University of Oslo
Department of Informatics

Formal System Development Using Method Integration: a Case Study

Demissie B. Aredo and Olaf Owe

Research Report 308

August 2004

Formal System Development Using Method Integration: a Case Study

Demissie B. Aredo[1] and Olaf Owe[2]

[1]Norwegian Computing Center
P. O. Box 114 Blidern, N-0314 Oslo, Norway.

[2]Department of Informatics, University of Oslo
P. O. Box 1080 Blidern, N-0316 Oslo, Norway.

Abstract

In this paper, we demonstrate feasibility of a development framework that integrates semi-formal graphical modeling techniques with formal methods (FMs). In particular, the framework integrates the Unified Modeling Language (UML) with the PVS environment to exploit the synergy between them. System descriptions are given in the graphical UML notations and translated into PVS specifications based on semantic definitions, which we have proposed for the UML notations. The resulting semantic models are rigorously analyzed using the PVS toolkit. The translation of UML models into PVS specifications is automated by the PrUDE tool. This work contributes towards the improvement of the use of FMs in the development of highly dependable systems in industrial settings and narrows the gap between the theoretical foundation underlying FMs and their practical application.

Keywords: Formal Methods, UML, OCL, OUN, PVS, Method Integration

1 Introduction

Semi-formal object-oriented analysis and design (OOAD) techniques such as the UML (Unified Modeling Language) [28] have become quite popular among software developers. The structuring mechanisms, and intuitively appealing graphical notations are among the features that have contributed to their acceptance. Their major limitation in the context of critical systems development is, however, the lack of precise semantic definitions for their notations - a significant barrier to their application to critical system development in industrial settings. A greatly improved development process can be obtained if tools are augmented with deeper semantic analysis of the graphical models [45].

On the other hand, formal methods (FMs) [46] have enormous potential in the development of highly dependable systems, and are increasingly finding practical uses due to recent development towards automated tools. FMs are development approaches based on a mathematical foundation allowing precise and rigorous specification of system requirements, and ensure that the final software product meets the initial expectations of the customer in terms of functionality as well as quality. Despite the rigor, practical usability of formal verification approaches is limited due to their esoteric nature. A framework that integrates a semi-formal modeling language, namely the UML, and a formal verification environment, namely the PVS, and a supporting tool is the focus of this paper.

The main objective of formal development methods is to specify system behavior and desired functionalities precisely, and verify that the system meets the original requirements. Formal specification is a basis of a meaningful and rigorous analysis of system properties. Some verification environments provide specification languages tailored towards a specific application domain together with a simulator, a model checker or both, e.g. LOTOS [18], and the SPIN system [16]. Due to features inherent in distributed systems, e.g. concurrency, dynamic reconfiguration, and complexity, a simulation can examine only a fraction of possible system runs. Techniques related to model checking, on the other hand, provide complete exploration of all possible runs exhibited by a finite-state machine describing the system. Model checking has become very popular because experiences indicate that checking all runs is more effective in finding bugs [35] while requiring little or no insight in the formalism, and no user interaction is required. Model checking can also be complemented with interactive proof-checking if necessary. A major limitation of model checking is that the state space must be finite even though advances involving symbolic execution have been made.

The benefits of introducing FMs into a development process includes:

- Improved understanding of system requirements and reduced errors and omissions;

- Possibility to check consistency and completeness of system specifications, and prove that an implementation conforms with the specifications;

- Semantically-based CASE tools can be built to assist developers in analysis, design, implementation and program debugging. They may also support animation and execution of formal specifications to provide a prototype of the system; and

- Formal specifications are used as guidelines in the identification of appropriate test cases and their evaluation.

