIDed &
                        njh.projectDataTransformRequired and
                     allowDeIDedTransform[njh, q]}

/********** ********** ********** ********** **********
   AT: DeIDED
   Transform: wel formed instances imply it
   Query Status: DA
   Conformance: yes
********** ********** ********** ********** **********/
private pred showDeIDedDA(
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, DeIDed] and
         some q->DownloadAllowed & njh.VDAllowed and
            totallyDeIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   AT: IDED
   Transform: TotallyIDed
   Query Status: DA
   Conformance: yes
********** ********** ********** ********** **********/
private pred showIdentifiedTotallyIDedDA(
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, Identified] and
         some p->TotallyIDed &
               njh.projectDataTransformRequired and
            some q->DownloadAllowed & njh.VDAllowed and
               totallyIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   AT: IDED
   Transform: AllowDeIDed
   Query Status: DA
   Conformance: yes
********** ********** ********** ********** **********/
private pred showIdentifiedAllowDeIDedDA (
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, Identified] and
         some p->AllowDeIDed & njh.projectDataTransformRequired and
            some q->DownloadAllowed & njh.VDAllowed and
               allowDeIDedTransform[njh, q]}

/********** ********** ********** ********** **********
   AT: DeIDED
   Transform: wel formed instances imply it
   Query Status: DA
   Conformance: no
```

```
********** ********** ********** ********** **********/
private pred showDeIDedNCDA (
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, DeIDed] and
         some q->DownloadAllowed & njh.VDAllowed and
            not totallyDeIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   AT: DeIDED
   Transform: wel formed instances imply it
   Query Status: DA
   Conformance: no
********** ********** ********** ********** **********/
private pred showIdentifiedNCTotallyIDedDA (
   njh: NJH, p: Project, q: Query) {
   setPartialInstanceConfig[njh] and
      common_inst[njh, p, q, Identified] and
         some p.(njh.projectDataTransformRequired) & TotallyIDed and
            some q->DownloadAllowed & njh.VDAllowed and
               not totallyIDedTransform[njh, q] }

/********** ********** ********** ********** **********
   Give me any instance of the system
********** ********** ********** ********** **********/
private pred show (njh: NJH) {}

/********** ********** ********** ********** **********
   Give me an instance of the system where a
   query has no VDAllowed
********** ********** ********** ********** **********/
pred showg(njh: NJH, p: Project, q: Query) {
   some p & (njh.projects) and
      some p->q & njh.projectQueries and
         some p.(njh.projectAT) and
            no q.(njh.VDAllowed) and
               some q.(njh.qryReturns) }

/*
run show for 7but 1NJH expect 1
run showg for 7expect 1/*
run common_inst for 7expect 1*/

run showDeIDedDD for 7but 1NJH expect 1
run showDeIDedDA for 7but 1NJH expect 1
run showDeIDedNCDA for 7but 1NJH expect 0

run showIdentifiedTotallyIDedDD for 7but 1NJH expect 1
run showIdentifiedTotallyIDedDA for 7but 1NJH expect 1
run showIdentifiedNCTotallyIDedDA for 7but 1NJH expect 0

run showIdentifiedAllowDeIDedDD for 7but 1NJH expect 0
run showIdentifiedAllowDeIDedDA for 7but 1NJH expect 1

/********** ********** ********** ********** ********** ********** **********
   End of Predicates/Assertions for other MODEL Instances
********** ********** ********** ********** ********** ********** **********/


/********** ********** ********** ********** ********** ********** **********
   HIPAA Conformance Checks
   Asserts MODEL Instances well formed for VD Allowed
```

```
********** ********** ********** ********** ********** ********** **********/
private pred conform_overlap (njh: NJH, q: Query, at: AccessTicket ) {
   someOfAllRelationsSatisfyingInvAndConfig[njh] and
      some (njh.projectQueries).q.(njh.projectAT) & at }

pred conformanceQryIdentifiedAllowed (
   njh: NJH, p: Project , q: Query) {
   some p.(njh.projectDataTransformRequired) & TotallyIDed implies
      all
         r: applicableDates[njh, q] |
      identifiedDate[r] }

pred conformanceQryIdentifiedDisabled (njh: NJH, p: Project , q: Query) {
   (some p.(njh.projectDataTransformRequired) & TotallyIDed implies
      some
         r: applicableDates[njh, q] |
      not identifiedDate[r] ) }

pred conformanceQryDeIDedAllowed (njh: NJH, p: Project , q: Query) {
   all
      r: applicableDates[njh, q] |
   not identifiedDate[r] }

pred conformanceQryDeIDedDisabled (njh: NJH, p: Project , q: Query) {
   some
      r: applicableDates[njh, q] |
   identifiedDate[r] }

