Integration of Structured Review and Model-based Verification: a Case Study

DART /1001/04

Demissie B. Aredo
Issa Traore
M. Liu Yanguo
Hong Ye

Tittel/Title: **Integration of Structured Review and Model-based Verification: a Case Study**

Dato/Date: August
År/Year: 2004
ISBN: 82-539-0509-2
Publikasjonsnr.:DART/1001/04
Publication no.:

Forfatter/Authors: Demissie B. Aredo,
Issa Traore,
M. Liu Yanguo,
Hong Ye

Sammendrag/Abstract: *In this report, we discuss how structured reviews and formal verification and validation (V&V) can be integrated into a single development framework to exploit the synergy between them. The integrated approach uses graphical modeling techniques and the supporting tools as a front-end to formal V&V in order to improve feasibility of the framework. This in turns increases acceptability of formal V&V techniques among software developers by hiding their esoteric features behind the graphical modeling techniques, which are popular among the software developers.*

Emneord/Keywords: Structured reviews, Formal methods, Verification & Validation (V&V), Model-based Verification

Tilgjengelighet/Availability: Public

Prosjektnr./Project no.:

Satsningsfelt/Research field: Software Engineering

Antall sider/No. of pages: 36

Integration of Structured Review and Model-based Verification: a Case Study

Demissie B. Aredo[1], Issa Traore[2], M. Liu Yanguo[2] and Hong Ye[2]

[1]Norwegian Computing Center
N-0314 Oslo, Norway
E-mail: demissie.aredo@nr.no

[2]Dept of Electrical and Computer Engineering, University of Victoria,
Victoria, BC, V8W 3P6, Canada
Email: itraore,yliu,hye@ece.uvic.ca

Abstract: *In this report, we discuss how structured reviews and formal verification and validation (V&V) can be integrated into a single development framework to exploit the synergy between them. The integrated approach uses graphical modeling techniques and the supporting tools as a front-end to formal V&V in order to improve feasibility of the framework. This in turns increases acceptability of formal V&V techniques among software developers by hiding their esoteric features behind the graphical modeling techniques, which are popular among the software developers.*

Keywords: Structured reviews, Formal methods, Verification & Validation (V&V), Model-based Verification

1 Introduction

Software systems are increasingly becoming pervasive in several sectors of the contemporary society. E-banking, e-commerce, aircraft control, mission-critical satellite launchers are some examples of applications where high level of dependability is a crucial requirement. Yet, lower production cost and shorter time-to-market, at the expense of software quality, have become the driving forces for most software development organizations. Fortunately, that is not yet the case in organizations developing critical software systems, e.g. nuclear reactor control, avionics, and mission control systems where a failure may cause loss of human lives, or economic disaster.

Structured reviews and formal verification and validation (V&V) techniques are among the principal methods contributing to the improvement of quality of software products and processes. These techniques have inherent strengths and limitations, e.g. with respect to cost and coverage. An approach that exploits the synergy between their strengths improves the reliability of software products significantly.
V&V is a software analysis process, which encompasses requirement and design reviews, code inspection, and testing. In general, reviews are conducted manually and they are efficient at checking a limited number of correctness arguments such as completeness, robustness, and optimality.
The level of quality achieved with informal review techniques may not be sufficient for critical systems, where high level of dependability and reliability is crucial. An alternative is to integrate formal methods (FM) into V&V process. FMs are centered on mathematical theories allowing precise specification of system requirements, and rigorous analysis to ensure that a product meets the expectations of users, in functionality as well as quality. Some benefits of introducing FMs into a development process include:

- Improves our understanding of requirements and system design, and reduces errors and omissions;
- A possibility to mechanically check consistency and completeness of a specification, and prove that the implementation conforms to the specification;
- Semantically based CASE tools for automation of analysis, design, implementation and debugging, and animation of specifications in developing prototypes.
- Formal specifications can be used as a guide for generating appropriate test cases.

Despite these benefits, FMs still have difficulty in breaking into the software industry. Very few organizations have introduced FMs into their development process. A number of reasons have been put forward as to why the formal development methods have not been widely used in the software industry [1]:

- FMs are esoteric – software engineers have not been trained in the discrete mathematics and logic at the required level. Moreover, customers are not familiar with formal development methods, and hence are not willing to pay for development activities they do not monitor.

- Lack of tool support – most of the research work on FMs focus on the development of languages and their theoretical underpinning, yet a little effort is devoted to their practical feasibility, e.g. tool support.

Several approaches have been proposed [2, 3] to integrate FMs into software development processes. They advocate a lightweight and selective application of FMs using modeling languages such as the UML [4] as a front-end. Models are created using graphical notations familiar to the developers. Inspired by this approach, we propose a framework that integrates structured reviews and formal V&V methods into a single development framework to exploit the synergy between them. V&V steps that are not fully automated are reviewed manually, whereas mechanized verification complements structured reviews in detecting inconsistencies and omissions. The later allows reviewers to focus on aspects that cannot be automated.

The rest of the paper is organized as follows. In Section 2, an overview of the methodology underpinning our approach is presented. In Section 3, major concepts of structured reviews are summarized. In Section 4, a tool for automating our framework is briefly discussed. In Section 5, feasibility of the proposed framework and the tool is illustrated. Finally, in Section 6 we conclude.

2 Model-based Verifications

In this section, we briefly discuss major aspects of our approach to model-based verification. For a more detailed discussion, readers are referred to [5].

2.1 The Unified Modeling Language

The Unified Modeling Language (UML) [4] is among the most popular modeling language currently used in the software industry. The visual notations, which can easily be learned by system developers and the availability of several industrial-strength CASE tools, are among the factors that contributed to its popularity. UML is an industry standard for OO modeling languages and enhances communication between different stakeholders. However, due to the lack of formal semantics for UML notations, V&V techniques may not be applied directly to UML models. To bridge the gap, we proposed formal semantics for the UML notations [6, 7] in the PVS specification language. The proposed semantics is implemented in the PrUDE (Precise UML Development Environment) tool [8]. The PrUDE tool supports automatic transformation of UML models into PVS specifications, which are manipulated at the back-end using the PVS toolkit.

2.2 A Semantic Domain

The Prototype Verification System (PVS) [9] is based on strongly typed higher-order logic with powerful mechanisms for verification and validation. PVS consists of a highly expressive specification language (SL) tightly integrated with a type-checker, and an interactive general-purpose theorem-prover. The PVS-SL provides a very general

semantic foundation. A particular strength of PVS is that it exploits the synergy between its tools. It is beyond the scope of this paper to give a detailed presentation of PVS. Interested readers are referred to [9].

2.3 Semantics of UML in PVS

The lack of formal semantics for UML notations hampers application of formal verifications to UML models. There is a great deal of work on providing mathematical basis for the concepts underlying the UML notations. Several approaches are proposed [10]: supplemental – informal OO modeling constructs are replaced with more formal constructs; OO-extension – a novel or an existing formal notation is extended with OO features to make it compatible with OO features; method integration - an informal or a semi-formal notation is combined with a suitable formalism to make it precise and amenable to rigorous analysis.

The first two approaches require developers to deal with a certain amount of formal artifacts - a major barrier for whole-scale utilization of formal methods in industrial settings - and suffer from lack of supporting tools. Method integration is a widely used approach that allows developers to manipulate the graphical models they have created without having in-depth knowledge of the underlying formalism. We proposed semantics of a subset of UML notations (class, interaction, and statechart diagrams) [7, 6] using the method integration approach and the PVS specification language [11] as underlying semantic foundation. The informal semantics of UML notations [4] is used as a requirement document. Formal semantic definitions for UML notations facilitate a development of semantically based CASE tools for rigorous analysis.

We briefly summarize semantic model [6] for UML sequence diagrams. A UML sequence diagram describes a specific pattern of interaction between objects in terms of messages they exchange as the interaction unfolds over time to realize the desired property. The simplicity of sequence diagrams makes them suitable for requirement specifications that can easily be understood by customers, requirements engineers, and software developers alike. An interaction captured by a sequence diagram consists of messages communicated between interacting objects. A message has associated events specifying significant occurrences having location in time and space, and sender and receiver objects etc. In our framework, messages are interpreted as pair of send and receive events. A sequence diagram is interpreted as a prefix-closed set of traces of events having generic properties such as causality.
A semantic model for a sequence diagram captures properties that a system is expected to exhibit. Assumptions and invariants on the system are stated as axioms and predicates. A trace of events is a possible run of the system specified by the sequence diagram if and only if it satisfies properties stated as predicates provided that the assumption are satisfied. The static semantics of each model element given as a set of Well-formedness rules, usually expressed in the Object Constraint Language (OCL) [12], can be captured similarly.

2.4 PVS Proof Strategies

The ultimate goal of defining formal semantics is to precisely express important system properties and rigorously verify them. Using primitive proof rules provided in the PVS theorem-prover requires some expertise and is quite tedious to handle. PVS provides a mechanism for defining more powerful proof strategies that can significantly increase proof automation and hence reduce user interaction with the prover. This enables us to treat a complex proof in a single 'atomic' step, hiding the tedious intermediary steps from the user. We have identified and implemented some proof strategies that allow complete automation of proof of properties based on our semantic models [13]. For instance, for properties based on sequence diagrams, the proof pattern is quite simple and involves only two PVS primitive proof rules, namely skolem and grind. These strategies are implemented in the PrUDE tool and executed in a batch mode.

2.5 Model-based Testing

Program testing - checking whether or not a program exhibits behaviors stated in the requirement specification -- is an important step in development process.
Using formal specification as a basis of generating test cases contributes significantly to testing [14]. We present a testing approach based on validation of UML models using formal semantics and system requirements. The valid models are used to generate test cases from constraints such as invariants and pre-and post-conditions associated with model elements. Some UML models are more suitable for model-based testing. The statechart, sequence and class diagrams can provide a good testing coverage. To generate test cases from a sequence diagram, for instance, we use a trace-based testing strategy. After the sequence diagram is validated, graph matrices are built from the sequence diagram, and then reduced using the node-reduction algorithm [15], in order to generate test cases.

For statechart diagrams, we propose a transition test model consisting of a set of transitions associated to the diagrams. This allows generation of test cases at class and method levels. In UML state-charts, an event corresponds to a method call. Since a method may be invoked several times, a transition provides only partial pre- and post-conditions. The global pre- and post-condition is the conjunction of the partial pre- and post-conditions. Test cases are generated from a partial pre/post-condition pairs, by decomposing the precondition into disjunctive normal form (DNF), and yielding elementary sub-expressions. The sub-expressions are then refined into executable expressions, and then using the domain test matrix technique, suitable test cases are defined. The PrUDE tool provides a spreadsheet-like table that assists users in applying the domain test matrix technique.
For Java programs, the PrUDE tool provides a test execution component to which the generated test cases may be submitted and executed automatically. The tester, based on abstract expressions extracted from the specification by the PrUDE test component, provides executable expressions used to generate test cases. During the review process, a reviewer checks correctness of the executable expressions with respect to the abstract expressions, as well as the specification-based coverage criteria corresponding to the test strategies used. For more discussion on test expressions and coverage criteria please refer to [5].

3 Structured Design Reviews

3.1 Correctness Arguments

Most of the steps in the formal V&V can be carried out automatically. But, some steps cannot be automated and require human interaction and guidance. We argue that use of informal correctness arguments to deal with steps that cannot be automated results in an improved and more affordable verification process. Our approach draws on the work of Britcher [18], where the key program attributes such as topology, algebra, invariance, and robustness are defined for procedural programs. The correctness arguments are presented as a series of questions that should be answered by inspectors and authors. The idea of the questionnaire follows the Active Design Review approach developed by Parnas [19]. For instance, for a given correctness argument that cannot be checked automatically, a model analyst may provide and record an informal proof. During the review, the inspector is expected to challenge the correctness arguments using a carefully designed review process. In our case, we consider correctness arguments that encompass and extend the criteria defined in [18]: validity, traceability, optimality, robustness, well-formedness, completeness and consistency.

3.2 Sample Review Questions

A review process is preceded with a discovery of user requirements documented by the reviewer. Even before reading the exhibits, the reviewer needs to make an initial analysis of the requirements. The discovery of the requirements must go beyond the traditional meetings that take place at the beginning of reviews in order to present the system. The reviewer needs to build an informed and independent opinion about user requirements under review. Then, it is easier for the reviewer to challenge the rules defined in the exhibits and discover possible gaps, omissions or inconsistencies. In this phase, the reviewer should answer the following questions:

1. What are the main business rules, the properties and invariants characterizing the system?
2. What are significant scenarios underlying functionalities of the system?
3. What are exceptional conditions under which the system is expected to function?

After the discovery phase, the reviewer starts the actual review by reading the exhibits and examining the correctness arguments. The following are among the questions that need to be answered:

1. Do the exhibits provide a complete coverage of the business rules, the properties and invariants characterizing the system?
2. Are the exhibits consistent with user requirements, and do they derive naturally from the user requirements?