Despite all these benefits, FMs still have difficulties in breaking through the software industry. Very few organizations or projects are using FMs. A number of reasons have been put forward as to why the formal development methods have not been widely used in the software industry [36]:

- FMs are considered esoteric, due to the lack of training for software engineers in the discrete mathematics and logic at the required level. Moreover, customers are unlikely to be familiar with FMs, and hence they are not willing to pay for the development activities they cannot monitor; and

- Lack of tool support: most of the effort in research on formal methods focused on the development of languages and their mathematical underpinning and less effort has been devoted to tool support.

As argued by Sommerville [36], the major challenge facing the software community is not developing new techniques and methods, but transferring the existing software engineering research results into the software industry. To address this issue, a number of strategies for introducing FMs into software development process have been proposed by the research community. Most of the strategies [11, 24, 42] advocate a lightweight and selective application of FMs using visual modeling notations such as the UML [28] as a front-end. FMs are used solely for analyzing specific aspects or properties of a system. The baseline specification used to conduct further development activities, is created and maintained using the graphical notations familiar to and popular among software developers. In [41, 39], we proposed a development framework integrating the UML specification techniques [28, 34] with the Prototype Verification System (PVS) [30] to support formal development of distributed systems. The integrated approach has the following major contributions to the software engineering process:

- A formal specification of syntactic well-formedness constraints for UML in the PVS specification language, which significantly improves the acceptance of FMs among software developers by enhancing the development process with OOAD techniques, and supported by a CASE tool.

- Defining formal semantics of graphical modeling language addresses the limitations of OOAD techniques in the context of the development of highly dependable systems by making UML models amenable to formal analysis.

In the sequel, we demonstrate practical usability of the integrated approach by presenting an example of a security-critical system. Major components and concepts of the framework and a supporting CASE tool are revisited to make this paper self contained.

1.1 Outline of the Report

The rest of the report is outlined as follows. In Section 2, major aspects of the development framework and the supporting CASE tool, namely, the PrUDE tool are briefly revisited in order to make the report self-contained. Our focus is mainly on concepts and notations that might be encountered in later sections. In Section 3, we demonstrate practical usability of the integrated platform and the supporting tool by presenting an example of the development of a security-critical system. Finally, in Section 4, we summarize, draw some conclusions and discuss future research issues.

2 The Integrated Platform Revisited

The development of critical systems such as the e-banking, and access control systems requires high-level of rigor and reliability. Integrating formal methods (FMs) into a software development process improves software quality and reliability by revealing subtle errors that may not be, otherwise, discovered before it is too late and too expensive to fix. It also increases productivity by supporting development of semantically based tools.

Usually, developers describe different aspects of a system, using several description techniques and notations. For instance, one might want to describe the functional behavior of a system as a composition of the functional behaviors of the modules constituting the system. Moreover, one might want to specify structural relationships between the modules, e.g. modules that may directly communicate. At the time of this writing, there is no single description technique or notation that conveniently can capture complete behavior of a system from different view points, and at the level of rigor necessary for reasoning about reliable systems. Hence, integrating several specification techniques, notations, and formalisms is necessary.

When several description techniques and notations are involved in a development platform, using a common underlying semantic domain is very essential. This significantly reduces the effort to check consistency across language boundaries, by allowing reasoning about system properties in a uniform manner. As mentioned in the previous section, when it comes to practical applications, both the semiformal OOAD techniques and the FMs have inherent strengths and limitations. We argue that a development platform that pulls together strengths of FMs and OO graphical modeling technique significantly improves the reliability of critical systems. The main objective of method integration approach is to obtain a development framework and a supporting tool that enhance application of FMs in an industrial setting, and at the same time make the OOAD techniques amenable to rigorous analysis.

2.1 Notations and Formalisms

In the rest of this section, we present a brief overview of the notations and formalisms involved in the integrated platform. We do not present a complete tutorial on the notations, instead we focus only on key features that will be encountered in later sections. For detailed presentations, interested readers should refer to respective relevant literatures.