/** fault in the invDownloadAllowedAllowIDed predicate allows a
   counterexample here, i.e.,
   conformanceQryIdentifiedAllowed fails */
private pred HIPAADateNonConformanceIdentified
   (njh: NJH, p: Project, q: Query) {
   p = (njh.projectQueries).q and
      conform_overlap[njh, q, Identified] and
         some p.(njh.projectDataTransformRequired) & TotallyIDed and
            some q.(njh.VDAllowed) & DownloadAllowed and
               not conformanceQryIdentifiedAllowed[njh, p, q] }
run HIPAADateNonConformanceIdentified for 7expect 0

/** fault in the invDownloadAllowedAllowIDed predicate allows a
   counterexample here, i.e.,
   conformanceQryIdentifiedAllowed fails */
assert HIPAADateConformanceIdentified {
   all
      njh: NJH,
      q: njh.queries |
   let
      p = (njh.projectQueries).q | {

   (conform_overlap[njh, q, Identified] and
      some q.(njh.VDAllowed) & DownloadAllowed) implies
         conformanceQryIdentifiedAllowed[njh, p, q]

   (conform_overlap[njh, q, Identified] and
      some q.(njh.VDAllowed) & DownloadDisabled) implies
         conformanceQryIdentifiedDisabled[njh, p, q] }}
check HIPAADateConformanceIdentified for 7expect 0

assert HIPAADateConformanceDeIDed {
   all
      njh: NJH,
```

```
      q: njh.queries |
    let
      p = (njh.projectQueries).q | {

    (conform_overlap[njh, q, DeIDed] and
      some q.(njh.VDAllowed) & DownloadAllowed) implies
        conformanceQryDeIDedAllowed[njh, p, q]

    (conform_overlap[njh, q, DeIDed] and
      some q.(njh.VDAllowed) & DownloadDisabled) implies
        conformanceQryDeIDedDisabled[njh, p, q] }}
check HIPAADateConformanceDeIDed for 7expect 0

/********** ********** ********** ********** ********** ********** **********
   End  HIPAA Conformance Checks
********** ********** ********** ********** ********** ********** **********/
```

D.2 Updated USE Class Model Specifications and Constraints for Slice 3 to *ApproveAccessTicket* Operation

Listing D.3: *USE Class Model for Slice 3 to Approve Access Ticket*

```
/*
 Model slice for NJH to
3. approve project licence,

Written by Phillipa Bennett
Date August 18, 2016
Version 4
 */

model NJHg_slice_1

/* Abstract CLASSES */
abstract class DataSource end
abstract class DataTransform end
abstract class Permission end
abstract class Rule
attributes
operations
    applyRule()
end
abstract class Purpose end

/* Extended abstract classes */
abstract class AccessTicket < Permission end

class TotallyDeIDed < DataTransform end
class TotallyIDed < DataTransform end
class AllowDeIDed < DataTransform end

abstract class Licence < Permission end
abstract class DecisionRule < Rule end

/* Unextended concrete classes */
class Personnel end
class Query
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end


/*  Extended concrete classes */
class Project < DataSource end
class ClinicalDB < DataSource end


class Fishing < Licence end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class CanUseTotallyDeIDed < DecisionRule end
```