Next, the reviewer considers traceability argument. Some of the questions that may be answered include:

1. Which aspects of the model have changed, and which ones remain unchanged after refinement?

2. Are relationships between abstract and concrete features defined adequately and consistently?

As achieving traceability is not sufficient, checking optimality of the refinement is important. The argument of optimality may be analyzed by answering questions like:

1. Are representations chosen during design refinement efficient with respect to requirements?
2. Are there better alternative solutions?

Well-formedness arguments can be analyzed automatically using the PrUDE tool. The main goal of the reviewer is to identify potential syntactic inconsistencies. Consistency arguments are the broadest arguments among the correctness arguments defined so far. The goal of the reviewer is to check that there are no contradictory requirements involved in the models under review. The following are some of the questions that should be raised during robustness checking:

1. What are the normal conditions under which the system operates?
2. What are the exceptional and abnormal conditions related to the system operation?
3. Do the exhibits handle all exceptions and abnormal conditions?

The set of sample questions given in each step are not complete, and they are rather meant to illustrate types of questions that should be answered in each step.

3.3 Review Process

Defining an efficient review process requires selection of a rigorous development process, in which the steps and modeling artifacts are precisely specified. We use a development process consistent with the Rational Unified Process (RUP) [16] that is driven by use cases. Use cases are identified and prioritized by their degree of criticality at the beginning of the development process. The process proceeds iteratively starting with the most critical use case. At the end of iteration, stable software artifacts handling specific aspects and risks of the system are produced. Subsequent iterations are built on the previous ones by assessing and revising corresponding risks. The review activities can be performed at the major milestone within iteration and discovered errors should be fixed before the next iteration starts. Moreover, review comments are used in planning the next iteration. We use a hybrid unit of inspection that combines the traditional document-centric approach with the architectural approach proposed by Laitenberger et al. [17]. The key architectural building blocks, namely the use cases, are used as units of inspection, and within a use case we organize inspection around different documents.

As shown in Figure 1, major activities in the review process are organized into four phases (the rectangular boxes) each of which is based on a specific document: requirements, analysis, design, and test documents describing scenarios underlying the use case under consideration.

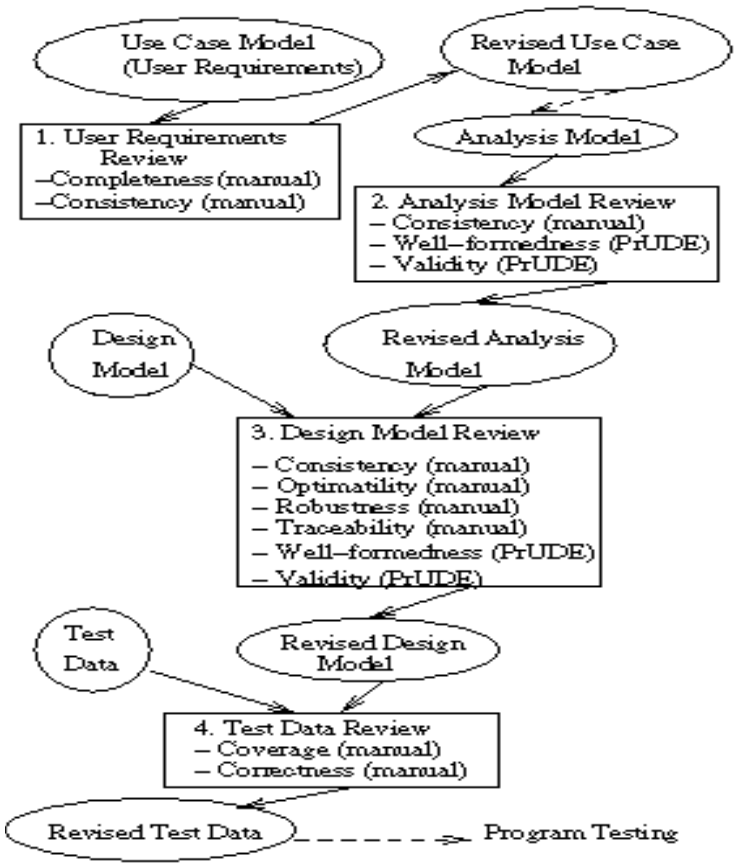

Figure 1: Review Process

User Requirements Review: Review activities in this phase include checking completeness and consistency arguments. Completeness refers to checking whether or not useful information is missing from a model by checking that every functional and quality requirement is covered at least by one use case. For every use case, the reviewer must check that every possible scenario is captured by a description of event flows. The reviewer manually checks consistency of the use case descriptions with user requirements.

Analysis Model Review: Analysis model is derived from textual descriptions of use cases. It consists of business rules; set of sequence/collaboration diagrams describing scenarios, class diagrams, and possibly state diagrams. In analysis model review, three correctness arguments are checked: consistency, well-formedness and validity. After consistency of the analysis model is manually checked and discovered defects are fixed, the model is input in to the PrUDE tool, where well-formedness and validity arguments are checked, successively. Well-formedness and validity are checked by generating PVS semantic models automatically. Then, the reviewer establishes these properties by discharging conjectures by invoking the PVS prover in a batch mode. Most of the conjectures can be discharged automatically using PVS proof strategies implemented in the PrUDE tool.

Design Model Review: Design models are obtained by successively refining analysis models. A design model consists of a class diagram, a set of interaction and statechart diagrams, a static structure diagram, a deployment diagram, and design traceability documentation. Review of design models consists of checking consistency, traceability, robustness and optimality arguments. Design traceability is documented by briefly describing the changes made to the analysis model to obtain the design model. The document may describe how design classes are related to analysis classes by defining retrieval functions, and if necessary, informal refinement proof. Design traceability documentation is produced by a designer, and challenged by a re-viewer.

Test Data Review: Artifacts submitted to a reviewer consist of test cases generated and expressions used to generate them. The role of the reviewer is to establish correctness of the expressions, by checking accuracy of system representation. The inspector needs to check that the coverage criteria for specification-based testing strategies used to generate the test cases are met.

4 Automation

The integrated approach presented in the sequel is automated by a tool suite called PrUDE[1] (Precise UML Development Environment) [20]. The formalisms and notations underlying the PrUDE platform are the UML [4] and the PVS (Prototype Verification System) [9] and their respective tools. Model-checking and proof-checking are based on the PVS toolkit that is invoked in batch mode, whereas models are created using a UML CASE tool. The interface of the PrUDE tool to a UML CASE tool is based on the XMI [21] format. Since most of UML CASE tools support model export in the XMI format, the PrUDE platform is UML tool vendor independent, making it easily adaptable to existing software development environments. A major strength of the PrUDE tool is that it allows developers to deal with UML models they have created while semantic models generated from the models are processed at the back-end. This is achieved by identifying proof strategies that allow automated verification of system properties based on the underlying semantic definitions. Figure 2 shows architecture of the PrUDE platform. The rectangular boxes represent V&V steps, whereas the ovals show artifacts. An input to the PrUDE tool is a requirement specification expressed in UML, and augmented with business rules expressed in OCL [12]. A corresponding PVS specification is generated automatically and serves as a basis of rigorous analysis. When a valid UML model is obtained after a series of V&V steps, a designer may refine the model to achieve an implementation of the system. The resulting program code can be tested with the PrUDE tool.

Test cases are generated from valid UML specifications obtained after the series of V&V steps. They are derived from various constraints related to the model, e.g. invariants, pre- and post-conditions. The current version of the PrUDE tool provides a test case generator and a test execution component for Java programs. If a proof attempt fails, a PVS log message that can be interpreted and traced back to the UML specification is generated. Although the log message is sufficient to indicate the source of errors in the UML

[1] The current version of PrUDE v1.2 can be downloaded from www.isot.ece.uvic.ca

specification, in the future we plan to implement a parser that extracts textual ”English-only” messages from PVS log messages.

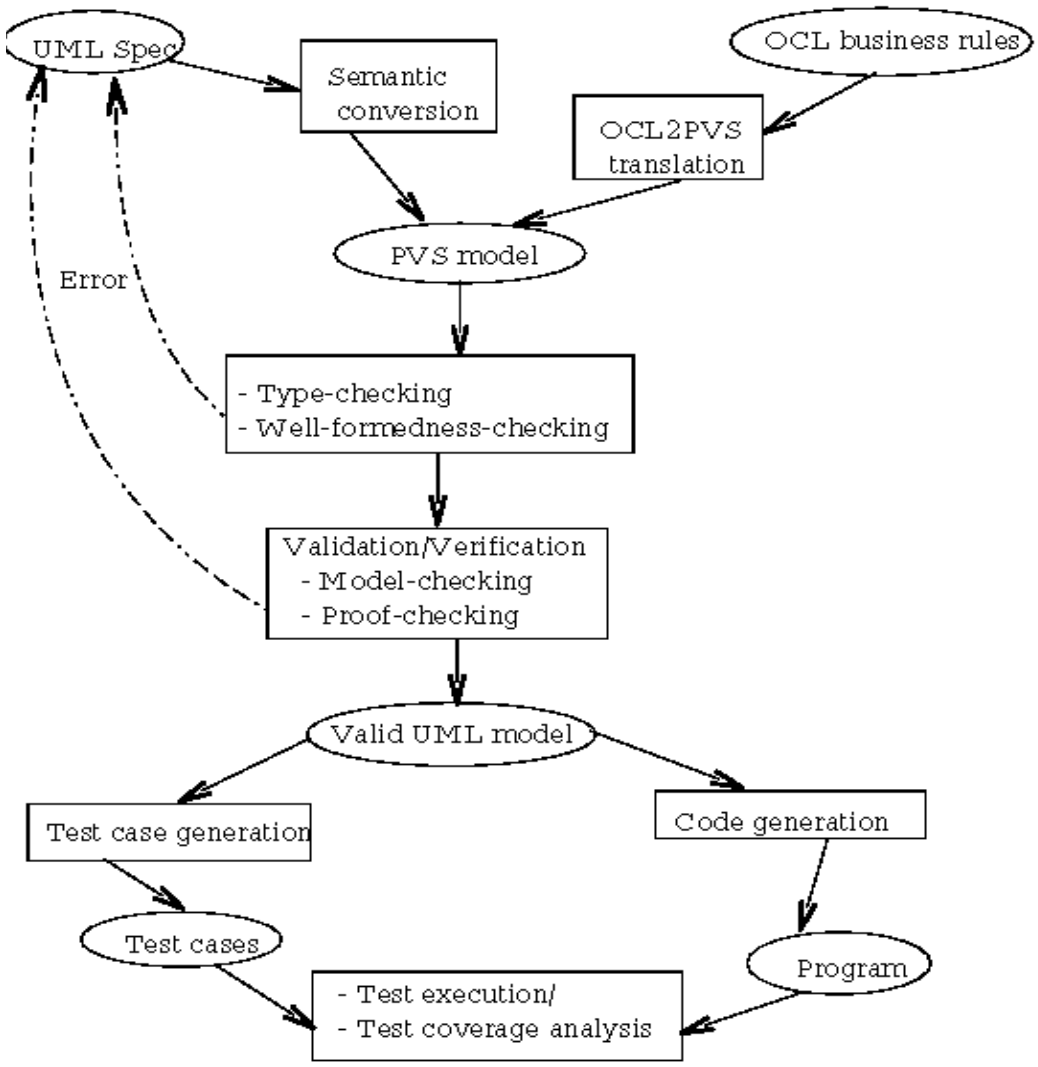

Figure 2: Architecture of the PrUDE Platform

5 Feasibility Study

5.1 Study Setup and Results

To illustrate feasibility of our approach, we have conducted a small experiment on the development of a critical system that provides a secured patient document service (PDS). The main functionality of the PDS system is to provide secured accesses to patient medical records worldwide. The system is required to provide security, i.e. it must provide authenticity, integrity, confidentiality, and authorization. In the sequel we briefly summarize major aspects of the experiment to show feasibility of the approach. The

experiment involved a group of six graduate students with varying background in modeling techniques and formal methods.
Three of them have industrial background and played the role of reviewers. The remaining participants were in charge of developing UML models. The result of the experiment shows that it is possible for a designer of a critical application, with a little knowledge of mathematical logic, to get the best out of graphical modeling techniques and formal methods: design using a visual notation, and design rigorously by taking advantage of the features provided by structured review and formal analysis techniques. Review of user requirements was done on the full document, which contains eight use cases. Subsequent review steps were conducted on sample use cases selected from the most critical ones. The analysis model involves eight business rules, six sequence diagrams, a statechart diagram, and a class diagram. The design model consists of six sequence diagrams, a statechart diagram, a class diagram, a static structural diagram and a collaboration diagram describing subsystems and their relationships, and design traceability documentation. We used a small test set consisting of fifteen expressions and twenty test cases.

