Integrating Formal V&V and Structured Design Reviews

Issa Traoré, Demissie B. Aredo

Abstract— **In the new Internet economy time-to-market has replaced quality as a primary concern in most software development organizations. Fortunately, that is not yet the case in organizations concerned with the development of critical systems. Most of these organizations consider design review as an efficient approach to improve the quality of their software products. In this paper, we discuss a set of correctness arguments that may be used in conjunction with formal validation and verification (V&V) in order to improve the quality of critical systems in a cost-effective way.**

Keywords— **Software inspection, Critical Systems, Formal Methods, UML, PVS, OCL**

I. Introduction

One of the major challenges that the software community has to face nowadays is to develop systems that provide a high level of quality at reasonable cost and time delay. The pressure to be the first in the market has drastically compressed the development process so that software products are often delivered without meeting the minimum quality assurance criteria, with vendors often relying on the patience and skills of customers to discover and report bugs. Though lower costs and rapid delivery seem to be the main issues in the contemporary marketing environment, meeting some level of quality assurance is still an important concern in highly competitive markets.

Software quality may improve significantly by integrating formal validation and verification (V&V) activities into the development process. V&V is a whole range of software analysis process that encompasses reviews of requirement and design, program inspection, and testing. Testing requires a prototype or an executable program code. In contrast, inspection may be used at all stages of the development process, especially at the earlier phases, where fixing errors is far more cheaper [6]. According to several studies in the literature, inspection can be more effective and cheaper error detection technique than testing [39], [20], [33]. However, inspection and testing shall be viewed as complementary V&V techniques. Inspection is good at checking conformance of a system with its specification, whereas testing appears to be a cost-effective technique for validation of dynamic behaviours.

The level of quality obtained with conventional V&V techniques may be insufficient for critical systems - where a failure may result in significant economic losses, physical damage, or threat to human life. Achieving a high level of dependability (i.e. availability, reliability, safety and security) is usually the most important quality criteria that must be met before launching the system. Although a better reliability can be achieved by using formal development techniques, the esoteric nature of formal methods, however, imposes a significant barrier on their large scale utilization [27]. As mentioned in [15],

Normal software developers will not, in the foreseeable future, be willing to use abstract formal languages and notations to design software systems, regardless of how theoretically desirable it might be to do so.

To overcome these barriers, several strategies for incorporating formal methods into software development process have been proposed in the literature [19], [1], [31]. Most of the strategies integrate the strengths of formal and semi-formal methods [22], [13], [42]. For instance, in [32] a visual formalism based on tables is used in the first place to write the specification. Then the verification is performed by generating automatically a PVS model based on the tables, and invoking the PVS theorem-prover tool.

The work reported in the sequel draws on the same principle by highlighting the major limitations of formal V&V and by compensating them with alternative strategies to facilitate its large scale utilization. We have developed a platform known as *Precise UML Development Environment (PrUDE)* [4], [46] that integrates formal methods with suitable existing graphical object-oriented notation(s). The graphical object-oriented notations are easy to learn and use, and in most cases they have industrial strength tool supports. PrUDE is based on four principles that are direct consequences of the argument of Evans *et al* quoted above. The first three principles are concerned with the following capabilities of the notations involved in the PrUDE platform:

1. a notation that can easily be grasped and used in an industrial context, and that has properties such as communicability and friendliness.
2. the ability to produce a formal specification which is amenable to rigorous analysis.
3. that it has efficient tool support, a prerequisite for large-scale application.

In order to achieve these objectives, a formal semantic for UML (Unified Modeling Language) [7] is defined using PVS (Prototype Verification System) [37]. In this way, the formal notation is hidden behind the graphical notation that is used by system developers as usual. At the same time, features of the formal notations are available for rigorous reasoning during the V&V process.

The fourth principle is motivated by the fact that there are several aspects of formal V&V that cannot be automated, or can only be automated partially with intensive user interactions, resulting in a complex and time-

I. Traoré is with Department of Electrical and Computer Engineering, University of Victoria, Canada. E-mail: itraore@ece.uvic.ca

D. B. Aredo is with Norwegian Computing Center, Oslo, Norway. E-mail:demissie.aredo@nr.no

consuming process. The idea promoted by PrUDE replaces these "dark sides" of formal V&V by informal designs complemented by systematic manual reviews. More specifically, formal design steps that cannot be automated are carried out using informal arguments such as informal correctness arguments which are recorded and challenged during a review process.

The rest of this paper is organized as follows. In Section II, we give a general overview of the PrUDE platform and discuss the role of inspection in this platform. In Section III, we introduce and discuss our inspection criteria. In Section IV, we demonstrate our approach through a case study of a security critical system. Finally, in Section V, we present some concluding remarks.

II. Overview of the PrUDE Platform

A. Foundation

The core notation used in the PrUDE platform is the UML [7]. The choice of UML was dictated by the fact that it provides an underlying notation for specification, a graphical notation which contributes to communicability and user friendliness. Moreover, UML is an international standard for object-oriented modeling, which is popular among industrial community and is supported by industrial-strength CASE-tools.