```
class ClinicalDBNeedsDataCollector < DecisionRule end
class DataAccessAgreementPresent < DecisionRule end
class DataSourcePriorityOK < DecisionRule end
class LicenedTeamAndPI < DecisionRule end
class NoOverlapPITeamDC < DecisionRule end
class NoSupsInPIandDC < DecisionRule end
class PIDefined < DecisionRule end
class ProjectMembersDefined < DecisionRule end
class QualifierPresent < DecisionRule end
class SomePurposeNotDirectTreatment < DecisionRule end
class SomeQueriesDefined < DecisionRule end
class SomeSourcesDefined < DecisionRule end

class DirectTreatment < Purpose end
class Research < Purpose end

/* These classes are defined using the 'in' keyword in the Alloy model.
   How will we achieve this in OCL? */
class Qualifier < Personnel
attributes
operations
   QualifyResearcher(res: Researcher)
end
class Researcher < Personnel end


/* ASSOCIATIONS */

association ATPriority between
   AccessTicket[*] role ant
   AccessTicket[*] role desc
end

association DataAccessAgreement between
   Project[*] role owner
   Project[*] role user
end

association PermRules between
   Permission[*]
   Rule[1..*]
end

association ProjectAT between
   Project[*]
   AccessTicket[0..1]
end

association ProjectDataCollector between
   Project[*]
   Personnel[0..1] role dc
end

association ProjectDataTransformRequired between
   Project[*]
   DataTransform[0..1]
end

association ProjectMembers between
   Project[*] role proj
   Researcher[*] role members
end
```

```

association ProjectPI between
    Project[*] role pi_proj
    Researcher[0..1] role pi
end

association ProjectPurpose between
    Project[*]
    Purpose[0..1]
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
end

association ResearcherL between
    Researcher[*]
    Licence[0..1]
end

association Supervisors between
    Personnel[*] role supervisor
    Personnel[*] role supervised
end
```

Listing D.4: *USE Class Model for Slice 5 to Check Conformance*

```
/*
 Model slice for NJH to
4. execute query

Written by Phillipa Bennett
Date Sept 20, 2016
Version 4
 */

model NJHg_slice_5

/* Abstract CLASSES */

abstract class Data end
abstract class Permission end
abstract class DataTransform end

/* Extended abstract classes */
abstract class AccessTicket < Permission end
class TotallyDeIDed < DataTransform end
class TotallyIDed < DataTransform end
class AllowDeIDed < DataTransform end

/* Unextended concrete classes */
class DataItem
attributes
    name: String
```

```
end

class Query
attributes
operations
    download()
    view()
end

abstract class Status end

/*  Extended concrete classes */

class Date < Data
attributes
    day: Integer
    month: Integer
    year: Integer
operations
    isIdentified(): Boolean
    isNotIdentified(): Boolean
end

class DStr < Data
attributes
    sVal: String
end

class Project end

class QryData < DataItem end
class RetData < DataItem end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class DownloadDisabled < Status end
class DownloadAllowed < Status end

/* ASSOCIATIONS */
association DataValues between
    DataItem[*]
    Data[1]
end

association EnteredOn between
    DataItem[*] role item
    Date[0..1] role date
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectDataTransformRequired between
    Project[*]
    DataTransform[0..1]
end

association ProjectQueries between
```

```
    Project[*] /* relax from 1to * to allow generation program to work, enforced as 1in a
        constraint */
    Query[*]
end

association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association VDAllowed between
    Query[*]
    Status[0..1]
end
```

Listing D.5: *Additional USE Constraints applicable only to Slices 3 and 5 to Approve Access Ticket and Check Conformance respectively*

```
context DataTransform
inv singletonEachDT:
    DataTransform.allInstances.select(
        oclIsTypeOf(TotallyDeIDed)=true)->size()<=1
    and
    DataTransform.allInstances.select(
        oclIsTypeOf(TotallyIDed)=true)->size()<=1
    and
    DataTransform.allInstances.select(
        oclIsTypeOf(AllowDeIDed)=true)->size()<=1

context Project
inv invProjectATDataTransform1:
    projectAT.select(
        oclIsTypeOf(Identified)=true)->size()=1 implies
    dataTransform.select(oclIsTypeOf(TotallyDeIDed)=true)->size()=0

inv invProjectATDataTransform2:
    projectAT.select(
        oclIsTypeOf(DeIDed)=true)->size()=1 implies (
    dataTransform.select(oclIsTypeOf(AllowDeIDed)=true)->size()=0 and
    dataTransform.select(oclIsTypeOf(TotallyIDed)=true)->size()=0)
```

E.1 Updated USE Class Model Specifications and Constraints for Slice 3 to *ApproveAccessTicket* Operation

Listing E.1: *USE Class Model for Slice 3 to Approve Access Ticket*