We noticed that the effectiveness and cost of detecting defects vary significantly based on several factors: types of defects detected; whether the defects are detected manually or automatically; whether the detection method follows precise rules or is driven by experience and intuition, or both; backgrounds of the reviewers; and the size and complexity of user requirements.
Based on the cost and ease of detection, we group the defects discovered into five categories.

1. Defects discovered manually by using precise and systematic guidelines, e.g. consistency of UML diagrams, and test coverage analysis, which were easily and rapidly detected.
2. Defects discovered manually, which require some logical thinking and for which no clear guidelines were provided, e.g. consistency of business rules etc. These defects were discovered with a little more effort than the previous ones.
3. Defects discovered manually, which require some intuition and experience, and for which no strict guidelines were available, e.g. optimality, and robustness. These defects took more time to discover, and only half of them were detected.
4. Defects discovered automatically using the PrUDE tool, e.g. well-formedness of UML models, which were detected easily and quickly.
5. Defects related to validity were discovered using the PrUDE tool, but required some prior intuitive work from the reviewers in defining appropriate conjectures. The conjectures can be checked using the PVS proof strategies implemented in the PrUDE tool in less than a minute.

Though the size of the study materiel and the number of the participants don't allow us to draw quantitatively significant statistical conclusions, the results obtained, i.e. the number and kind of defects discovered are promising and consistent with our expectations.
In the future, we extend the experiment to a larger number of users and extend the study materiel to cover an entire system model.

5.2 The Patient Document Service (PDS)

5.2.1 Summary of the PDS Requirements

Overview: Binkadi Life, an insurance company needs to rapidly create an online healthcare marketplace. The central and initial component of that marketplace would be a patient document service (PDS) that provides support for the company 1,000,000 insured, care providers, benefit coordinators and agents. The initial version of the PDS will only maintain securely patient medical records and make them available to authorized persons worldwide. Subsequent versions are expected to expand the basic functionalities with several new services.

The goal of Binkadi Life is to deliver the services of the PDS at no additional cost to its insured. That'll allow them to increase their market share. At the same time they don't want to increase their operational costs. Hence it is essential for them to lower the development cost and to minimize the product support cost. Due to the highly competitive insurance market, it is also important for them to bring the product to the market the earliest possible, and to reduce the installation time (e.g. fastest deployment). Other important concerns include the following.

- Scalability: the system must scale to manage millions of users and work in complex computing environments.
- Availability: the system must have no more than 1 hour per month of down time.
- Performance: The system must be able to respond quickly to user requests, unless the network connection is broken (in which case the user should be notified).
- Portability: the system must work in diverse and complex computing environments involving various platforms and technologies.
- Ease of learning: the time for 90% of the buyers to learn (through supplied step-by-step instructions) how to use the first time the system must not be more than 10 minutes.

The system must also be fully integrated into existing enterprise security infrastructure. More specifically the PDS will reuse an existing secured database that provides the list of the primary users of the system (e.g. the company’s customers). The database provides also the list of registered doctors. This database is maintained by an already existing customer management system.

Functional Requirements: The main function of the PDS system is to provide secured accesses to patient medical record worldwide. The system must provide special protection features dealing with suspicious users and disclosure of unauthorized information. The actors involved in this system are the patients, patients' relatives and friends, doctors, and site administrators. The main resources to be secured are medical records of patients. A patient may choose a unique family doctor who is automatically granted the right to read and modify medical records of the patient. Only authorized doctors can read or modify a medical record. Every doctor is solely responsible for the modification that he made to the medical record database, and the system is expected to

enforce this responsibility. An authorized doctor is a registered doctor that a patient has chosen either as his family doctor or as "guest" doctor, e.g. a specialist, or for travel reasons or unavailability of family doctor etc. The patient is the only person that is allowed to choose his own doctor. A patient may have read access to his own medical record, but he cannot modify it. He may grant read access to his friends and family members. The site administrator is the only person who can create, delete, read and modify a patient record. The system is required to be secure, i.e. it must ensure that authenticity, and integrity, confidentiality, and authorization are always preserved. Additionally, the company would like to be able to use the system to maintain statistics about customers' behaviors in order to adapt its services to their needs, and also to send them some advertisements when they are using the system.

5.2.2 UML Specification

Use Case Diagram: We identify 10 use cases and 6 actors, which are described in the following.

Actors:

- User: a user can be a patient, a patient's friend, a chosen and registered doctor, a security officer; any authorized person.
- Patient: primary user of the system; corresponds to a regular customer of the company.
- Doctor: registered doctor specified by the company, and possibly selected by a patient.
- Friend: friend or relatives of a patient, is granted by a patient the right to access his record.
- Company database: provides the list of patients and registered doctors.
- Administrator: maintains the system.

Use cases:

- Registration: allows a user to register with the system; the first time, the user access the system using a default password delivered to him. Then during the registration, a new password is created that he must use for subsequent access to the system. Default registration of administrator, patients and doctors is made directly by the administrator using the company database. Default registrations of friends are made directly by patients.
- Transaction management: receives the user requests and handles them. The requests may consist of creating, getting, querying, updating or deleting a medical record. The request may be granted or denied, in which case an appropriate message is sent back to the user.
- Transaction processing: consists of the actual processing of the requests received from the user.
- Access control: checks whether any request or action by a user is legal or not. Its role consists of checking the request against the security policy, make appropriate decision, and enforce that decision.

- Login: identifies and authenticates a user in order to check whether he is an authorized user of the system. If that's the case, a session object carrying the security credentials of the user is created and the user is granted access to the system. Otherwise, access is denied and the user is kept out of the system.
- Auditing: any critical actions must be logged by specifying the type of action, the author and time.
- Administration: provides a series of functions that allows the administrator to maintain the system.
- Fault management: provides a series of functions that allows system monitoring, fault detection and recovery; sends notification to the user in case of unavailability.
- Maintain Statistics: maintain information related to the habits of the users of the system.
- Advertise: send advertisement to the users.

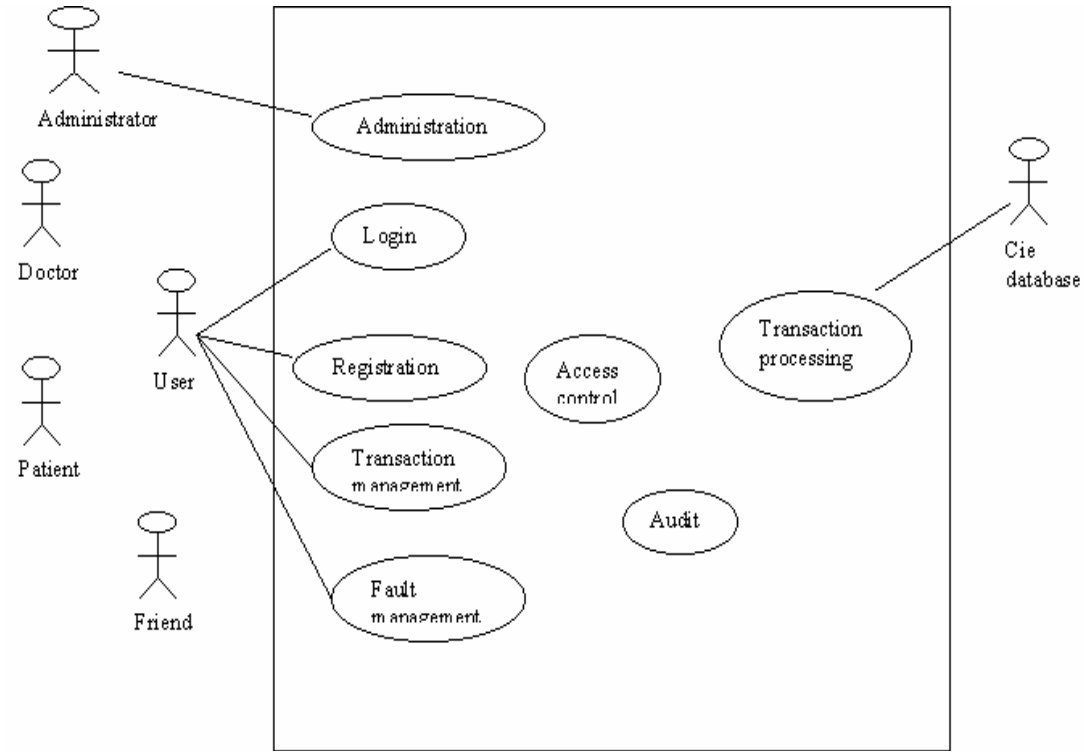

Figure 3: Use Case Diagram

The 10 use cases identified above cover the most important requirements. However, not all the use cases are significant for the design of the architecture. We must consider only use cases that involve the most important risks. These include use cases that are important for the user or the main purpose of the system, or that cover the quality and non-functional requirements. Secondary, ancillary use cases (e.g. features that are nice to have), or optional use cases are in general not significant for the architecture. In this perspective, we keep only 8 of the 10 use cases identified; the selected use cases are represented in the use case diagram given in Figure 3. Use cases Maintain statistics and

Advertise cover secondary requirements, which are not fundamental for the user. Hence they may be postponed and dealt with later.

Interaction Diagrams: Each use case involves one or more scenarios, each of which can be described using interactions diagrams (e.g. sequence or collaboration diagrams). Figure 4 depicts a scenario underlying the Login use case. The user identifies him by specifying his userid and password; the information is forwarded to the document server, which checks them against the security policy via a security manager. If the access is granted, a session object is created that carries the security profile of the user. If access is denied the user is notified by sending him an appropriate message.

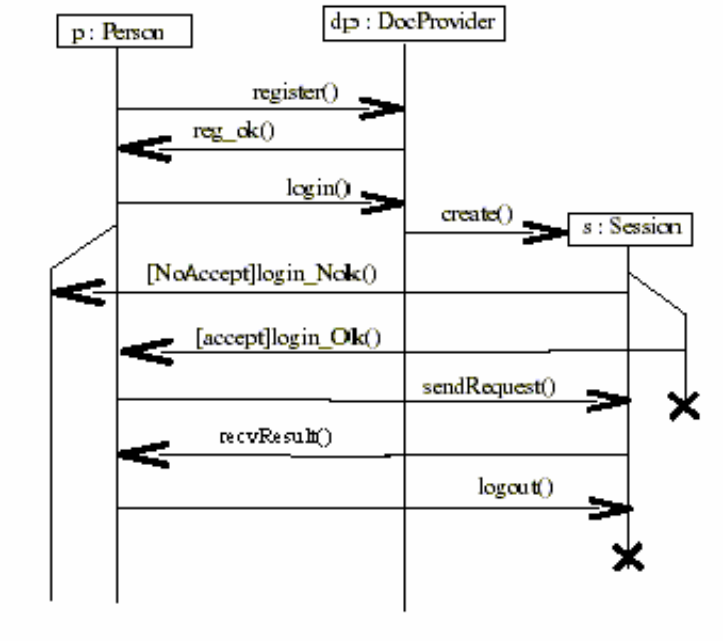

Figure 4: Sequence diagram describing login scenario

Class Diagram: The class diagram provided in Figure 5 depicts structural components of the system described above. The Patient, Doctor, Administrator and Friend classes represent potential users of the system. These classes are subclasses of the Person class that describes a set of common attributes. The DocProvider class manages the access to and delivery of medical records, which are described by the MedicalRecord class. The SecurityProfile of a user is defined as a set of AccessRight associated to the Person class.