In spite of these features, UML lacks the semantic foundation necessary for precise specification and rigorous analysis of systems. The Object Constraint Language (OCL) [47] is an assertional language used in conjunction with UML notations. Although OCL complements the expressiveness of UML, its contribution in the context of rigorous reasoning is limited due to the lack of formal semantics. Several works on formalization of UML notations are available in the literature [17], [14], [30]. The formalization proposed in [14], for example, uses Z [41] as the underlying semantic foundation. In our case, we decided to use the PVS Specification Language as it provides a very general semantic foundation, and is supported by a powerful toolkit which integrates model-checker and proof-checker. Our previous works [2], [44], [3], focus on UML structural and behavioral diagrams, namely the class diagram, interaction diagram, and statechart diagram.

B. Automation

The PrUDE platform is automated by a tool suite consisting of an integrated V&V environment that supports consistency-checking, model-checking, proof-checking, and testing [4]. Model-checking and proof-checking are based on the PVS toolkit. The interface of PrUDE to the UML is based on XMI, which provides an explicit model exchange format for UML based tools. Since any tool support for UML is expected to export models in XMI format, the PrUDE platform is independent of any tool vendor. This makes it possible to easily adapt the platform to an existing software development environment. PrUDE's main strength is that it allows users to deal with models described in graphical notations that are user friendly, easy to learn and use. All formal specifications in PVS are processed at the back end. PrUDE also provides a specification-based testing component that consists of the current version of a test case generator and a test execution tool [45]. In the future, we extend the testing component by adding a test coverage analyzer. Test cases are generated from valid UML specifications.

C. V&V Strategy

The V&V strategy followed in the PrUDE platform is depicted by Figure 1. Typically, a designer develops his model using his favorite UML CASE-tool, and then submits the model to the PrUDE CASE-tool, which automatically generates formal semantic models in the PVS specification language. Ideally, the UML specification is accom-

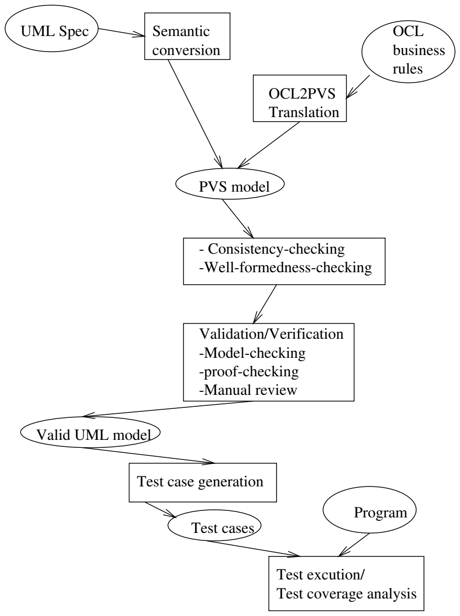

Fig. 1. V&V Strategy using the PrUDE Platform

panied by business rules (e.g. invariants, preconditions, post-conditions, system properties etc.) expressed in OCL. The OCL expressions are also translated into PVS and integrated with the semantic model [1]. The business rules may also be written and inserted directly using PVS. Then, the resulting model may be checked for well-formedness and consistency. Well-formedness is checked based on the rules defined in the informal semantic of UML constructs and abstract syntax [36]. A consistency check focuses mainly on the interrelationships among the various UML diagrams involved. In the next step, the model is checked against the business rules by invoking the PVS toolkit in batch mode. The business rules expressed as PVS conjectures and theorems are analysed using model-checking and/or proof-checking. Model-checking is conducted automatically. Proof-checking may be conducted automatically in batch mode for simple proofs. However, when the proofs

[1]In the current version of the PrUDE, the OCL expressions are converted into PVS manually. It is our intention to implement, in the future, an OCL to PVS translator for potential users of OCL.

reach a certain level of complexity, the proof-checker requires some user interaction. In that case, PrUDE offers the option to run the PVS proof-checker interactively. In principle, if an error is discovered, the analyst goes back to the OCL and/or UML models to fix the error.

Having a valid UML model, the designer may refine the model through subsequent steps and then implement the system. The program code may be tested with PrUDE using the UML specification. The valid UML model obtained after the series of V&V steps is used to generate test cases. The test cases are derived from various constraints related to the model, e.g. invariants, preconditions, postconditions etc.

D. The Role of a Reviewer

Most of the steps involved in the V&V strategy presented above can be carried out automatically. Unfortunately, some of the most error-prone and trickiest aspects cannot be automated and often rely on human guidance or ingenuity. The refinement and correctness-checking activities are among the most vital aspects involved. That is, where a strong review may be very helpful. In order to make formal V&V process more affordable, we advocate conducting of these steps using informal arguments. 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 procedure.

The role of the reviewer is not limited to these specific aspects, although they are the most critical in terms of review, since the influence of the human factor is higher. The reviewer may need to redo some of the other V&V activities that the designer may have already performed automatically. However, there is a difference between the analysis performed by the designer and the one performed by the reviewer. For instance, the designer may call the consistency-checker on unfinished design, whereas the reviewer will work only with a stable model [9]. During the design, a designer may decide to postpone consistency-checking, or not to resolve some of the inconsistencies at all. Sometimes it is better to find ways to live with inconsistencies, since a systematic removal may constrain unnecessarily the development process, or may result in new ones [24], [16]. During the review, the designer should be able to justify the need to keep specific inconsistencies, and the reviewer should be able to challenge the arguments of the designer.