```
/*
 Model slice for NJH to
3. Approve project licence when rules for Children Protected Populations
    are to be considered

Written by Phillipa Bennett
Date December 20, 2016
Version 5

 Updated Dec 28, 2016
    with additional requirements
    for IRB to specify if consent/assent required

 */

model NJHg_slice_1

/* Abstract CLASSES */
abstract class Consent end
abstract class ConsentRequirement end
abstract class DataSource end
abstract class DataTransform end
abstract class Permission end
abstract class PersonRole end
abstract class ResearchRisk end
abstract class Rule
attributes
operations
    applyRule()
end
abstract class Purpose end

/* Extended abstract classes */
abstract class AccessTicket < Permission end

class ResponsiblityRole < PersonRole end
abstract class SpecialSubject < PersonRole end

abstract class Licence < Permission end

abstract class ChildrenResearchRisk < ResearchRisk end

abstract class DecisionRule < Rule end

/* Unextended concrete classes */
class IRB end
class Person end
class Personnel < Person end
class Query
```

```
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end

/* Extended concrete classes */
class DeIDed < AccessTicket end
class Identified < AccessTicket end

class RiskNotAllowed < ChildrenResearchRisk end
class MinimalRisk < ChildrenResearchRisk end
class DirectBenefit < ChildrenResearchRisk end
class DirectBenefitGeneralisable < ChildrenResearchRisk end
class FurtherUnderstandingPreventionAlleviation < ChildrenResearchRisk end

class Allow < Consent end
class DisAllow < Consent end

class Required < ConsentRequirement end
class NotRequired < ConsentRequirement end

class Project < DataSource end
class ClinicalDB < DataSource end

class TotallyDeIDed < DataTransform end
class TotallyIDed < DataTransform end
class AllowDeIDed < DataTransform end

class CanUseTotallyDeIDed < DecisionRule end
class ClinicalDBNeedsDataCollector < DecisionRule end
class DataAccessAgreementPresent < DecisionRule end
class DataSourcePriorityOK < DecisionRule end
class LicenedTeamAndPI < DecisionRule end
class NoOverlapPITeamDCIRB < DecisionRule end
class NoSupsInPIandDC < DecisionRule end
class PIDefined < DecisionRule end
class ProjectMembersDefined < DecisionRule end
class QualifierPresent < DecisionRule end
class SomePurposeNotDirectTreatment < DecisionRule end
class SomeQueriesDefined < DecisionRule end
class SomeSourcesDefined < DecisionRule end
class SpecialResearchApproved < DecisionRule end

class Fishing < Licence end

class DirectTreatment < Purpose end
class Research < Purpose end

class Researcher < Personnel end

class Parent < ResponsiblityRole end
class Guardian < ResponsiblityRole end
class WardOfState < ResponsiblityRole end

class Children < SpecialSubject end

/* ASSOCIATIONS */

association ATPriority between
    AccessTicket[*] role ant
```

```
    AccessTicket[*] role desc
end

association DataAccessAgreement between
    Project[*] role owner
    Project[*] role user
end

association IRBMembers between
    IRB[0..1] role irb
    Personnel[2..*]
end

association PermRules between
    Permission[*]
    Rule[1..*]
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectConsentAssssentReq between
    Project[*]
    PersonRole[*]
    ConsentRequirement[0..1]
end

association ProjectDataCollector between
    Project[*]
    Personnel[0..1] role dc
end

association ProjectDataTransformRequired between
    Project[*]
    DataTransform[0..1]
end

association ProjectMembers between
    Project[*] role proj
    Researcher[*] role members
end

association ProjectPI between
    Project[*] role pi_proj
    Researcher[0..1] role pi
end

association ProjectPurpose between
    Project[*]
    Purpose[0..1]
end

association ProjectQueries between
    Project[*] /* relax from 1, to * to allow generation program to work */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
```