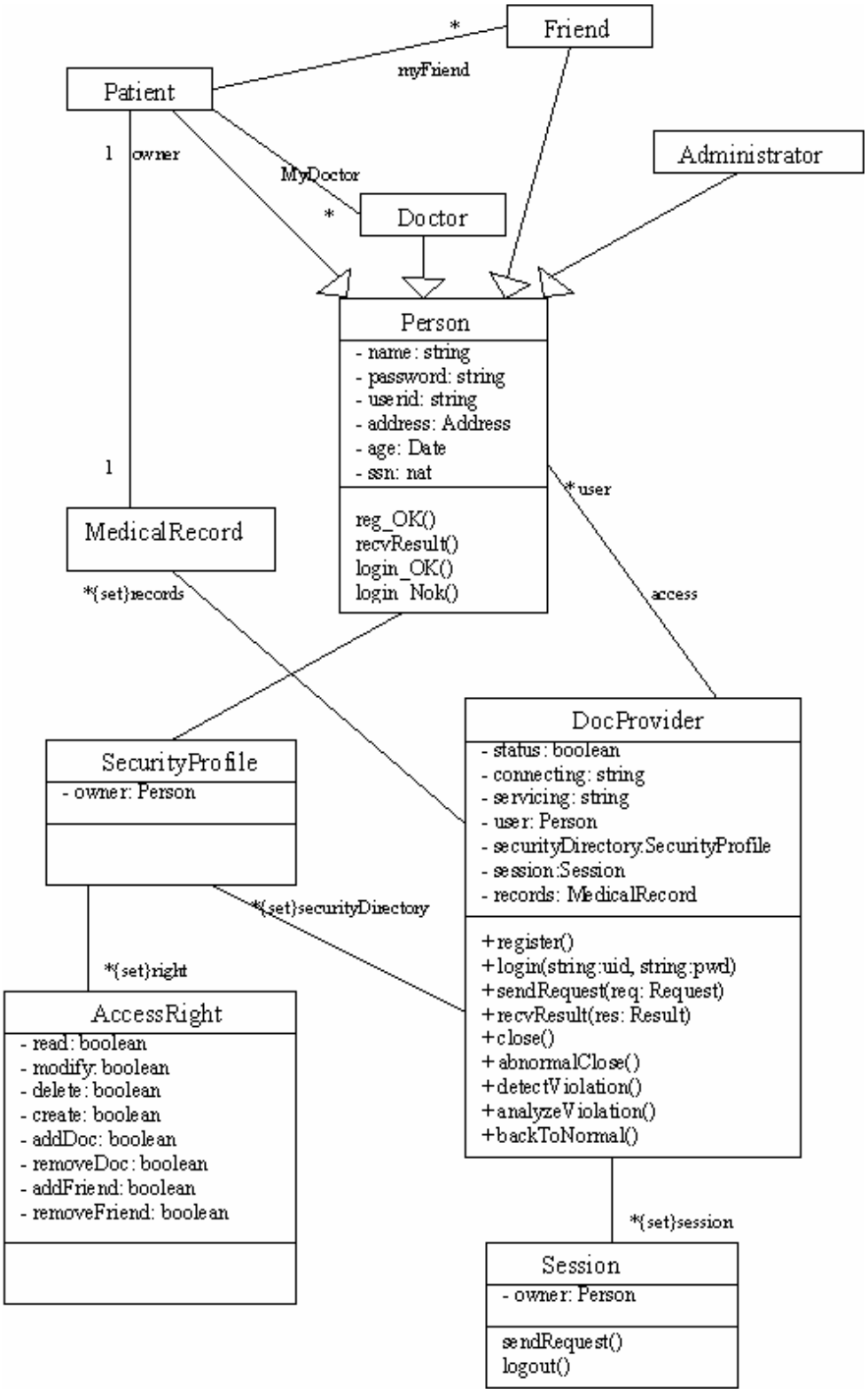

Figure 5: Class Diagram-Analysis

Statechart Diagrams: Statechart diagrams are provided for classes exhibiting significant dynamic behavior. For instance, the statechart diagram shown in Figure 6 describes dynamic behavior of the class DocProvider. The system starts in an initial state where security parameters are initialized. Then, it moves to an idle state where it waits for requests from users. When a request is received, the security profile of the user is checked and the request is either served or rejected.

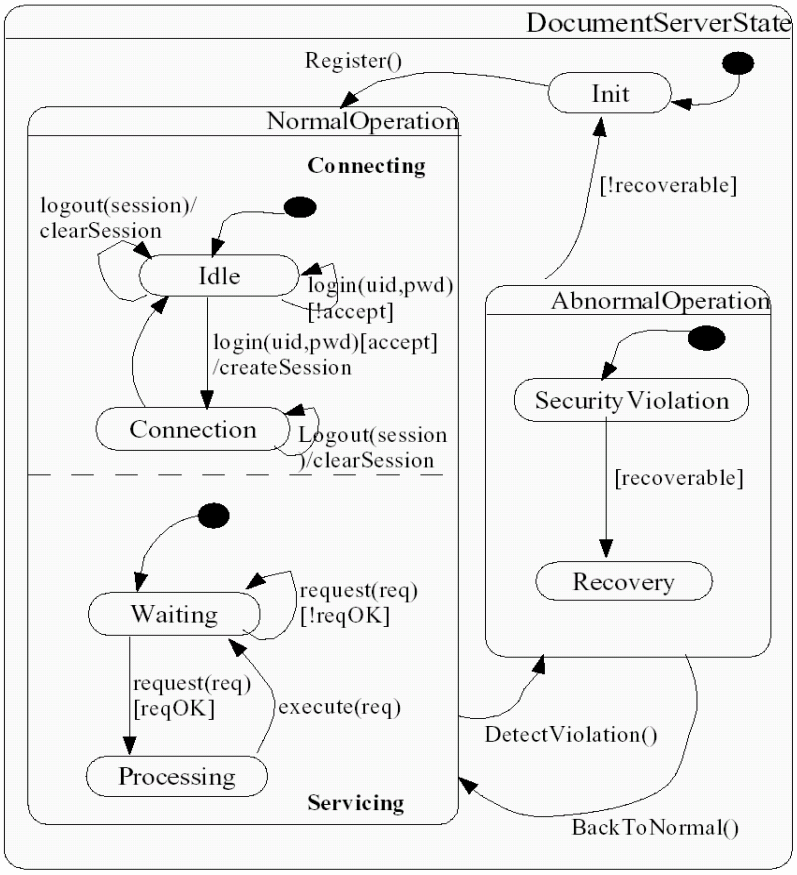

Figure 6: Statechart Diagram for Class DocProvider

Architecture: The general architecture of our system is based on the generalized framework for access control (GFAC)) that defines an architectural pattern for access control-based systems [22].

The collaboration diagram in Figure 7 depicts how subsystems collaborate in order to achieve a basic scenario in which a user makes a request to the system. The request is received by the enforcement facility, which submits it to the decision facility for verification. The decision facility refers to the security rules in order to check the validity of the request. Based on the response of the decision facility, the enforcement facility either executes the request or rejects it. In either case, the security information base must be updated.

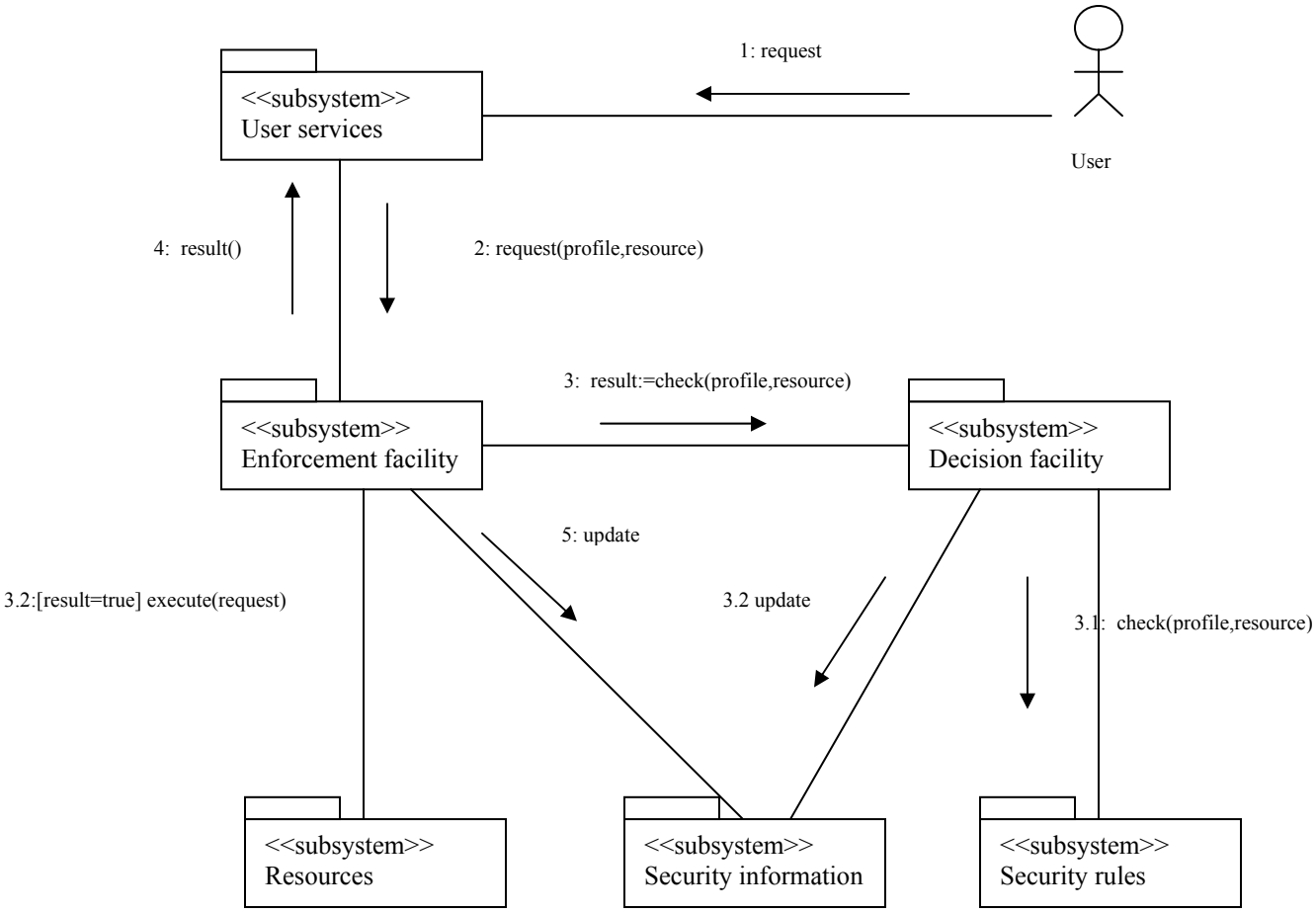

Figure 7: Collaboration among major subsystems

Figure 8 describes the hardware and network topology and shows how the major components are deployed on the hardware infrastructure.

5.2.3 Complementary Semantics

The standard UML notation provides only a partial specification of the system. The UML specification produced needs to be extended by providing complementary semantics for the elementary features (e.g. state, actions, conditions etc) and properties involved using language like the Object Constraints Language [12] or any other mathematical or textual languages. We define in the following the complementary semantics for the statechart shown in Figure 6 using OCL. The context of the expression is a DocProvider object.

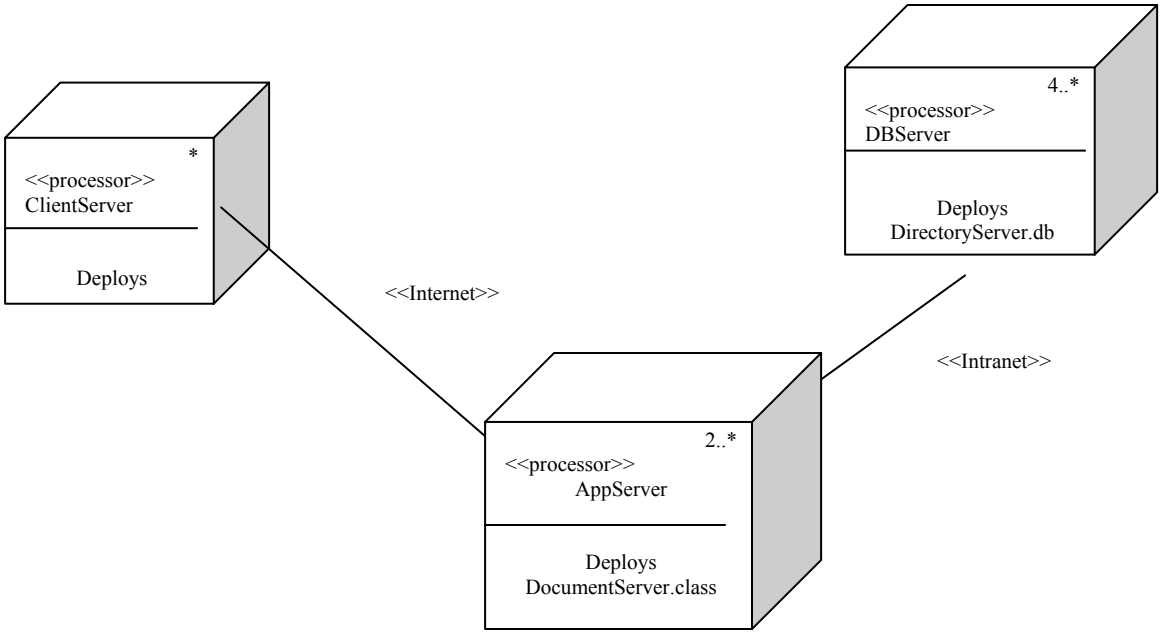

Figure 8: Deployment Diagram

Predicates associated to states

predIdle(): Boolean
   self.status = true AND self.connection = false

predConnection: Boolean
   self.status = true AND self.connection = true AND self.users → notEmpty

predWaiting(): Boolean
   self.status = true AND self.service = false

predProcessing(): Boolean
   self.status = true AND self.service = true AND self.sessions → notEmpty

predSecurityViolation(): Boolean
   self.status = false AND self.securityOK = false

predRecovery(): Boolean
   self.status = false AND self.securityOK = true

Predicates associated to guard conditions

predAccept(sp:SecurityProfile): Boolean
   exists (sp | self.securityDirectory.includes(sp) AND
                sp.owner.userid=uid AND
                sp.owner.password = pwd)

Predicates associated to actions

```
predReqOK(sp:SecurityProfile, ac:AccessRight, req:Request): Boolean
   exists ((sp, r, req) | self.securityDirectory.includes(sp) AND
                          sp.owner=req.source   AND
                          sp.right.includes( ac) AND
                          ac=req.action)

predCreateSession(): Boolean
   exists (self.sessions→size = self.sessions→size + 1)
```

5.2.4 Business Rules

The UML business model needs also to be augmented by defining the business rules. These rules can be expressed using OCL. We give in the following some examples of business rules.