In principle, a reviewer is not expected to spend excessive time on parts of exhibits that have already undergone automatic checks for obvious reasons. Ideally, the reviewer selects suitable samples of these parts of exhibits, e.g. the most critical components and focuses the inspection on them. For instance, the main limitation of test cases generated automatically in the PrUDE platform is that they are generated from a design model which is supposed to be formally validated. But, since there could possibly be some arrangements made during the refinement steps, by adding some informality, the validity of the test model would be overridden to some extent. So, the role of the reviewer will be to challenge the expressions, e.g. invariants, preconditions, post-conditions, that served for the design of the test cases, by analyzing selected test cases.

III. Design Reviews

A. Review Arguments

As we have already noticed, the most critical aspect of the review process in the PrUDE platform is concerned with parts of the exhibits which are directly related to the refinement process. There is a common belief that the traceability of an OO design to its analysis is straightforward. In practice, however, OO designs can get pretty far from the original analysis models, for instance by adding some design patterns or some mechanisms for decoupling or performance. It is important to be able to relate implementation or design elements to requirements. Generating these relationships tends to expose important mistakes, misconceptions, and omissions. By using formal methods, it is possible to establish these relationships precisely. A design step in that context consists of recording a series of assumptions about subsequent development and showing a correctness of the design step under the given assumptions [26]. The correctness of a design step is shown by discharging some proof obligations. Achieving that level of precision is rarely cost-effective, especially when the proof obligations involved are completely formal [11]. Moreover, the notion of formal proofs is outside the skills or experiences of an average software developer. Therefore, we advocate the use of informal (correctness) arguments in order to bridge the gap between specifications, designs and implementations. As we mentioned earlier, the role of a reviewer is to challenge these arguments in the light of the original requirements.

Our approach draws on the work performed by Britcher [9], where the key program attributes, namely *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 the inspectors and the author. The idea of the questionnaire follows the Active Design Review approach developed by Parnas [38].

In the sequel, we consider the following six correctness arguments that encompass and extend the criteria defined in [9]: *validity, traceability, optimality, robustness, well-formedness* and *consistency*. Though some of these arguments are overlapping, they provide a good coverage of the most important concerns raised w.r.t. design correctness.

Validity is concerned with the conformance of a specification to the customer requirements. In order to check the validity of a model, the analyst will typically draw some conjectures from the requirements and check these conjectures against the model. The conjectures may be stated either informally or formally in the form of proof obligations that need to be discharged.

Traceability consists of relating requirement and design specifications. Questions that should be answered by an inspector are targeted towards achieving two main goals: structural and behavioural conformances among the corresponding abstract and refined specifications. Structural conformance ensures the preservation of the static properties, e.g. operations and attributes, of the data object involved. Behavioural conformance ensures that the state structure, e.g. specifically sequences of events, accepted at an abstract level must be accepted at the corresponding concrete representation and leave it in the same state. In the case of UML, structural conformance is checked among involved classifiers such as classes, components, nodes, etc., whereas behavioural conformance is checked among state and interaction models such as statechart, sequence, and collaboration diagrams. For instance, a class defined in an abstract specification may be refined in one or more classes, possibly with new attributes and operations. However, it should be possible to establish clearly how the information defined in the abstract class is restated in the refined ones. In the case of a statechart, new states, events, or transitions may appear in the refined diagram, according to some well-defined rules, and it is the responsibility of the inspector to check that the rules are respected.

Optimality is concerned with appropriateness and efficiency of design choices. Choosing a representation of a data type can have considerable impact on the performance and scalability of the whole application. For instance, the choice of a concrete representation of a collection of items should consider whether or not a duplication is necessary, which specific operations are performed on the data involved, etc. If the application requires an efficient search mechanism in the collection, the choice must take into account not only efficiency of algorithms but also optimal use of storage, since wasted storage could considerably reduce the performance of the search algorithm. It is the responsibility of the inspector to establish the optimality of the choices made by the designer by analyzing the rationale behind the choices.

Robustness deals with how abnormal or exceptional situations are handled. That means, developers need to take into account unexpected behaviors a system may exhibit even if that was not explicitly required in the original requirement document. The responsibility of the analyst is to specify what should be done, and describe how it should be done during the design phase by sticking to the original requirements. However, a good designer should go far beyond the basic requirements and identify unexpected behaviors. That may serve as a base to renegotiate the requirements. Moreover, it may provide information needed by the implementor to handle these situations. Questions asked during the inspection will be drawn on the omissions and gaps in the design. For instance, in a statechart diagram, it is possible to highlight easily such kinds of omissions by considering non-specified transitions or sneak paths [5].

Well-formedness is mainly concerned with the correct utilization of the notations used to describe the design model. A model is said to be *well-formed* when the syntactic rules underlying the notation are all enforced.