```
end

association ProjectSpecialResearch between
    Project[*] role ssSubject
    SpecialSubject[*]
end

association ProjectSpecialResearchApproval between
    Project[*] role spProject
    SpecialSubject[*] role spSubject
    ResearchRisk[0..1]
    IRB[0..1] role irb
    Consent[0..1]
end

association ResearcherL between
    Researcher[*]
    Licence[0..1]
end

association Supervisors between
    Personnel[*] role supervisor
    Personnel[*] role supervised
end
```

Listing E.2: *USE Class Model for Slice 4 to Execute Query*

```
/*
 Model slice for NJH to
4. execute query with Protected Children

Written by Phillipa Bennett
Date December 20, 2016
Version 5

 Updated Dec 28, 2016
 with changed and additional requirements
    1. advocate can be IRB member; and
    2. advocate cannot be associated with guardian organisation
 */

model NJHgv_pc_slice_4

/* Abstract CLASSES */
abstract class Category end
abstract class Consent end
abstract class ConsentRequirement end
abstract class Data end
abstract class DataSource end
abstract class DataTransform end
abstract class Permission end
abstract class PersonRole end
abstract class Rule
attributes
operations
    applyRule()
end

/* Extended abstract classes */
abstract class HIPAACat < Category end

class TotallyDeIDed < DataTransform end
class TotallyIDed < DataTransform end
class AllowDeIDed < DataTransform end

abstract class SpecialPopn < HIPAACat end

abstract class AccessTicket < Permission end

class ResponsiblityRole < PersonRole end
abstract class SpecialSubject < PersonRole end

abstract class AccessRule < Rule end

abstract class Type end

/* Unextended concrete classes */
class DataItem
attributes
    name: String
end

class IRB end
class Person end

/*  Extended concrete classes */
```

```
class ChildAdvocateForWardOfState < AccessRule end
class ChildAssentAndResponsibilityConsent < AccessRule end
class HideSpecialPopn < AccessRule end
class ChildAdvocateNotAssocWithResearchOrWardOrg < AccessRule end
class PatientConsent < AccessRule end
class TransformHDate < AccessRule end

class DeIDed < AccessTicket end
class Identified < AccessTicket end

class Allow < Consent end
class CannotGive < Consent end
class DisAllow < Consent end

class Required < ConsentRequirement end
class NotRequired < ConsentRequirement end

class Date < Data
attributes
    day: Integer
    month: Integer
    year: Integer
operations
    isIdentified(): Boolean
    isNotIdentified(): Boolean
end

class HDate < HIPAACat end
class HIPAAChild < SpecialPopn end

class Project < DataSource end
class ClinicalDB < DataSource end

class Researcher < Personnel end

class QryData < DataItem end
class RetData < DataItem end

class Patient < Person end
class Personnel < Person end

class Query
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end

class Parent < ResponsiblityRole end
class Guardian < ResponsiblityRole end
class WardOrg < ResponsiblityRole end

class Children < SpecialSubject end

class Individual < Type end
class Group < Type end

/* ASSOCIATIONS */
association ARAppliesTo between
    AccessRule[*] role accessrule
    Type[1..*] role type
```

```
end

association ARHides between
    AccessRule[*]
    Category[*]
end

association ARTransforms between
    AccessRule[*] role hAccessRules
    HIPAACat[*]
end

association ChildAdvocate between
    Patient[*] role advocatePt
    Person [0..1] role ptAdvocate
end

association ChildParticipationAssent between
    Patient[*] role spPatient
    Consent[*] role spPatientAssent
end

association ChildParticipationPerm between
    ResponsiblityRole[*]
    Person[*] role spPWPerson
    Patient[*] role spPWPatient
    Consent[0..1] role spPatientPerm
end

association DataValues between
    DataItem[*]
    Data[1]
end

association DICat between
    DataItem[*]
    HIPAACat[*]
end

association DISource between
    DataSource[0..1]
    DataItem[*]
end

association EnteredOn between
    DataItem[*] role item
    Date[0..1] role date
end

association IRBMembers between
    IRB[0..1] role irb
    Personnel[1..*]
end

association PatientData between
    Patient[0..1]
    DataItem[*]
    Consent[0..1]
end