Rule1: A patient cannot create, delete or modify his own medical records.

```
context Patient
inv self.profile.right  → forAll(r | not (r = r.create or r.modify or r.delete))
```

Rule 2: A doctor cannot create or delete a medical record.

```
context Patient
inv self.myDoctor.profile.right → forAll(r | not (r.create or r.delete))
```

Rule 3: A doctor that has not been chosen by a patient (as a family doctor or a friend), cannot access the patient's medical record.

```
context MedicalRecord
inv self.owner.myDoctor → (excludes(doc)) implies not
                                  self.owner.myDoctor.profile.right →
                                       exists(r | ((r.resource=self) and
                                                   (r.read or
                                                    r.modify or
                                                    r.delete or
                                                    r.addDoc or
                                                    r.removeDoc or
                                                    r.addFriend or
                                                    r.removeFriend))))
```

Rule 4: Only a site administrator can create or delete a medical record.

```
context MedicalRecord
inv self.person.profile.right →
        exists (r | (r.create or r.delete)) implies
                person.asType(Administrator))
```

Rule 5: A patient can read only his own medical record unless another patient has chosen him either as a "friend" or a doctor or he is a site administrator.

```
context MedicalRecord
```

```
inv self.patient.profile.right →
        exists(r | (r.resource =self and r.read) implies
                    (self.patient=self.owner or
                      owner.myFriend → includes(patient) or
                      owner.myDoctor → includes(patient))
```

5.3 Structured Reviews

Well-formedness and consistency arguments, as we already mentioned, may be checked automatically using the PrUDE toolkit. This is performed after the PVS semantic model corresponding to the UML model is generated. The remaining arguments are checked manually or semi-automatically. In the rest of this section, we show, by examples, how this can be conducted.

In order to check the traceability argument the reviewer will first examine the relationships between the structural and behavioral elements defined in the specification and the design documents. The analysis model provided in Figure 5 is refined into a new design model given in Figure 9. Instead of having several classes for different users of the system, Person, Patient etc., there is only one user class, namely the UserManager class which carries the same set of attributes as Person class, in addition to a role attribute that corresponds to the specific role played by the user.

The SecurityManager class is a new class that performs all necessary security checks before executing a request. There is also a standard directory service represented by DirectoryService class. Since the configuration of the model has changed, ensuring design traceability is important. That consists of showing that all information mentioned in the abstract model can be found in the design model.

For instance, the designer may consider that there is a direct correspondence between DocProvider class in the abstract model and SecurityManager class in the design model. The same correspondence may also exist between Patient, Doctor, Friend, Administrator and User. The correspondence is documented by providing retrieve functions that relate abstract and concrete representations. We use the following notation for retrieve function: retr: [Rep → Abs], where Abs is the abstraction and Rep is a representation. For instance, for the SecurityManager class, the following retrieve function can be defined:

```
retr: SecurityManager → DocProvider
context DocProvider
sm: SecurityManager
inv self = retr(sm) implies
    (self.records = retr(sm.records)  and
      self.securityDirectory = retr(sm.securityDirectory) and
                                self.users = retr(sm.users))
```

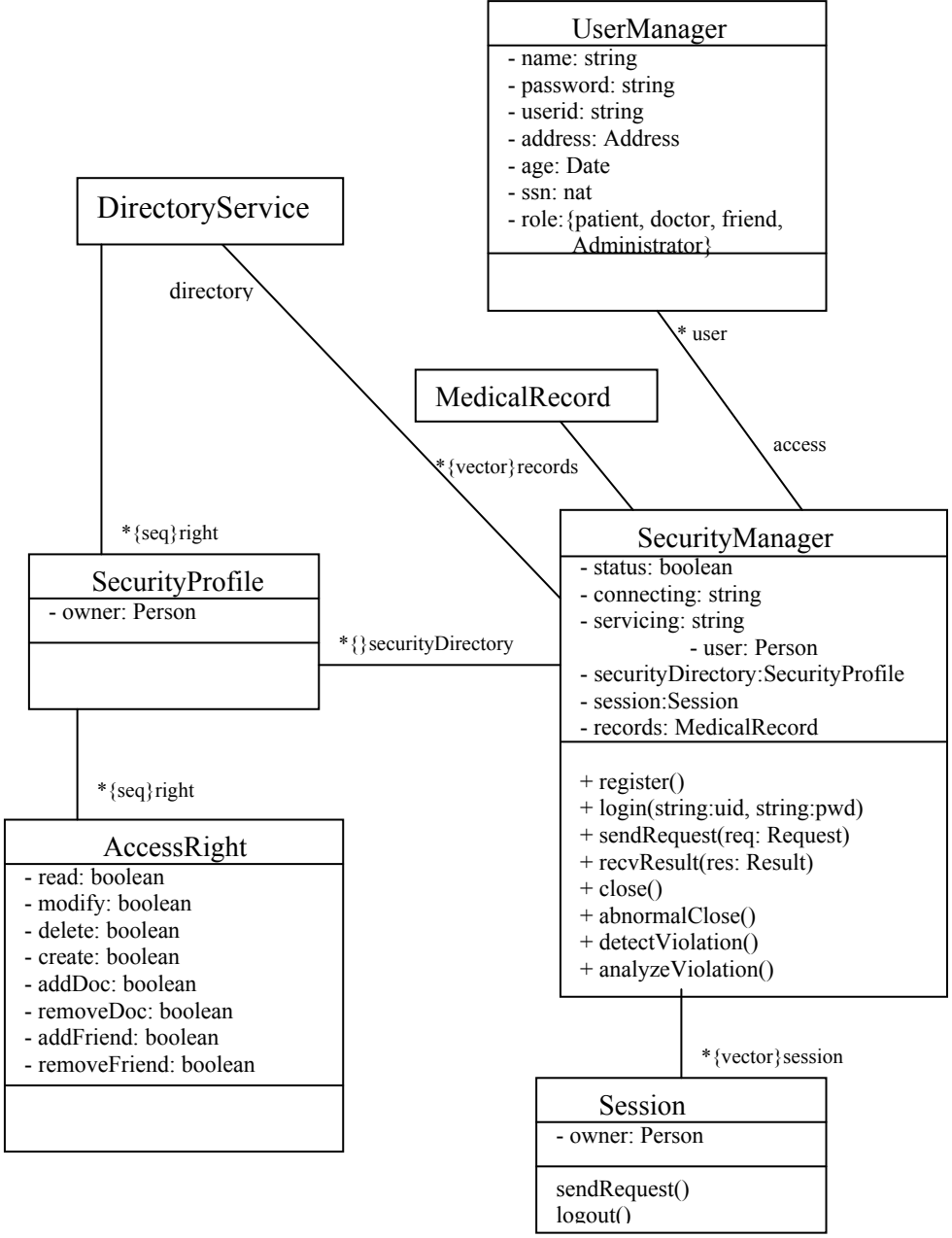

Figure 9: Class Diagram (Design)

The retrieve function for the classes is defined in terms of the retrieve functions of their attributes that must also be defined. The retrieve function can be as simple as the identity function or more complex in case where the data types involved are modified. For instance, the above retrieve function establishes correspondence between the records attributes in, respectively, the DocProvider and SecurityManager classes. However, their data types are different (see the respective class diagrams). The abstract records attribute is defined as a set of MedicalRecord whereas the refined one is defined as a vector of MedicalRecord, for example, an array. The retrieve function for the attribute records may be defined in this case as follows:

retr(sm.records) = {sm.records[i] | mid 0 <i < sm.records.size}

The abstract attribute records are defined by the retrieve function as the set of elements contained in the concrete representation vector. In order to establish correctness of the representation, an adequacy proof obligation may need to be discharged. The following proof obligation states that the retrieve function must be total:

context DocProvider
inv self → forAll(dp | SecurityManager →
exists(sm | retr(sm.records) = dp.records)))

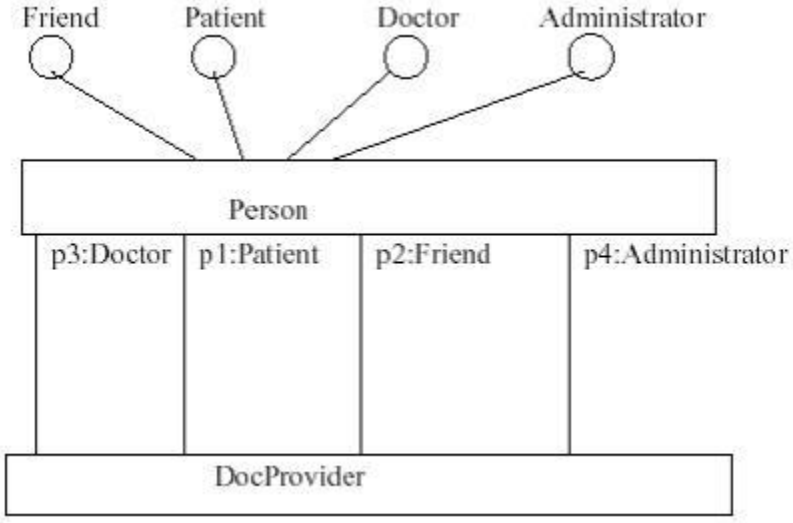

Figure 10: Dynamic Reconfiguration in the Patient Document Service

The proof obligation is discharged straightforwardly by providing the following informal constructive argument:

Given a finite set, it is always possible to arrange elements of the set into an array. The set will represent the collection of elements associated to that array.

The use of informal constructive arguments to discharge simple proof obligations is encouraged in [23]. Although the data representation chosen by the designer seems adequate, the reviewer may raise some concerns about its optimality. From the requirements, it appears that the attributes records where all medical records are stored should allow efficient searching. The question will be whether representing the records as a binary tree would be more efficient than using just a vector?
A robustness issue raised during the review process was due to the fact that the patient is the only person allowed to choose his doctor. How about the case when a serious accident has happened to the patient at the other end of the world where the authorized doctors listed in his record cannot reach him, and the patient is not in condition to choose a local doctor?
Another robustness issue is due to the assumption that there could be some security violations since no system is absolutely secure. Hence, we need to design a mechanism that allows the system to discover, analyze and recover from security violations. The

statechart diagram given in Figure 6 by specifying appropriate recovery mechanisms already addresses this concern.

The review also established that the design was not valid, because it failed to describe, consistently, user requirements that state the fact that a patient must not be able to modify his own record. A patient can be a doctor by profession in which case he can choose himself as a "guest" or family doctor, and grant himself the right to modify his own record, as the above system design does not prevent him from doing so. To be valid the business rules should be rephrased stating that a patient may choose, as a family or a "guest" doctor, any person who is a registered doctor, except himself or herself. An additional business rule may be stated as follows:

Rule 6: If a patient is a doctor, (s)he cannot choose himself as his doctor.

context Person
inv (self.asType(Patient) and
self.asType(Doctor)) implies
self.myDoctor → *excludes*(self))

Another possible solution is redesigning the model in order to incorporate some dynamic reconfiguration features as shown in Figure 10. The solution adopted in Figure 10 describes different roles a Person may play, with interfaces Patient, Doctor, Administrator and Friend. In this way, the interfaces may be constrained to prevent the same object of the Person class from playing roles that may violate the requirements.

5.4 Formal Verification

A reviewer can check well-formedness and validity arguments using the PrUDE tool. Importing the XMI file generated from the UML models does this, and PVS semantics models are automatically generated based on the XMI file. The business rules are translated into PVS and systematically integrated with the PVS semantic models using the property editor. Then, the model is checked based on the UML well-formedness rules, whereas invoking the PVS type-checker in a batch mode checks type-correctness. Finally, invoking the PVS theorem-prover checks every system property.

5.4.1 Statechart Diagrams

Figure 11 shows a snapshot of the PVS semantics generated for DocProvider' statechart diagram in PrUDE; the lower window is a log area where reports generated from the PVS tools are displayed. Figure 12 shows a snapshot of the property editor through which complementary semantics are inserted. The log areas show reports of well-formedness and type checking.

In order to check validity of the specification, the reviewer draws and checks conjectures based on requirements. An example conjecture suggested by one of the reviewers enabled us to discover an interesting bug in the statechart diagram of Figure 6. The conjecture is expressed as follows:

Rule 7: A user cannot logout unless (s)he is connected.