2.1.1 The Unified Modeling Language

The Unified Modeling Language (UML) [28, 34] provides a set of standard notations and modeling techniques for specifying, visualizing, and documenting artifacts of software systems. UML supports a highly iterative, distributed software development process, where every stage of the software life cycle, e.g. requirement analysis, and design, can be specified by using a combination of different description techniques. Our work is based on UML 1.3.

At the time of this writing, there is no standard formal semantics for UML notations, and this makes development of semantically-based CASE tools a difficult task. Most tool vendors use in-house semantic definitions for UML notations. In the UML standard [28] a semi-formal semantic guideline is provided for developers of UML tools.

Static structural system properties can be specified by UML diagrams such as class, and component diagram, whereas dynamic properties can be captured by diagrams such as the interaction diagrams, statecharts, and activity diagrams. An interaction is specified by a sequence diagram consisting of a list messages exchanged between the interacting objects involved in the interaction.

A sequence diagram is a particular type of diagram describing a specific pattern of interaction between objects in terms of messages exchanged as the interaction unfolds over time to effect the desired property. A message is a specification of a communication between objects, or an object and its environment, conveying information with the expectation that an activity will ensue. A sequence diagram specifies roles of the objects, i.e. sender or receiver, as well as the associated action that causes the communication to take place. However, it conveys a possible behavior rather than restricting all possible behaviors. UML sequence diagrams are efficient description technique for describing scenarios of systems with time-dependent functionality, like real-time applications. The simplicity of sequence diagrams makes them suitable for specification of intended behavior that can easily be understood by every stakeholder: customers, requirements engineers, and software developers alike [45].

We are interested only in externally visible properties of objects and ignore internal changes. We distinguish between *send* and *receive* events associated with each message when modeling the behavior of objects participating in the interaction specified by a sequence diagram. Hence, in a specification of a message, correspondence between the send and receive events constituting the message has to be established. In our framework, a message is interpreted as a pair of *send* and *receive* events. Hence, a sequence diagram is interpreted as a set of *traces* of events satisfying some specific properties, such as the *causality* and the general *ordering* requirements [3].

UML supports the notion of time (see [28, chap. 3, pp. 98]) and allows specification of the time when a message is sent and received. The notion of time can be captured by stamping events by the time of their occurrences. This sort of information is useful for expressing temporal properties of traces, e.g. the minimum time interval between the occurrences of two events. Stamping of events with global time is crucial, for example, to obtain the global history by merging traces of events by interleaving the events in temporal order of their occurrences. The resulting trace is a specification of the global history of the object under consideration.

An object participating in an interaction is represented as a set of infinite and finite traces reflecting, respectively, non-terminating and terminating executions. For safety properties, finite trace semantics is sufficient to specify behavior of a system over a finite time interval. Hence, we define the semantics of a sequence diagram as a prefix-closed set of finite traces, and represented in the PVS-SL as sets of *lists* of events.

2.1.2 The Object Constraint Language

The abstract syntax of UML constructs is given in terms of UML meta-models, using UML class diagrams enhanced with textual annotations. The graphical UML models are not expressive enough for precise and unambiguous specifications. There is a need for description of additional constraints on objects in UML models.

In the UML standard [28], constraints on modeling elements are given as a set of well-formedness rules expressed in the Object Constraint Language (OCL) [44] complementing the English language. OCL is a specification language extension to the UML notation provided as a part of the UML standard since UML v1.3 [28]. OCL is an expression language that enables developers to formulate constraints and object queries in the context of UML models. OCL expressions are used to specify invariants attached to static structural elements such as classes and types, pre- and post-condition of operations and guards for state transitions.

OCL is a declarative language, not a programming language, i.e. evaluation of OCL expressions does not have side-effects on the associated UML model. Consequently, it is not possible to write program logic or control-flow in OCL, or invoke processes or activate non-query operations within OCL. As a modelling language, all implementation issues, except their correctness, are out of the scope of OCL. Hence, unlike some other formal languages such as Z [37], OCL specifications (specially invariants) are not easily convertible into program code. However, in the development of larger systems heed to the implementation is needed as it would not be feasible to back off in the middle of the development and start coding from the scratch. A number of tools for parsing and checking syntax of OCL specifications are available, e.g. OCL tool [27] developed at the Dresden University of Technology, and Octopus [26] developed by Klasse Objecten.