association PermRules between
    Permission[*]
```

```
    Rule[1..*]
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectConsentAssssentReq between
    Project[*]
    PersonRole[*]
    ConsentRequirement[0..1]
end

association ProjectDataCollector between
    Project[*]
    Personnel[0..1] role dc
end

association ProjectDataTransformRequired between
    Project[*]
    DataTransform[0..1]
end

association ProjectMembers between
    Project[*] role proj
    Researcher[*] role members
end

association ProjectPI between
    Project[*] role pi_proj
    Researcher[0..1] role pi
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation program to work, enforced as 1in a
        constraint */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
end

association ProjectSpecialResearch between
    Project[*] role ssProject
    SpecialSubject[*]
end

association QryWorksOn between
    Query[*]
    QryData[*]
end

association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association RDType between
```

```
    Query[*] role rd_qry
    RetData[*] role rd_data
    Type[0..1]
end

association SpecialPatient between
    Patient [*]
    SpecialPopn[*]
end

association WardAssociates between
    WardOrg [*]
    Person[1..*]
end
```

Listing E.3: *Full USE Class Model for the NJH sysyem*

```
/*
 NJH Full

Written by Phillipa Bennett
 Updated January 26, 2017
Version 5
 */

model NJHgv_pc_full

/* Abstract CLASSES */
abstract class Category end
abstract class Consent end
abstract class ConsentRequirement end
abstract class Data end
abstract class DataSource end
abstract class DataTransform end
abstract class Permission end
abstract class PersonRole end
abstract class Purpose end
abstract class ResearchRisk end
abstract class Rule
attributes
operations
    applyRule()
end
abstract class Status end

/* Extended abstract classes */
abstract class HIPAACat < Category end

class TotallyDeIDed < DataTransform end
class TotallyIDed < DataTransform end
class AllowDeIDed < DataTransform end

abstract class SpecialPopn < HIPAACat end

abstract class AccessTicket < Permission end
abstract class Licence < Permission end



class ResponsiblityRole < PersonRole end
abstract class SpecialSubject < PersonRole end

abstract class ChildrenResearchRisk < ResearchRisk end

abstract class AccessRule < Rule end
abstract class DecisionRule < Rule end

abstract class Type end

/*  Extended concrete classes */
class ChildAdvocateForWardOfState < AccessRule end
class ChildAssentAndResponsibilityConsent < AccessRule end
class HideSpecialPopn < AccessRule end
class ChildAdvocateNotAssocWithResearchOrWardOrg < AccessRule end
class PatientConsent < AccessRule end
class TransformHDate < AccessRule end
```

```
class DeIDed < AccessTicket end
class Identified < AccessTicket end

class RiskNotAllowed < ChildrenResearchRisk end
class MinimalRisk < ChildrenResearchRisk end
class DirectBenefit < ChildrenResearchRisk end
class DirectBenefitGeneralisable < ChildrenResearchRisk end
class FurtherUnderstandingPreventionAlleviation < ChildrenResearchRisk end

class Allow < Consent end
class CannotGive < Consent end
class DisAllow < Consent end

class Required < ConsentRequirement end
class NotRequired < ConsentRequirement end

class Date < Data
attributes
    day: Integer
    month: Integer
    year: Integer
operations
    isIdentified(): Boolean
    isNotIdentified(): Boolean
end

class HDate < HIPAACat end
class HIPAAChild < SpecialPopn end

class QryData < DataItem end
class RetData < DataItem end

class Project < DataSource end
class ClinicalDB < DataSource end

class CanUseTotallyDeIDed < DecisionRule end
class ClinicalDBNeedsDataCollector < DecisionRule end
class DataAccessAgreementPresent < DecisionRule end
class DataSourcePriorityOK < DecisionRule end
class LicenedTeamAndPI < DecisionRule end
class NoOverlapPITeamDCIRB < DecisionRule end
class NoSupsInPIandDC < DecisionRule end
class PIDefined < DecisionRule end
class ProjectMembersDefined < DecisionRule end
class QualifierPresent < DecisionRule end
class SomePurposeNotDirectTreatment < DecisionRule end
class SomeQueriesDefined < DecisionRule end
class SomeSourcesDefined < DecisionRule end
class SpecialResearchApproved < DecisionRule end

class Fishing < Licence end

class Patient < Person end
class Personnel < Person end
class Researcher < Personnel end
class Qualifier < Personnel
attributes
operations
    QualifyResearcher(res: Researcher)
end
```