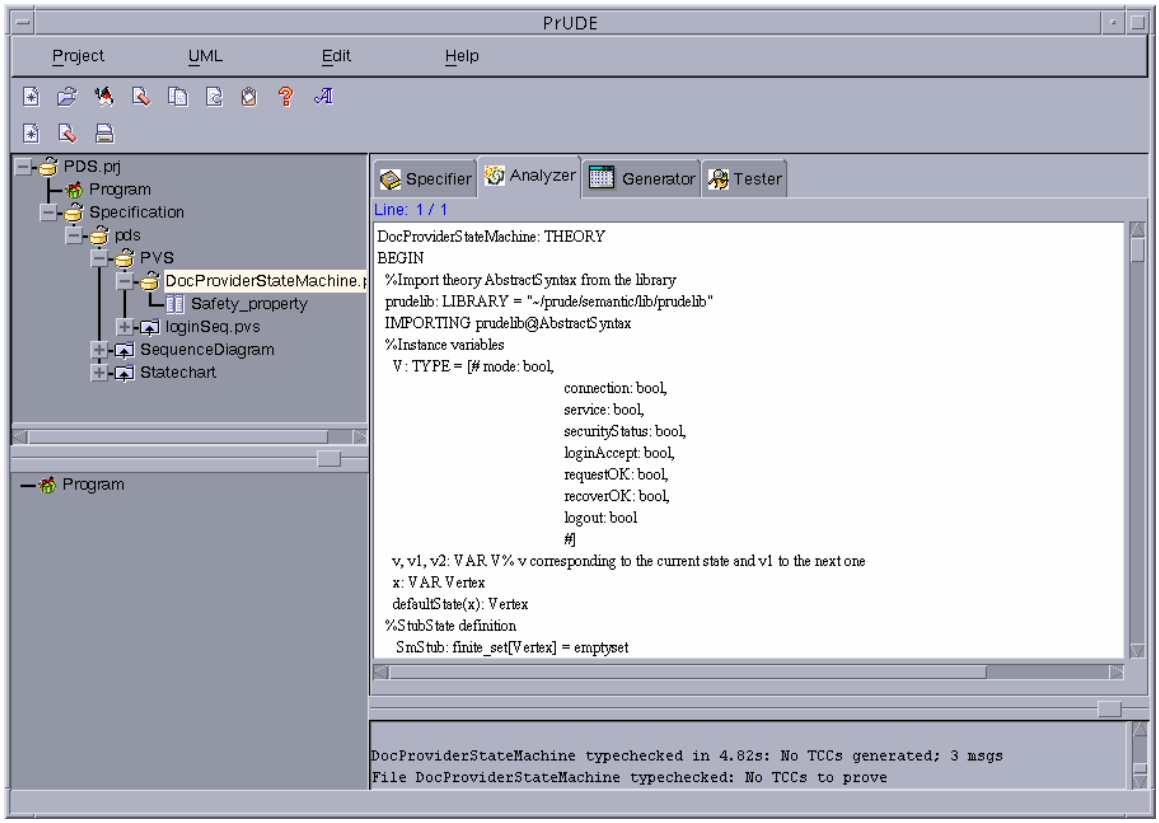

Figure 11: PVS Semantics Generation Using PrUDE

The reviewer invoked the theorem-prover to discharge the conjecture. The proof was unsuccessful as depicted in Figure 13. A counterexample described as a PVS debugging message was then returned; the counterexample is shown in Figure 14.

The message is expressed in the form of unproved sequent with several antecedents and no consequent to be proved. In such a case, either there is a conflict in the antecedents, or the antecedents are not sufficient to prove the sequent. Lines {-1} to {-4} refer to the simple state Connected. Line [-5] refers to a transition instance (labeled internally) tr!1 whose source and target is the state Connected, with triggering method logout, empty guard condition, and action clearSession. This corresponds to the self-transition associated to the state Connected. Lines [-6] to [-11] refer to the firing of transition tr!1. At this stage the reviewer inferred that the firing of transition tr!1 leads to an inconsistent state, and decided to examine closely the transition and its meaning as defined in the statechart diagram.

In a normal execution, the concurrent state Connecting contains a logical inconsistency. If we follow the single process of handling a user connecting to the Document Server we can determine the following normal operations:

1. The thread responsible for user connection is started in the *Idle* state.

2. If the thread receives invalid login request from unconnected user, it remains in the *Idle* state.
3. If the thread receives a login request with valid user ID and password from unconnected user, it enters the *Connected* state.
4. After the user is connected the thread responsible for handling user connections returns to the *Idle* state.
5. When the thread in the *Idle* state receives a logout request from a connected user, it handles the request and remains in the Idle state.

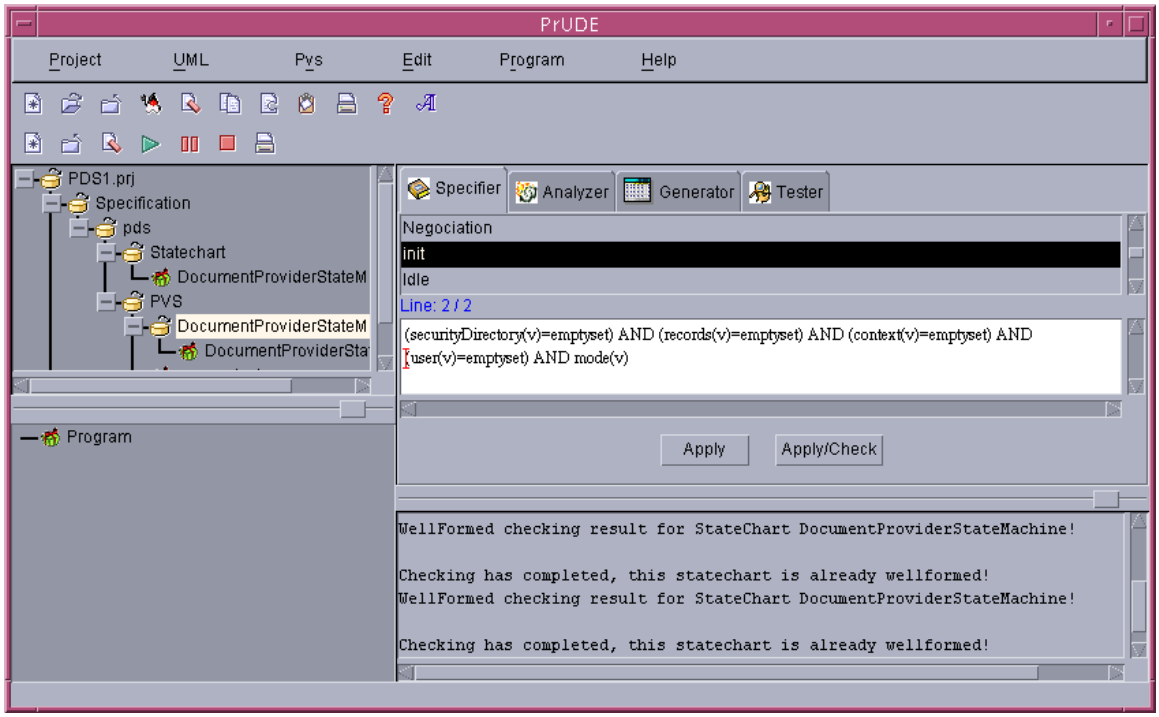

Figure 12: Property Editor

These operations seem consistent with a running server. The transition that appears logically inconsistent when compared to the implementation of the system is, as indicated by the counterexample, the transition from the *Connected* state to itself triggered by a logout request.
In reality, a logout request from a user who is not connected should not be processed. This problem could occur, if for example, the implementation code did not properly set the connection property of a client after it has successfully logged in; rather, it is set before completion of the connecting code. Combine this with conditional logic that checks the connection status of the client before allowing the logout operation to proceed. Although the detected error might seem trivial, it is, however, an example of typical errors that can easily be skipped over during manual review.

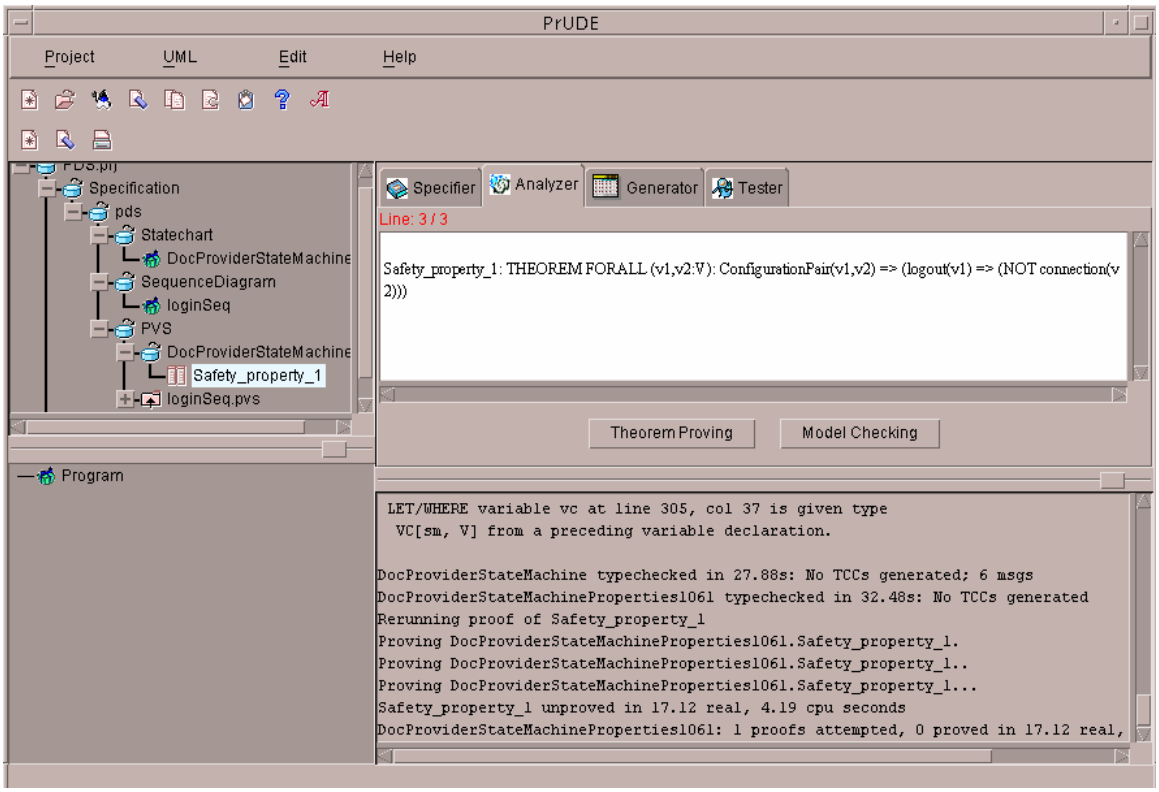

Fig. 13: Unsuccessful Proof attempt for Property 7.

5.4.2 Sequence Diagrams

Figure 15 shows a snapshot of the PVS specification automatically generated using the PrUDE toolkit from the UML sequence diagram shown in Figure 4.

Some essential conjectures suggested by reviewers are security requirements for authorization, authentication, accountability, and availability. An instance of suggested authorization property states that a non-discretionary authorization policy is needed in order to control PDS users' access to patient records.

Rule 8: Unauthorized access to a patient record is not allowed. Access to a patient record by an already authenticated user of PDS must be protected by the enforcement of an authorization policy.

An example of proposed accountability property is stated as the following property:

Rule 9: Every access to records must be logged, including unauthorized attempts to access the information.

Analyzing the interactions involved in the system, such as the one modeled by the sequence diagram given in Figure 4 can check such kind of properties. In the classical message sequence charts (MSC), deterministic ordering of events can be guaranteed by using the general ordering mechanism. The UML sequence diagram, however, does not

support such a mechanism, and hence a need for formal semantics that ensure enforcement of this sort of properties of systems. The sequence diagram shown in Figure 4 describes interactions among instances of Person, DocProvider, and MedicalRecord classes. It constrains the messages to occur in the order they appear in the diagram from top to bottom. The diagram does not, however, state whether any of the messages must occur or may occur. To model dependencies among messages, one needs formal representation of sequence diagrams. Suppose that, in Figure 4, the message create occurs only if messages reg_ok and login occur in that order. This property cannot be specified by the graphical notations and induces a strong need for formal semantics.

For instance, in our semantic definitions, Rule 8 is equivalent to the following:

For every valid trace of the sequence diagram (cf. fig. 4), an occurrence of data access event must be preceded by a successful registration and authentication. In other words, in a valid trace, the events sendRequest and recvResult must be preceded by both the reg_Ok and login_Ok events.