To integrate constraints into UML models, invariants, and pre- and postcondition are attached as comments to respective modeling elements. Constraints may, however, turn out to be quite complex, with the impact that they are often specified separately. The contextual modeling element is explicitly specified by the *context* clause.

OCL is a typed language based on the first-order logic. Logical operators and universal quantifiers in the first-order logic, and set operations lead to a powerful expressive language. Besides user-defined model types (e.g. classes, interfaces) and predefined basic types (e.g. integer, real, boolean), OCL has the notion of object collection types (e.g. sets, bags, sequences). Several operations such as the arrow operation → are predefined on the object collection types. For example, consider the

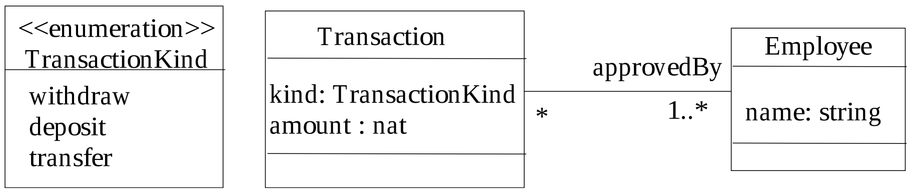

Figure 1: Partial Description of a UML Class Diagram

partial description of a UML class diagram shown in Figure 1. The `Transaction` and `Employee` classes are related by an association with one association end called `approvedBy`. The following OCL expression specifies that each transaction of kind *withdraw* or *transfer* involving an amount of funds above $10000 must be approved by at least two employees.

```
context Transaction inv:
 (self.kind = withdraw OR self.kind = transfer) AND self.amount > 10000
                    implies self.approvedBy->size ≥ 2
```

Let us briefly explain the parts of the above OCL expression. The class name following the keyword *context* specifies the class for which the invariant is defined. The keyword *inv* indicates that this expression is a specification of an invariant, i.e. the expression must always evaluate to *true* for each object of the context class. But, an invariant can be violated during an execution of an operation. In other words, an invariant must hold for an object when none of its operations is executing.

The keyword *self* is optional and refers to the object for which the expression is evaluated. Attributes, operations, and associations of the object can be accessed by dot notation, e.g. `self.approvedBy` results in a set of objects of class `Employee` associated with the `Transaction` object for which the invariant is currently evaluated.

The arrow notation (→) indicates that the collection of objects proceeding the arrow is manipulated by a predefined OCL operation following the arrow. For example, for a given collection `c`, the expression `c→size()` returns the number of elements in the collection.

There is a point to be made about constraints and inheritance in object-oriented models. In object-orientation, it is a rule that classes at the lower level of an inheritance hierarchy are always more specialized and concrete than the abstract classes at the higher level. This principle continues to hold for constraints, in that a subclass may strengthen constraints inherited from its superclass. In other words, a subclass inherit constraints from its super class, and may have additional constraints. This may cause problems where classes are freely reused.

Constraints are specification of conditions that should not be violated. But, OCL v1.0 does not describe the measure to be taken in case a constraint is violated. As OCL is an expression language, one may argue that action does not need to be taken, and the model will be in an invalid state. Kleppe *et al* [23], however, proposed an extension of OCL by *action* clauses. The *action* semantics and object query language definitions are among the main feature added to OCL v2.0 that is a part of UML v2.0.

Semantics of OCL expressions are described informally in the standard document [28]. Richters *et al* [33] proposed a formal semantics for the OCL constructs. Several extensions of OCL are proposed in the literature. Flake *et al* [12] propose temporal extension of OCL that enables developers to specify behavioral state-oriented constraints and present a formal semantics of state-oriented constraints [13].