```
class DirectTreatment < Purpose end
class Research < Purpose end

class DownloadDisabled < Status end
class DownloadAllowed < Status end

class Parent < ResponsiblityRole end
class Guardian < ResponsiblityRole end
class WardOrg < ResponsiblityRole end

class Children < SpecialSubject end

class Individual < Type end
class Group < Type end

/* Unextended concrete classes */
class DataItem
attributes
    name: String
end

class IRB end
class Person end

class Query
attributes
operations
    runQuery(res: Researcher, proj: Project)
    download()
    view()
end


/* ASSOCIATIONS */
association ARAppliesTo between
    AccessRule[*] role accessrule
    Type[1..*] role type
end

association ARHides between
    AccessRule[*]
    Category[*]
end

association ARTransforms between
    AccessRule[*] role hAccessRules
    HIPAACat[*]
end

association ATPriority between
    AccessTicket[*] role ant
    AccessTicket[*] role desc
end

association ChildAdvocate between
    Patient[*] role advocatePt
    Person [0..1] role ptAdvocate
end

association ChildParticipationAssent between
    Patient[*] role spPatient
    Consent[*] role spPatientAssent
```

```
end

association ChildParticipationPerm between
    ResponsiblityRole[*]
    Person[*] role spPWPerson
    Patient[*] role spPWPatient
    Consent[0..1] role spPatientPerm
end

association DataAccessAgreement between
    Project[*] role owner
    Project[*] role user
end

association DataValues between
    DataItem[*]
    Data[1]
end

association DICat between
    DataItem[*]
    HIPAACat[*]
end

association DISource between
    DataSource[0..1]
    DataItem[*]
end

association EnteredOn between
    DataItem[*] role item
    Date[0..1] role date
end

association IRBMembers between
    IRB[0..1] role irb
    Personnel[1..*]
end

association PatientData between
    Patient[0..1]
    DataItem[*]
    Consent[0..1]
end

association PermRules between
    Permission[*]
    Rule[1..*]
end

association ProjectAT between
    Project[*]
    AccessTicket[0..1]
end

association ProjectConsentAssentReq between
    Project[*]
    PersonRole[*]
    ConsentRequirement[0..1]
end

association ProjectDataCollector between
```

```
    Project[*]
    Personnel[0..1] role dc
end

association ProjectDataTransformRequired between
    Project[*]
    DataTransform[0..1]
end

association ProjectMembers between
    Project[*] role proj
    Researcher[*] role members
end

association ProjectPI between
    Project[*] role pi_proj
    Researcher[0..1] role pi
end

association ProjectPurpose between
    Project[*]
    Purpose[0..1]
end

association ProjectQueries between
    Project[*] /* relax from 1to * to allow generation program to work, enforced as 1in a
        constraint */
    Query[*]
end

association ProjectSources between
    Project [*]
    DataSource[*]
end

association ProjectSpecialResearch between
    Project[*] role ssProject
    SpecialSubject[*]
end

association ProjectSpecialResearchApproval between
    Project[*] role spProject
    SpecialSubject[*] role spSubject
    ResearchRisk[0..1]
    IRB[0..1] role irb
    Consent[0..1]
end

association QryWorksOn between
    Query[*]
    QryData[*]
end

association QryReturns between
    Query[*] role qry
    RetData[*] role rData
    QryData[*] role qData
end

association RDType between
    Query[*] role rd_qry
    RetData[*] role rd_data
```

```
    Type[0..1]
end

association ResearcherL between
    Researcher[*]
    Licence[0..1]
end

association ResearcherQualifier between
    Researcher[*]
    Qualifier[0..1]
end

association SpecialPatient between
    Patient [*]
    SpecialPopn[*]
end

association Supervisors between
    Personnel[*] role supervisor
    Personnel[*] role supervised
end

association VDAllowed between
    Query[*]
    Status[0..1]
end

association WardAssociates between
    WardOrg [*]
    Person[1..*]
end
```