Consistency is the broadest concept among the correctness arguments defined so far. Some of these arguments may fall easily in the consistency category. In fact, there are several kinds of inconsistencies, e.g. nine different kinds of inconsistencies are identified in [29]. In the sequel, we retain the interpretation of inconsistency provided in RM-ODP [25] as *a contradictory requirement involved in a software artifact or process.* Inconsistencies may arise from various sources. A development process is inconsistent if it involves contradictory activities; a software artifact is inconsistent if it contains contradictory requirements.

In the case of UML, three levels of inconsistencies are worth investigating: internal inconsistencies within a UML model, inconsistencies among UML models, and inconsistencies between UML abstract specifications and concrete design models. Some of these inconsistencies are captured by most of the existing UML CASE tools, but a few of them are not. The main reason behind that is the flexibility of the UML notations that the existing CASE tools are trying to preserve. Therefore, it is a responsibility of the designer, and a reviewer to ensure that there are no contradictions among the models, at least in their stable versions.

B. Review Procedure

The first step in the review process is a discovery of user requirements document 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 standard meeting that takes place at the beginning of reviews in order to present the system. The reviewer needs to make his own analysis in order to build an informed and independent opinion about the system under review. During the discovery phase, the reviewer will answer questions like:

1. What are the main business rules, the properties and invariants that characterize the system?
2. What are the significant scenarios underlying the system functionality?
3. What are the exceptional or abnormal conditions under which the system may function?

The goal of the initial discovery is to bring actively the reviewer in the review process, and at the same time to prevent his reasoning from being biased by the existing exhibits.

After the discovery phase, the reviewer starts the actual review by reading the exhibits. The first correctness argument that the reviewer may consider is the validity of the design. The following are among the questions that need to be answered during this step:

1. Do the exhibits provide a complete coverage of the business rules, the properties and invariants that characterize the system?
2. Do the exhibits derive naturally from the requirements?
3. Are the exhibits consistent with the requirements?

Having made his own assessment during the discovery phase, it may be easier for the reviewer to challenge the rules defined in the exhibits and to discover possible gaps, omissions or inconsistencies.

The next argument to be considered by the reviewer is traceability. Some of the questions that may be answered during this step are the following:

1. How many refinement levels are involved in the design?
2. Do the exhibits provide a complete traceability between the levels?
3. Are the relationships between abstract and concrete features adequately and consistently defined?

It is not sufficient to achieve traceability. The optimality of the refinement is important as well. The argument of optimality may be analysed by answering questions like:

1. Are the representations chosen during design refinement efficient with respect to the requirements?
2. Are there other alternatives and better solutions?

Consistency and well-formedness arguments can be analysed automatically. They may be analysed before or after the four other ones. The main goal of the reviewer will be to identify potential inconsistencies, and to discuss the consequences of keeping or removing them.

The last argument to be considered is robustness of the system. The following are some of the questions that could be raised during this step:

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 the exceptions and abnormal conditions?

A suitable process in which our review strategy may fit efficiently is the Rational Unified Process [28] which is used in conjunction with UML in most software organizations. The Rational Unified Process consists of an iterative and incremental development approach aimed at risks mitigation. At the end of every iteration, stable software artifacts that handle specific aspects and risks of the system are produced. Subsequent iterations are built on previous ones by assessing and revising corresponding risks. Our review strategy can be integrated at the end of each iteration and before starting the next one. The errors discovered by the reviewers should be fixed before starting the next iteration. Alternatively, the comments of the reviewers can also be included in the planning of the next iteration.

IV. Case Study

In order to illustrate our approach, we present an overview of a case study on a security critical system that provides a secured patient document service (PDS) [34].

A. Summary of User 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, integrity, confidentiality, and authorization are always preserved.

B. Overview of UML Business Model

Some selected properties of the system are discussed below. Static structural aspects are modeled by class diagrams, whereas dynamic behavioral aspects are described by statechart diagrams. The class diagram provided in

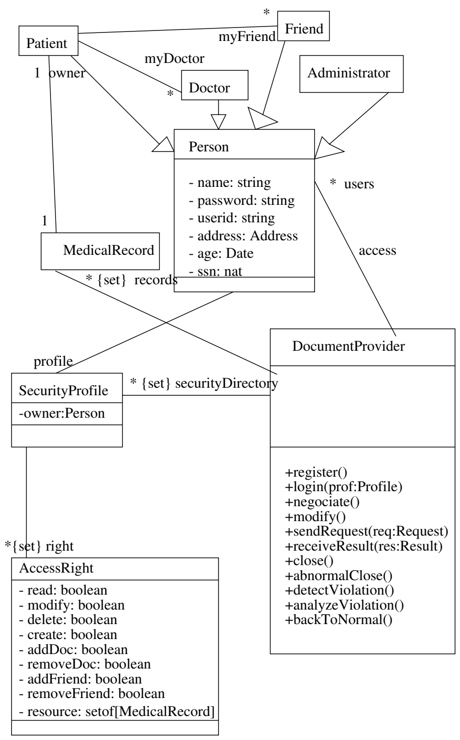

Fig. 2. Class Diagram of the Patient Document Service

Figure 2 depicts structural components of the system described above. The potential users of the system are represented by the *Patient*, *Doctor*, *Administrator* and *Friend* classes. These classes are subclasses of the *Person* class that describes a set of common attributes. The *Document-Provider* 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.