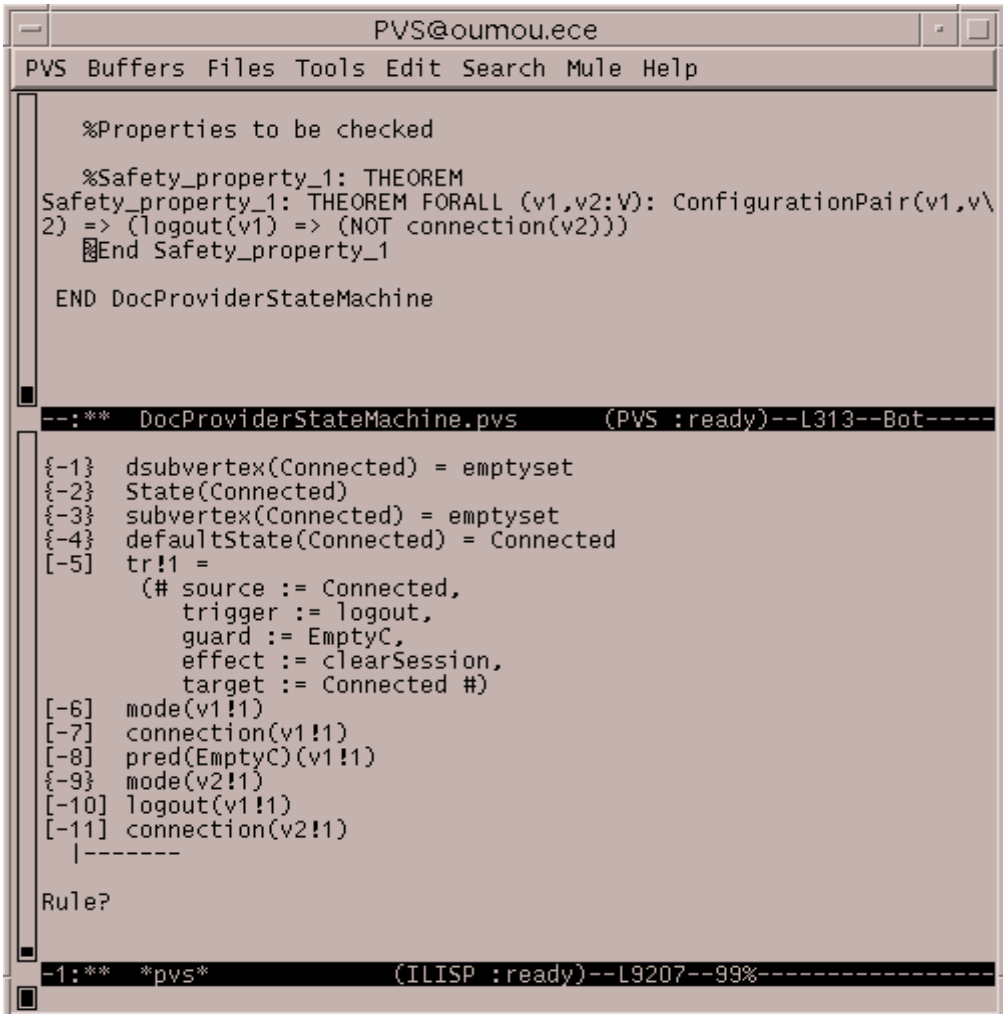

Figure 14: Counterexample for Property 7.

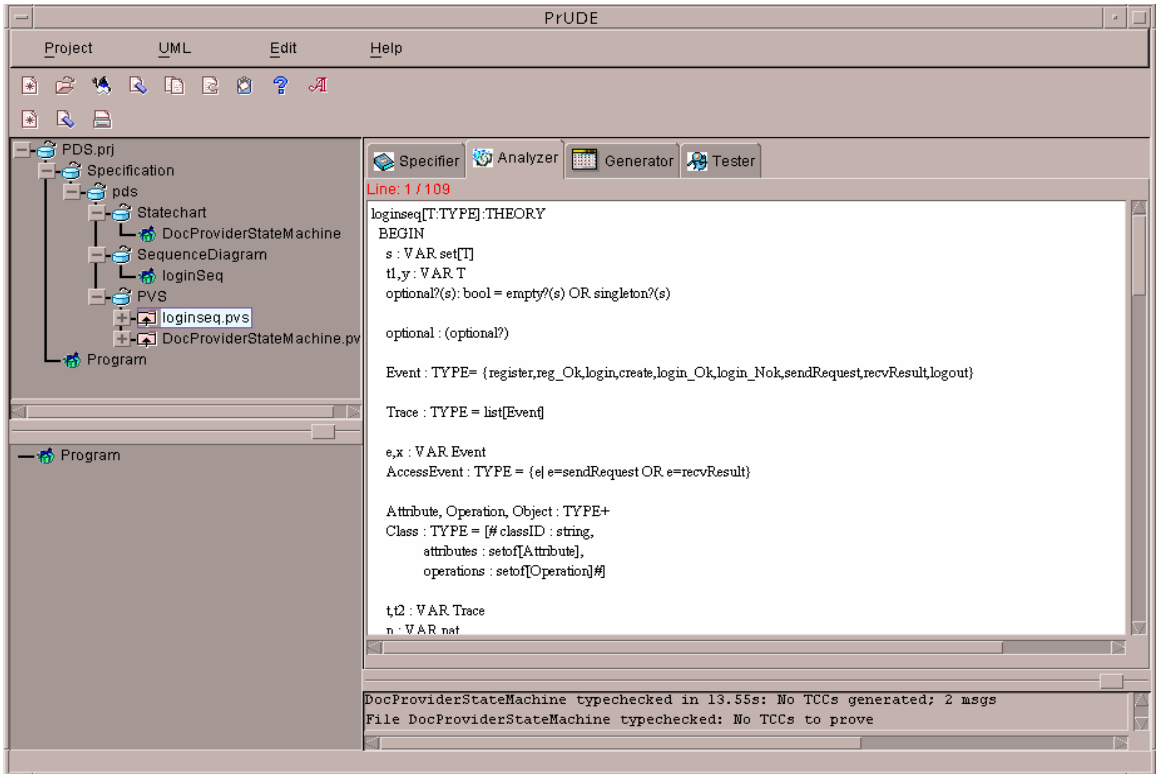

Figure 15: Semantics Generated Using PrUDE for a UML Sequence Diagram

System requirements are specified as predicates. To ensure that the system specification fulfills the requirements, appropriate conjectures or theorems are stated and discharged by invoking the PVS prover. The requirement specified by Rule 8 is stated as follows:

th1: **THEOREM**
FORALL e: Event, t: Trace: (e = sendRequest **OR** e = recvResult) **IMPLIES**
((t ∈ traces(loginseq) **AND** e ∈ t) **IMPLIES** {reg_Ok, login_Ok\} ⊂ prefix_upto(rank(e,t),t))

The security requirement characterized by rule 8 can be implemented in several ways. The most critical case is when an authorized user is tampering with records without leaving a trace of his identity. Users must not be allowed to access patient records after they have invoked the logout operation. This is stated as follows:

th2: **THEOREM**
FORALL e, t, sqdr: (t ∈ traces(sqdr) **AND** e ∈ t **AND** e=logout)
IMPLIES isucc(e,t) ⊆ {e|e=login}

Where the *isucc*(e,t) function returns the set of immediate successors of the event e in trace t. Invoking the PVS theorem-prover in a batch mode and using the proof strategies defined previously discharges the above theorems. Figure 16 shows a snapshot of the proof-checking report for rule 8 in PrUDE.

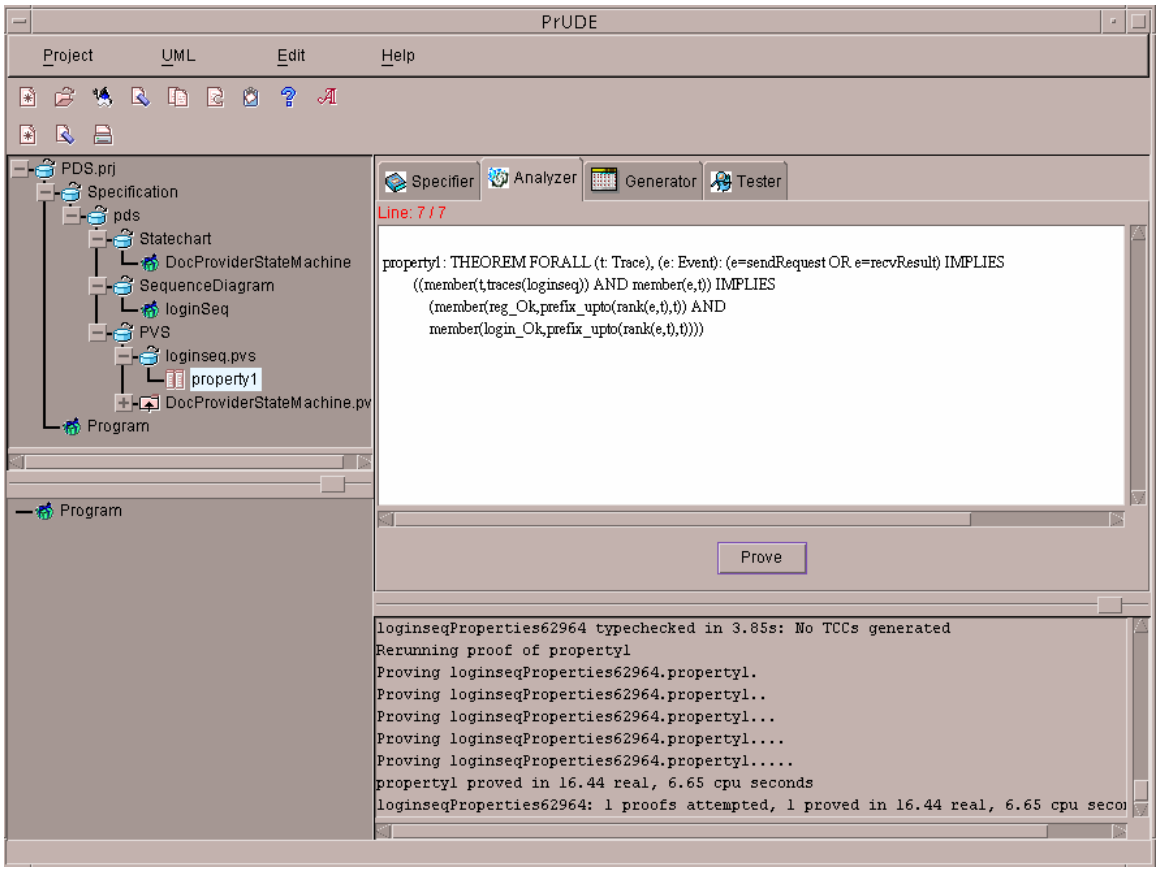

Figure 16: Proof Checking Report for Rule 8

5.5 Model-based Testing

5.5.1 Test Case Generation

Due to space limitations, we present in the sequel only results related to one test strategy, namely the transition test strategy. At the implementation level, test cases are collected and generated based on the constraints and invariants involved in the UML and OCL specifications. We present in this section an example of test case generation involved in object domain analysis for method login() of the class DocProvider based on the transition test strategy.

There are 2 transitions involving method *login*(): a transition that originates from state Idle and arrives in state Connected, and a self transition that loops in state Idle. Based on the predicates associated with the elements of each transition, we identify the two pre-post condition pairs associated to method *login*() as follows:

DocProvider::login(uid:string, pwd:string):true
pre1: predIdle( ) and predAccept( )
post1: predConnected( ) and predCreateSession( )

DocProvider::login(uid:string, pwd:string):false

pre2: predIdle( ) and not predAccept( )
post2: predIdle( )

Due to the object variables involved in the pre and post condition pairs, the object domain analysis technique is used for test case generation. Having replaced the predicates involved with their respective referenced expressions based on the rules mentioned previously, we obtain the following expressions:

$\forall$ dp: DocProvider, $\exists$ sp $\in$ dp.securityDirectory, owner: Person
pre1 = (dp.mode = = true $\wedge$ dp.connection = = false) $\wedge$
$\sim$(dp.sp.owner.userid = uid $\wedge$ dp.sp.owner.password = pwd)
post1= (dp.mode = = true $\wedge$ dp.connection = = true) $\wedge$
(dp.sessions.size()' = dp.sessions.size()+1)

$\forall$ dp: DocProvider, $\exists$ sp $\in$ dp.securityDirectory, owner: Person
pre2 = (dp.mode = = true $\wedge$ dp.connection = = false) $\wedge$
(dp.sp.owner.userid = uid $\wedge$ dp.sp.owner.password = pwd)
post2= (dp.mode = = true $\wedge$ dp.connection = = false)

where dp.securityDirectory is a set of SecurityProfiles and dp.owner is a Person object.