We have given a brief summary of basic concepts of OCL used in later sections, and refer interested reader to the latest proposal of OCL 2.0 language definition [43] for more details.

2.1.3 Motivation for Creating a more Expressive Language

The main goal of the ADAPT-FT project is to develop a platform supporting precise modeling of systems that are distributed, object oriented, and open. We wished to address high level specification of such systems, as well as high level models and implementations, based on a semantical foundation enabling formal methods suitable for the setting of open distributed systems. In order to integrate well with UML (for obvious reasons) we deliberately used well known UML concepts, and developed a modeling language, which may act as a textual counterpart to more graphical languages, and with more expressiveness capturing complete behavior. The language, known as OUN, includes executable imperatives for high-level system implementation, as well as a non-executable sub-language for system specification purposes. A compiler from implementation in OUN to Java was developed, allowing execution of OUN programs as well as an executable operational semantics in Maude [8].

We wished to contribute to the research direction of developing observable specifications of components, allowing top-down design of components where a "black box" specification of the observable behavior of aspects of a the component comes before the design of its inside structure. This is a development strategy recommended by theoreticians as well as practitioners; however, according to state of the art it seems that the questions of how to formulate behavioral specifications, and how to integrate them into an object oriented setting, are not quite settled – at least, when considering specification methods understandable for programmers without special mathematical training. In contrast, the state based style of specifying components requires the definition of a state-space within the components and requirements specifications can then be given by means of invariants expressed in, say, first order logic or by means of temporal requirements expressed in temporal logic. OCL is oriented towards specification of invariants, pre- and post-conditions by means of a language built upon first order logic (with some adjustments). In particular, it does not support specification of observable behaviors of objects and components.

We therefore found it interesting to develop OUN [29], allowing observable specification of (component) interfaces, supporting aspect oriented specification, as well as specification of assumed or required environmental behaviors; along with implementation of interfaces through (component) classes defining state space, invariants as well as imperative implementation of methods. In the language, a component is captured by an object of such a class, equipped with a local processor, and a local "run" method. Distribution is enhanced by facilities for asynchronous communication, and object orientation is maintained by staying within a generalization of remote method invocation. High-level language constructs for programming of processor release points and passive waiting construct, through nested guards, allow components to dynamically change from active to reactive behavior, and give a reasonable efficiency control at a high level. In order to support openness such as dynamic reconfiguration, a dynamic class construct is provided, allowing software components to be upgraded during execution.

Thus OUN may be used both for specification purposes as well as (high level) implementation purposes. The language may be seen as an extension of the basic mechanisms of OCL, through the OUN mechanisms for class level reasoning, extended to black box specifications of observable behavior of aspects of components. In OUN, behavioral specifications can be related to class level (OCL-like) specifications through notions of abstraction and refinement.

Note that the OUN notation will not be used in the examples discussed in the sequel. The intention of the brief summary of OUN presented above is to provide an overview over the ADAPT-FT project, which greatly influences this work, by revisiting the integrated platform and the notation it involves. More details can be found in the OUN specific papers listed at the ADAPT-FT project web site, including [9, 21, 20].

2.1.4 PVS as Underlying Semantic Domain

The Prototype Verification System (PVS) [30] is an environment for constructing precise specifications and for developing proofs that can be mathematically verified. PVS is based on a strongly typed higher-order logic with powerful verification and validation mechanisms. A salient feature of PVS is its capacity to provide a highly expressive and strongly typed specification language (PVS-SL) [30] tightly integrated with a type-checker, and an interactive general-purpose theorem-prover.

The PVS type system has been augmented by *predicate subtyping* and *dependent typing* mechanisms. Subtyping makes type checking more powerful by allowing stronger checks for consistency and invariance in a uniform manner. Subtyping renders, however, type checking undecidable and proof obligations may be generated during type-checking. A great deal of proof obligations can be discharged automatically using the PVS theorem-prover, whereas more involved ones require interaction from the user.