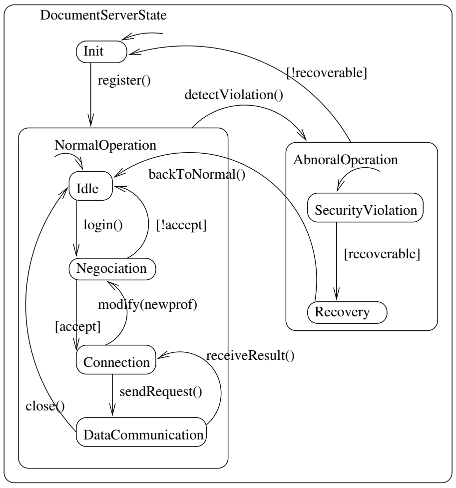

Fig. 3. State Diagram of class DocumentProvider

The statechart diagram shown in Figure 3 describes dynamic behaviors of the *DocumentProvider* class. 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.

B.1 Business rules

The UML business model needs to be augmented by a set of business rules stated in OCL. We discuss some examples of business rules below.

Rule 1: A patient cannot create, delete or modify his own medical records.

context Patient
inv self.profile.right → forAll(r | not(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 he has been chosen by another patient 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))

C. Reviews

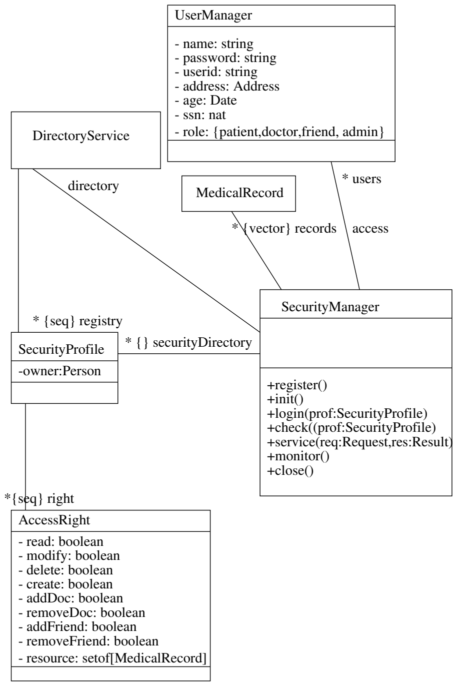

Fig. 4. Design Diagram of the Patient Document Service

As we already mentioned, *well-formedness* and *consistency* arguments may be checked automatically using the PrUDE toolkit. This is performed after PVS semantic model corresponding to the UML model is generated. For instance, Figure 5 shows a PVS model generated from the UML statechart diagram shown in Figure 3 using the PrUDE toolkit.