After that, we need to break the preconditions into DNF expressions, but, in this case, the preconditions are already in normal formal. Test cases can be defined by analyzing the domain of the object variables involved. The instances of these objects are first built based on the object decision tree, and then, our extended domain matrix technique is used to identify and organize the test cases. Table 1 from (a) to (d) shows the construction of the instances for objects Person, AccessRight, SecurityProfile and DocProvider, respectively. Table 2 shows the generated test cases for the pre-post condition pair 1 of the method login, and Table 3 shows the generated test cases for the pre-post condition pair 2 of the method login. We obtain in total eight potential test cases for both pre-post condition pairs. But only three of the eight test cases, which make the postcondition true (indicated by a "TRUE" in the Expect Results row), correspond to effective test cases. The remaining potential test cases falsify the preconditions, so we can't conclude anything after executing them.

Table 1(a): Instances For Class Person

No.	Object Var.	Instance Variable					
		Name	userid	password	address	ssn	age
1	p1	Alex	alex	camry	40 Bay St.	1234567	20
2	p2	Alex	alex	camry	40 Bay St.	1234567	20
3	p3	Alex	alex	camry	40 Bay St.	1234567	20
4	p4	Alex	alex	camry	40 Bay St.	1234567	20
5	p5	Alex	alex	camry	40 Bay St.	1234567	20

Table 1 (b): Instances For Class AccessRight

No.	Object Var.	Instance Variable					
		Read	modify	create	delete	addFriend	addDoctor
1	ac1	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
2	ac2	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
3	ac3	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
4	ac4	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
5	ac5	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE

Table 1 (c) Instances For Class SecurityProfile

No.	Object Var.	Instance Variable	
		owner	right
1	sp1	p1	ac1
2	sp2	p2	ac2
3	sp3	p3	ac3
4	sp4	p4	ac4
5	sp5	p5	ac5

Table 1 (d) Instances For Class DocProvider

No.	Object Var.	Instance Variable				
		mode	connection	service	securityStatus	securityDirectory
1	dp1	TRUE	FALSE	False	False	sp1
2	dp2	TRUE	FALSE	False	False	sp2
3	dp3	TRUE	FALSE	False	False	sp3
4	dp4	TRUE	FALSE	False	False	sp4
5	dp5	TRUE	FALSE	False	False	sp5

Table 1: Construction of the Instances for Object Variables

Domain Matrix For method login() in Class DocProvider
(for pre/post pair 1)

Boundary			Test Case			
Instance Var.	Condition	type	1	2	3	4

	dp.mode==true&& dp.connection==false&& dp.sp.p.userid=uid && dp.sp.p.password=pwd	on	dp1			
		off		dp2	dp3	dp4
Dp	Typical	in				
		on				
		off				
Uid	Typical	in	alex	alex	smith	smith
		on				
		off				
Pwd	Typical	in	camry	honda	camry	honda
Expected Results			TRUE	FALSE	FALSE	FALSE

Table 2: Test Cases for the Pre and Post Condition Pair 1 of Method Login

Domain Matrix For method login() in Class DocProvider
(for pre/post pair 2)

Boundary			Test Case			
Instance Var.	Condition	type	1	2	3	4
	dp.mode==true&& dp.connection==false&& dp.sp.p.userid!=uid	on			dp3	dp4
		off	dp1	dp2		
Dp	Typical	in				
		on				
		off				
Uid	Typical	in	alex	Alex	smith	smith
		on				
		off				
Pwd	Typical	in	camry	honda	camry	honda
Expected Results			FALSE	FALSE	TRUE	TRUE

Table 3 Test Cases for the Pre and Post Condition Pair 2 of Method Login

5.5.2 Test Data Review

The review of test data consists of checking the expressions used to generate the test cases in the PrUDE tool. Although the test expressions look simple, they are still subject to errors. The role of the reviewer is to ensure their correctness with respect to their specification, i.e. the abstract specification.

The coverage criteria guiding the review are specification-based testing. For the transition test strategy, we define three coverage criteria, namely *transition coverage, DNF coverage*, and *condition coverage*.

- *Transition coverage* criterion is defined in terms of the state diagram of a class. At a minimum, a tester should test every transition in the state diagram at least once. Transition coverage is analogous to statement or branch coverage at the code level.
- *Precondition coverage* criterion requires that every DNF involved in a precondition be covered by at least one test case. A DNF consists of one or several elementary boolean conditions.
- *DNF coverage* criterion is based on the rationale that each condition should be tested independently without interference from other conditions. In order to achieve that, the test set must include at least one test case that makes all conditions true, and test cases that falsify each condition at least once.

5.5.3 Test Execution

After reviewing them, test data are used as basis for test execution. Test execution starts at the class level by testing the individual methods involved in the class. Individual methods are tested by creating an instance of the class and setting the test values (i.e. an initial state) using the reflection API. After calling the method on the modified instance, we get the new state of the object still using the reflection API, and then finally evaluate the post conditions.

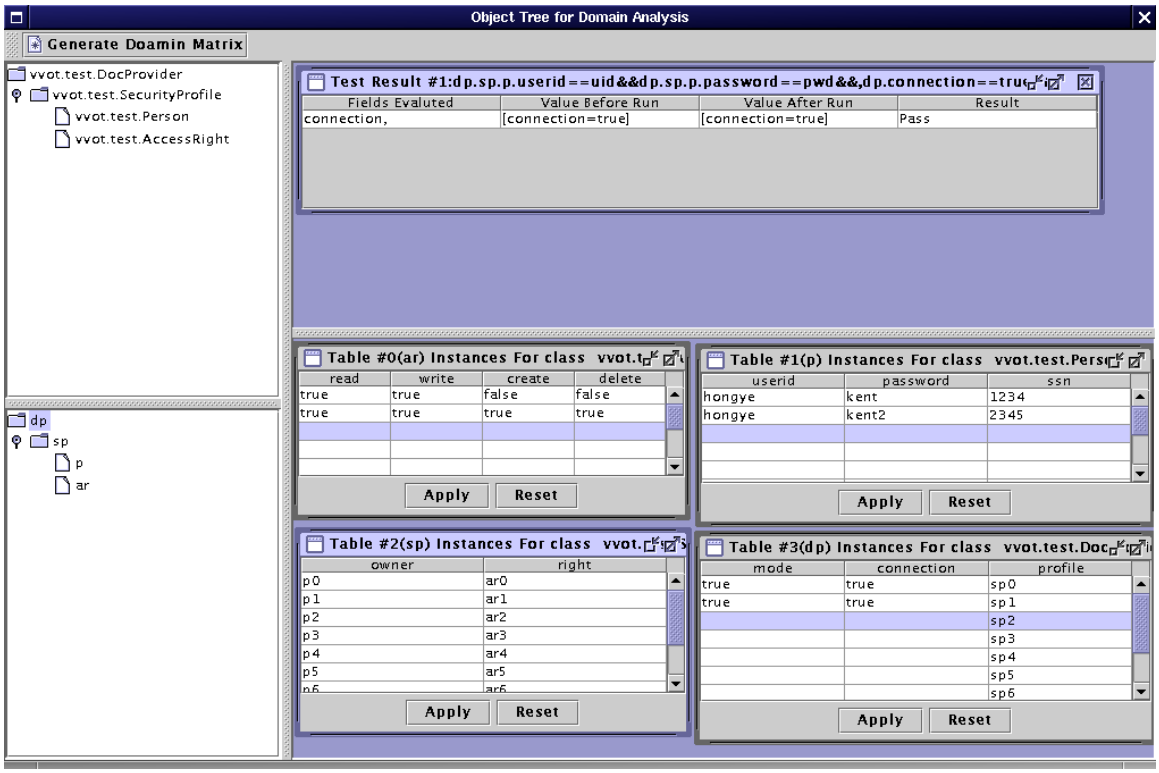

Figure 17: Test Case Generation Using the PrUDE Toolkit

The general approach to do so consists of writing test drivers or scripts. However, in our approach as mentioned earlier, the Java reflection mechanism is applied to directly modify and access object internal states. Also, this has been implemented in the PrUDE

toolkit for executing a Java program automatically. Figure 17 gives a snapshot of generated test data and test execution results for the test cases shown in Table 2 for method login.

6 Conclusion

Structured review is an effective way of finding certain types of deficiencies and bugs in programs, hence improving the level of dependability of software products significantly. We argue that the efficiency of structured reviews can be improved if combined with model-based and automated verification. On the other hand, formal verification techniques such as theorem-proving, supported with tools, are more practical than manual inspection for verification tasks related to well-formedness and validity checking. However, there are classes of correctness arguments such as optimality and robustness that cannot be fully mechanized, and for which structured review is more appropriate. This work builds on the strengths of the two techniques to develop an integrated development framework. We show how structured reviews and formal V&V can be combined effectively into a single development platform to complement one another. As argued in [2], the formalization process is one of the most time consuming aspect of using formal methods in system development. By defining formal semantics, and automatically generating formal specification from graphical semi-formal models, we considerably reduce the difficulties underlying the use of formal methods.

References

[1] I. Sommerville, Software Engineering, Addison-Wesley, 5th edition, 1996.

[2] S. Easterbrook, J. Callahan, and V. Wiels, "V&V through Inconsistency Tracking and Analysis," in the Proc. of International Work-shop on Software Specification and Design, Ise- Shima, Japan, April 16-18 1998.

[3] M. Lawford, P. Froebel, and G. Moum, "Practical Application of Functional and Relational
Methods for the Specification and Verification of Safety Critical Software," in the Proc. of Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000, T. Rus, Ed. 2000, vol. 1816 of Lecture Notes in Computer Science, pp. 73–88, Springer.

[4] OMG, "OMG Unified Modeling Language Specification, version 1.3," June 1999, OMG standard.

[5] I. Traor´e and D. B. Aredo, "Enhancing Structured Review with Model-based Verification," IEEE Transaction on Software Engineering (to appear), August 2004.

[6] D. B. Aredo, "A Framework for Semantics of UML Sequence Diagrams in PVS," Journal of Universal Computer Science (JUCS), Know-Center in cooperation with Springer Pub. Co., Joanneum Research and the IICM, Graz University of Technology, vol. 8, no. 7, pp. 674–697,
July 2002.

[7] I. Traor´e, "An Outline of PVS Semantics for UML Statecharts," Journal of Universal Computer Science (J. UCS), vol. 6, no. 11, pp. 1088–1108, 2000.

[8] M. Belaid and I. Traor´e, "The Precise UML Development Environment (PrUDE) Reference Guide," Tech. Rep. ECE01-2, Department of Electrical and Computer Eng., University of Victoria,

April 2001.
[9] S. Owre, N. Shankar, J. Rushby, and D. W. Stringer-Calvert, PVS Language Reference, version 2.3, Computer Science Laboratory, SRI International, Melon Park, CA, USA, September
1999.
[10] R. B. France, A. Evans, K. Lano and B. Rumpe, "The UML as a Formal Modeling Notation", Computer Standards & Interfaces, vol. 19, pp. 325–334, 1998.
[11] S. Owre, J. Rushby, N. Shankar, and F.V. Henke, "Formal Verification for Fault-tolerant Architectures: Prolegomena to the design of PVS," IEEE Trans. on Software Eng., vol. 21, no. 2, pp. 107–125, February 1995.
[12] J. B. Warmer and A. G. Kleppe, The Object Constraint Language: Precise Modeling with
UML, Addison Wesley Longman Inc., 1999.
[13] M. Y. Liu and I. Traor´e, "PVS Proof Pat-terns for UML-Based Verification," in the Proc. of ECBS workshop on Formal Specification of Computer-Based Systems (FSBCS02), Lund, Sweden, April 10-11, 2002.
[14] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," IEEE Trans. on Software Engineering vol. 22, no. 11, 1996.
[15] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 2nd edition, 1990.
[16] P. Kruchten, the Rational Unified Process, Addison Wesley, Sept. 1999.
[17] O. Laitenberger, C. Atkison, M. Schlich, and K. El Emam, "Using Inspection Technology in Object-oriented Development Projects," Technical report NRC/ERB-1077, NRC, Canada, June 2000.
[18] R. N. Britcher, "Using Inspections to Investigate Program Correctness," IEEE Computer,
November 1988.
[19] D. L. Parnas and D. M. Weiss, "Active Design Reviews: Principles and Practices," Journal of Systems and Softwares, pp. 259–265, 1987.
[20] I. Traor´e, "An Integrated V&V Environment for Critical Systems Development," in the Proc. of 5th IEEE International Symposium on Requirements Engineering, Toronto, Canada, August
2001.
[21] F. Keienburg and A. Rausch, "Using XML/XMI for Tool Supported Evolution of UML Models," in the Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Maui, Hawaii, January 3-6, 2001, IEEE Computer Society.

[22] M. D. Abrams, S. Jajodia, H. J. Podell, Information Security – An Integrated Collection of Essays, IEEE Computer Society Press, Los Alamitos, CA, 1995.
[23] C. Jones, Systematic Software Development using VDM, 2nd ed. Englewood Cliffs,NJ: Prentice-Hall, 1990.

Appendix

The PVS semantics generated for the statechart diagram in Figure 6 and the sequence diagram in Figure 4 are provided in the directory named *appendix*. The archive also contains the proof steps for the example properties checked in this report.