The PVS environment provides semi-automatic tools with significant automation including decision procedures for several common theories such as equality and linear arithmetic [30]. A particular strength of PVS is its capacity to exploit the synergy between its tools. For instance, the theorem proving can be used in type checking, and information obtained from type checking and model checking can be used in theorem proving. As the main goal of the ADAPT-FT project was to adapt, tune, redevelop,

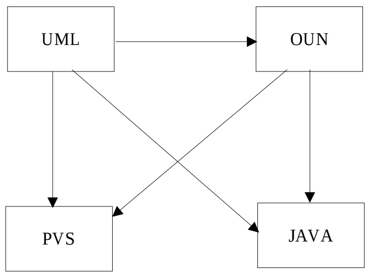

Figure 2: Translations in the ADAPT-FT Platform

and extend, formal methods towards the special needs of open distributed systems, an underlying semantical foundation was needed, preferably a foundation already implemented with a series of powerful tools. PVS [30, 31] was a natural choice in this respect, especially due to its strong type systems and functional sub-language, covering inductive data types and inductively defined functions, and its reasoning capabilities and tools, including some model checking facilities.

PVS provides a vehicle for defining the semantics of the OUN language, in a precise manner, and for defining the associated specification formalism, including concepts for refinement and composition, and at the same time allowing development and reuse of the semantical definitions in the design of tools, such as forms of reasoning tools. Even though the nature of PVS may be mathematically challenging to software engineers, a semantical basis is needed, from which engineering tools that are less esoteric may be developed. For instance, in the ADAPT-FT platform, integrating UML, OUN, Java and PVS, and by translating UML to OUN, Java and PVS, and OUN to java and PVS (see the arrows in Figure 2), one may develop tools at the level of UML diagrams or OUN programs, where the implementation of the tool is done at the PVS level (by means of PVS translations). Tools giving yes/no answers require no insight in PVS, and may provide useful feedback to the engineer. It would of course be desirable to have tools giving UML or OUN related feedback, built from PVS related tools; however, this is beyond the scope of the ADPAT-FT project.

2.2 Semantics of UML Notations in PVS

Rigorous analysis of UML models of large applications involves manipulation of huge software artifacts, in which case tool support is crucial. This in turn calls for formal semantic definitions for the graphical UML notations. Consequently, a formal semantics facilitates verification, validation and simulation of models and improves the quality of models and software design. In our case, formal semantic definitions for the UML notations are proposed by representing them in a well-founded formalism, namely the PVS specification language (PVS-SL).

A semantic definition for a UML sequence diagram captures properties that a system is expected to exhibit, i.e. system interaction described by the sequence diagram. Assumptions and invariants on the system are expressed in the PVS specification language as *axioms* and *conjectures* respectively. A trace of events specifies a possible run of the application specified by the sequence diagram if and only if the trace satisfies the requirements stated as predicates, provided that the assumption are fulfilled. For instance, for a trace that specifies a possible scenario of the interaction specified by the sequence diagram, and a given object participating in the interaction, the projection of the trace onto the set of events on the object must satisfy the requirements on the traces of the object. The requirements are stated as predicates on the set of traces of events. Static semantic constraints on modeling elements given as a set of well-formedness rules expressed in the Object Constraint Language (OCL) [44] can be specified similarly.

The formalization approach adopted for UML statecharts consists of definition of a set of elementary predicates describing properties of system states or operations. The set of elementary predicates is then partitioned into elementary states and events. A state describes a condition of the system that has a non-zero duration. We make a clear distinction between concrete states of the system and the abstract notion of states in UML statecharts. We define three categories of predicates associated with the notions of state vertex, guard condition, and action respectively. The predicate associated with a state corresponds to a condition that must hold for the state to be activated. Predicates associated with an action corresponds to a condition that holds after the execution of the action; that can be understood as action's postcondition. Whereas the state and guard conditions are boolean functions of values of the state variables before the execution of an operation starts, the postcondition is a boolean function of values of the state variables both before and after the execution of the operation.