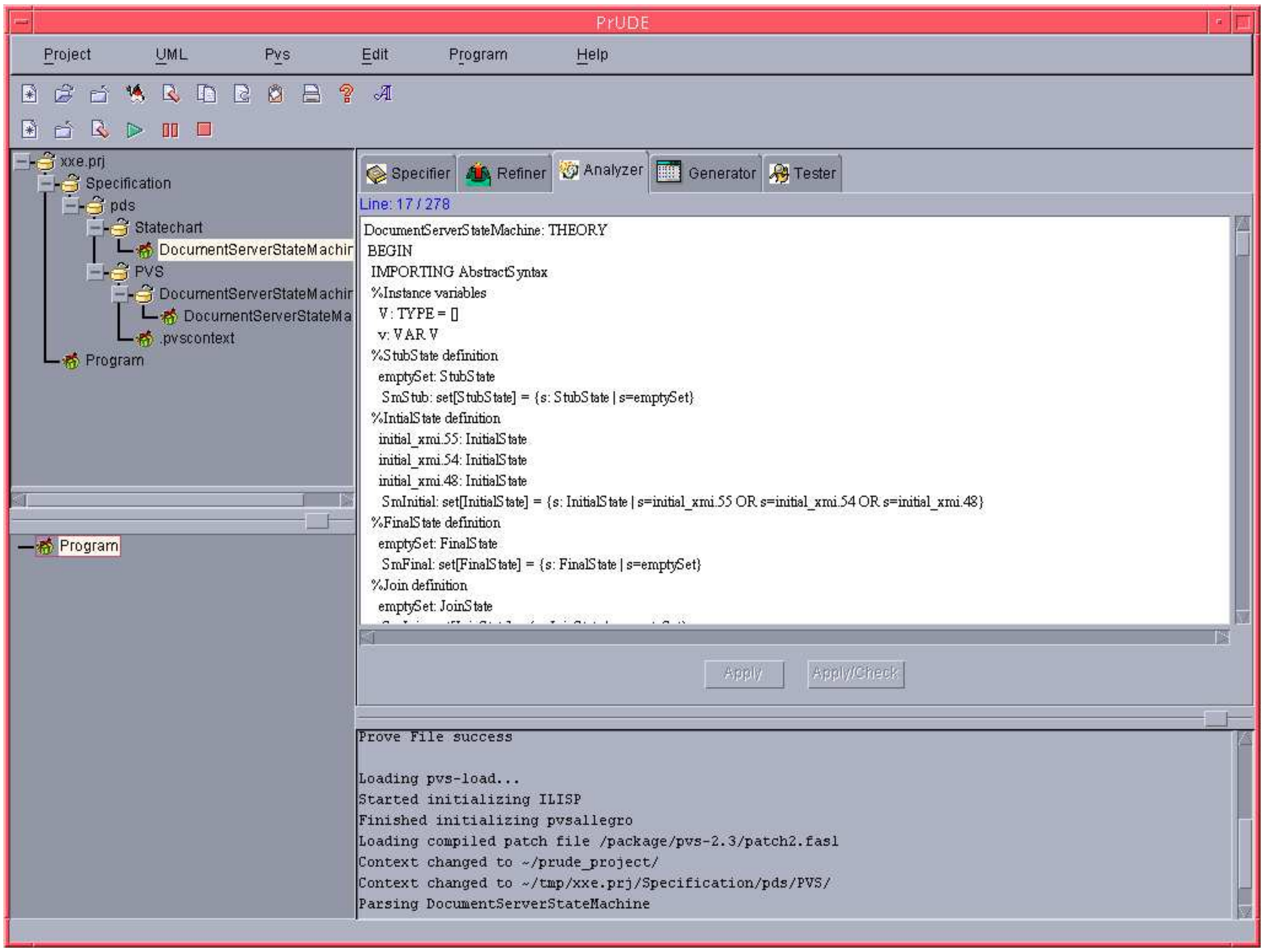

Fig. 5. PVS Semantic Generated Using the PruDE Toolkit

The lower window is a log area that shows the report from the PVS theorem-prover that is started in batch mode in order to check the business rules. The remaining arguments are checked manually. In the rest of this section, we show, by examples, how that 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 business model provided in Figure 2 is refined into a new design model given in Figure 4. Instead of having several classes for different users of the system, e.g. *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* [2] class is a new class that performs all necessary security checks before executing a request. There is also a standard directory service represented by the *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 *DocumentProvider* 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: $retr : Rep \rightarrow Abs$, where Abs is the abstraction and Rep is a representation. For instance, for the *SecurityManager* class, the following retrieve function can be defined:

$retr : SecurityManager \rightarrow DocumentProvider$
context DocumentProvider
sm: SecurityManager
inv self = retr(sm) **implies**
(self.records = retr(sm.records) **and**
self.securityDirectory =
retr(sm.securityDirectory) **and**
self.users = retr(sm.users))

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 the data types involved are modified. For instance, the above retrieve function establishes correspondence between the *records* attributes in, respectively, the *DocumentProvider* 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*, e.g. an array. The retrieve function for the attribute *records* may be defined in this case as follows:

$$\text{retr(sm.records)} = \{sm.records[i] \mid 0 \leq i < sm.records.size\}$$

[2]This has nothing to do with the standard SecurityManager class provided in the Java security API.

The abstract attribute *records* is 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 DocumentProvider
inv self → forAll(dp | (SecurityManager →
exists(sm | retr(sm.records)
= dp.records)))

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

Given any finite set, it is always possible to arrange its elements 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 [26]. 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? The review may also establish that the design is not *valid*, because it fails 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 patients may choose, as family or "guest" doctor, any person who is a registered doctor, except themselves. An additional business rule may be stated as follows:

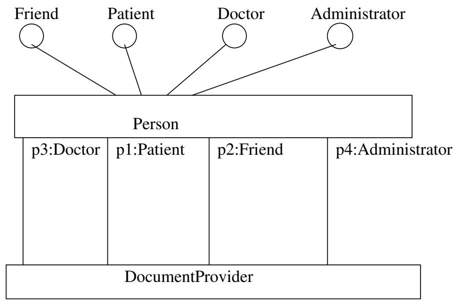

Fig. 6. Dynamic Reconfiguration in the Patient Document Service

Rule 6:

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 (cf. Figure 6). The solution adopted in Figure 6 describes the different roles a *Person* may play, as 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. A discussion on how dynamic reconfiguration can be described using UML can be found in [43].

The robustness issue raised during the review process may be 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, analyse and recover from security violations. This concern is already handled by the statechart diagram given in Figure 3 by specifying appropriate recovery mechanisms.

V. Concluding Remarks

Though inspection can be quite effective in finding deficiencies and bugs in programs, it should not be considered as a replacement for other formal V&V techniques such as testing and formal reasoning. For instance, testing is more practical than inspection for verification tasks related to system integration, performance analysis, reliability assessment or user interface validation. Formal reasoning may significantly improve the level of precision and rigor of a software product. But both testing and formal reasoning may involve high costs. This work builds on the strengths of such technique to develop an efficient and cost-effective integrated V&V framework. We show how design review can be used effectively as a replacement for the trickiest phases of formal V&V. Though this work focusses mainly on design reviews, the overall framework encompasses various V&V techniques such as testing and formal reasoning. The PrUDE platform provides a CASE-tool aimed at development and rigorous V&V of critical systems. Unfortunately, not all steps of software V&V can be automated. Several complex steps rely on human guidance and ingenuity. The approach presented in this paper can help in improving the confidence in V&V of critical systems by bridging the gap by using systematic manual reviews based on selected correctness criteria.

Currently, we are also investigating additional testing strategies in order to strengthen our framework for testing. We are exploring how test cases can be generated using model-checking based on UML statecharts. We observed that one of the most complex aspects of the design phase is the underlying refinement process. We believe that an integration of a thorough review into refinement process is urgently needed. Our future work will focus on identifying more systematic mechanisms involved in the UML refinement process. That may help in increasing the level of automation of the refinement process, and consequently, the quality of software products.

Several works have been performed on using correctness arguments in design reviews. Closely related to our work is the work of Parnas and Weiss [38] on *active design review* (ADR). The ADR method is guided by questionnaires provided to the reviewers by the authors. The questions are designed in such a way that they can only be answered after a careful review. The goal of the questionnaire is to force the reviewer to play a more active role in the review process than just reading the exhibits passively. Based on the ideas of the questionnaire, Britcher [9] later proposed an approach that combines the strength of formal correctness arguments with informal review. Four correctness arguments namely *algebra, topology, invariance* and *topology* are examined using a questionnaire based on the ADR method. In our case, we define additional arguments that broaden the scope of the review process, and thereby increase the number of potential defects that may be uncovered. In contrast to ours, the *cleanroom* process developed at the IBM put a strong emphasis on formal proof-checking, which is used as an alternative to unit testing [35]. The goal of the cleanroom process is to achieve zero-defect software by using rigorous inspection processes. The software is developed and validated incrementally through successive refinement steps. The software code obtained at the end is verified using rigorous software inspections which replace unit testing. After integrating the increments, reliability of the overall system is tested using statistical tests. The stepwise refinement that contributes significantly towards the efficiency of the cleanroom process is a source of its main weaknesses, because of the inherent complexity of formal verification. A successful application of the approach may require skilled and committed developers as reported in [40].

References

[1] M. Archer, C. Heitmeyer, S. Sims, *TAME: A PVS Interface to Simplify Proofs for Automata Models*, In Proc. User Interfaces for Theorem Provers, Eindhoven, Netherlands, Eindhoven Univ. Technical report, Eindhoven Univ. of Technology, July 1998.

[2] D. B. Aredo, I. Traoré, K. Stølen, *An Outline of PVS Semantics for UML Class Diagrams*, In Proc. of 11^{th} Nordic Workshop on Programming Theory (NWPT'99), Oct. 6-8, 1999, Upsala, Swedden.

[3] D. B. Aredo *Semantics of UML Sequence Diagrams in PVS*, In the Proceedings of the Workshop on Dynamic Behavior in UML Models, at UML2000, October 2-6, 2000, York, UK

[4] M. Belaid, I. Traoré, *The Precise UML Development Environment Reference Guide*, Technical Report N0 ECE01-2, Department of Electrical and Computer Engineering, University of Victoria, April 2001.

[5] R. V. Binder, *Testing Object-oriented System: Models, Patterns and Tools*, Reading, MA:Addison-Wesley Longman, 1999.

[6] B. Boehm, *Industrial Software Metrics Top 10 List*, IEEE Software, 4(5):84-85, September 1987.

[7] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman Inc, Reading Massachusetts 01867, 1999.

[8] H. Bowman, J. Derrick, M.W.A. Steen, *Viewpoint Consistency in ODP, a general interpretation*, In E. Najm and J.-B. Stefani, editors, 1st IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems, pages 189-204. Chapman & Hall, March 1996.

[9] R. N. Britcher, *Using Inspections to Investigate Program Correctness*, IEEE Computer, Nov. 1988.

[10] V. Cassigneul, *How to Control the Increase in Complexity of Civil Aircraft On-board Systems*, AEROSPATIALE Aircraft, Internal Report, Toulouse, France, 1994.

[11] D. F. D'Souza, A.C. Wills, *Objects, Components and Frameworks with UML-The Catalysis Approach*, Addison-Wesley Object Technology Series, 1999.

[12] S. Easterbrook, J. Callahan, V. Wiels, *V & V through Inconsistency tracking and Analysis*, International Workshop on Software specification and Design, April 16-18 1998, Ise-Shima, Japan.

[13] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, D. Hamilton, *Experiences Using Lightweight Formal Methods for Requirements Modeling*, IEEE Trans. on Soft. Eng., Jan. 1998, Vpl. 24, 4-14.

[14] A. Evans, *UML class diagrams - filling the semantic gap (draft)*, Technical Report, York University, 1998.

[15] A. Evans (moderator), S. Cook, S. Mellor, J. Warmer, A. Wills, *Advanced Methods and Tools for a Precise UML*, In the Proc. of 2nd International Conference on the Unified Modeling Language. Editors: B.Rumpe and R.B.France, Colorado, LNCS 1723, 1999.

[16] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, *Inconsistency Handling in Multi-Perspectives Specifications*, In the Proc. of 4th European Software Engineering Conference (ESEC'93), Garmisch-Partenkirchen, Germany, September 1993, 84-99, LNCS 717, Springer-Verlag.

[17] R. B. France, A. Evans, B. Rumpe, *The UML as a Formal Modeling Notation*, Computer Standards & Interfaces, 19 (1998), P. 325-334.

[18] M. D. Fraser, K. Kunar, V. K. Vaishnavi, *Informal and Formal Requirements Specification Languages: Bridging the Gap*, IEEE Trans. on Soft. Eng.,Vol. 18, NO. 17, May 1991, P454-466.

[19] M. D. Fraser, K. Kunar, V. K. Vaishnavi, *Strategies for Incorporating Formal Specification in Software Development*, Oct. 94, Vol 37, No. 10,Communications of ACM, p 74-86.

[20] T. Gilb, D. Graham, *Software Inspection*, Workingham: Addison-Wesley, 1993.

[21] J. Grundy, J. Hosking, W. B. Mugridge, *Inconsistency Management for Multiple-View Software Development Environments*, IEEE Trans. On Soft. Eng., Vol. 24, No. 10, Nov. 1998.

[22] M. P. E. Heimdahl and N. Leveson, *Completeness and Consistency in Hierarchical State-Based Requirements*, IEEE Trans. On Soft. Eng., Vol. 22, P. 363-377, 1996.

[23] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, *Automated Consitency Checking of Requirements Specifications*, ACM Trans. on Soft. Eng. and Meth., Vol. 5, No. 3, July 1996, P. 231-261.

[24] A. Hunter, B. Nuseibeh, *Analyzing Inconsistent Specifications*, Proc. RE'97, 3rd Int'l Symp. Req. Eng., P. 78-86, Annapolis, Md., 1997.

[25] ISO-IEC JTC1/SC21/WG7, *The Reference Model of Open Distributed Processing*, http://www-cs.open.ac.uk/ m_newton/odissey/RMODP.html

[26] C. B. Jones, *Systematic Software Development using VDM*, 2d ed.,Prentice-Hall, Englewood Cliffs,NJ,1990.

[27] R. Kneuper, *Limits of Formal Methods*, Formal Aspects of Computing (1997) 9: 379-394.

[28] P. Krutchen, *The Rational Unified Process*, Addison Wesley, Sept. 1999.

[29] A. V. Lamsweerde, R. Darimont, E. Letier, *Managing Conflicts in Goal-Driven Requirements Engineering*, IEEE Trans. On soft. Eng., Vol. 24, No. 10, Nov. 1998.

[30] D. Latella, I. Majzik, M. Massink, *Towards a formal Operational Semantics of UML Statechart Diagrams*, Proc. FMOOD'99, Feb. 1999, Florence, Italy.

[31] M. Lawford, J. McDougall, P. froebel, G. Moum, *Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software*, In Proc. Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, USA, May 2000.

[32] M. Lawford, P. Froebel, G. Moum, *Application of Tabular Methods to the Specification and verification of a Nuclear Reactor Shutdown System*, Submitted to Formal Methods in System Design, August, 2000.

[33] R. C. Linger, *Cleanroom Process Model*, IEEE Software, March 1994.

[34] M.Y. Liu, H. Ye, I. Traoré, *Using formal methods in Security Engineering: case Study of a Patient Document Service*, technical Report No. ECE01-3, Department of Electrical and Computer Engineering, University of Victoria, May 2001.

[35] H. D. Mills and M. Dyer, Cleanroom software engineering, IEEE Software, vol. 4, no. 5, pp. 19-25, 1987.

[36] Object Management Group, *OMG Unified Modeling Language Specification, Version 1.3*, OMG standard document, June 1999.

[37] S. Owre, N. Shankar, J. Rushby, D. W. Stringer-Calvert, *PVS Language Reference, version 2.3*, Computer Science Laboratory, SRI International, Melon Park, CA, September 1999.

[38] D. L. Parnas, D. M. Weiss, *Active Design Reviews: Principles and Practices*, Journal of Systems and Softwares 7, 259-265 (1987).

[39] R. W. Selby, V. R. Basili, *Cleanroom software development: an empirical evaluation*, IEEE trans. on Sof. Eng., SE-13990, 1027-37.

[40] I. Sommerville, *Software Engineering*, 6th Ed. Addison-Wesley, 2001.

[41] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd Ed. London: Prentice Hall.

[42] I. Traoré, A. Jeffroy, M. Romdhani, A.E.K. Sahraoui, *An Experience with a Multiformalism Specification of an Avionics System*, in Proc. INCOSE 98, July 25-31 1998, Vancouver, Canada.

[43] I. Traoré, D. Aredo, K. Stolen, *Formal Development of Open Distributed Systems: towards an Integrated Framework*, OOSDS Workshop, PLI Conference, Paris, Sept. 1999.

[44] I. Traoré, *An Outline of PVS Semantics for UML Statechart*, Journal of Universal Computer Science, Springer Pub. Co., Nov. 2000.

[45] I. Traoré, *A Framework for Rigorous Testing of Object-oriented Programs*, ECBS Conference, Workshop on Formal Specification of Computer- Based Systems (FSBCS01), April 2001, Washington D.C., USA.

[46] I. Traoré, *An Integrated V&V Environment for Critical Systems Development*, to be published in the Proc. of 5th IEEE International Symposium on Requirements Engineering (to be held), August 2001, Toronto, Canada.

[47] J. Warmer, A. kleppe, *The Object Constraint Language: precise Modeling with UML*, Addison-Wesley, 1999.