A transition is enabled if the event instance generated matches its trigger, its guard condition is true and its source state is active. An enabled transition may be eligible for firing. Firing a transition will activate its target state and execute its action.

2.3 Tool Support

A tool support is a crucial component for successful application of a development framework in industrial settings. A CASE tool enables developers to manage large-scale projects, which usually involve manipulation of large software artifacts, and reduces development time by enabling them to discover subtle errors automatically. Experiences show that even the most carefully crafted formal specification and proof, can still contain inconsistencies, omissions and other errors [14].

To address this issue, we have developed a research platform, called the PrUDE (Precise UML Development Environment) tool [5]. The PrUDE integrates the UML [28] modeling notations and the PVS [30] formalisms, and their respective tools. Most of the commercial UML tools support only syntactic checks and code generation. Semantic checks are crucial in the development of critical systems, and hence it is necessary to integrate UML tools with a verification environment. In this regard, we use the PVS specification and verification environment and its toolkit in developing of our CASE tool, namely the PrUDE tool, to support not only formal verification but also testing and structured reviews.

The PrUDE tool supports automated generation of formal specifications from UML models in PVS based on the UML semantics proposed in [1, 3, 4, 38]. UML models along with business rules are translated into PVS so that the theorem proving technique is exploited in checking their validity and consistency. The resulting specification will be an input to the PVS verification toolkit running at the back-end.

The PrUDE tool suite supports checking well-formedness, consistency, model checking, proof checking and testing. The design models are created using a UML tool, whereas model analysis steps are performed using the PVS toolkit. The interface of the PrUDE tool to UML tools is based on the XMI [22] thus providing an explicit data exchange format. Since most of the existing UML tools support model exchange in the XMI format, the PrUDE platform is 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 graphical UML models they have created, with minimal interaction with the formal stuff generated from the models and processed at the back-end. The latter is achieved by identifying and implementing proof strategies that provide automated solutions for verification of system properties based on the formal semantic definitions. Test cases are generated from UML models that are valid, i.e. well-formed and model checked successfully. The PrUDE tool provides an automatic test case generator and a test execution component.

2.3.1 V&V Strategy in the PrUDE Platform

The V&V strategy underlying the PrUDE platform is shown in Figure 3. The rectangular boxes denote major activities, whereas the eclipses denote the resulting artifacts. The main steps in formal V&V process using the PrUDE tool are summarized below.

- Start by developing design model using any UML CASE tool that supports model exchange in the XMI format. The UML models in the sequel are developed using the ArgoUML v0.12 [17] tool.

- Describe properties of the modeling elements more precisely by adding suitable assertions. The assertions can be specified either in standard mathematical notations or OCL expressions.

- The XMI model exported from the UML model is imported into the PrUDE tool.

- Invoke the PrUDE tool and import the XMI file generated from the UML model. That means, a project in the PrUDE tool consists of a UML model, possibly augmented with business rules expressed as OCL constraints [44]. By using the PrUDE tool we can check well-formedness of the UML models, generate semantic models in PVS specification language, and analyze the resulting semantic models. Translation of UML models into PVS results in specification templates that include generic assertions such as well-formedness rules defining static semantics of UML models, and serving as the basis for the verification process. To perform a meaningful analysis, we need to complete the specification by adding some domain-specific assertions using the PVS property editor.

- Finally, we analyze the semantic models by invoking PVS tools within the PrUDE tool. Type-checking, model-checking, and proof-checking are among the major analysis steps. In PrUDE, the PVS theorem prover can be invoked either in a batch mode or in an interactive mode allowing users to guide the proof steps. If a verification step fails, a PVS log file consisting of messages indicating errors or omissions is output. We interpret the message and trace the discovered errors back to the UML model, fix the errors and iterate through the above steps.

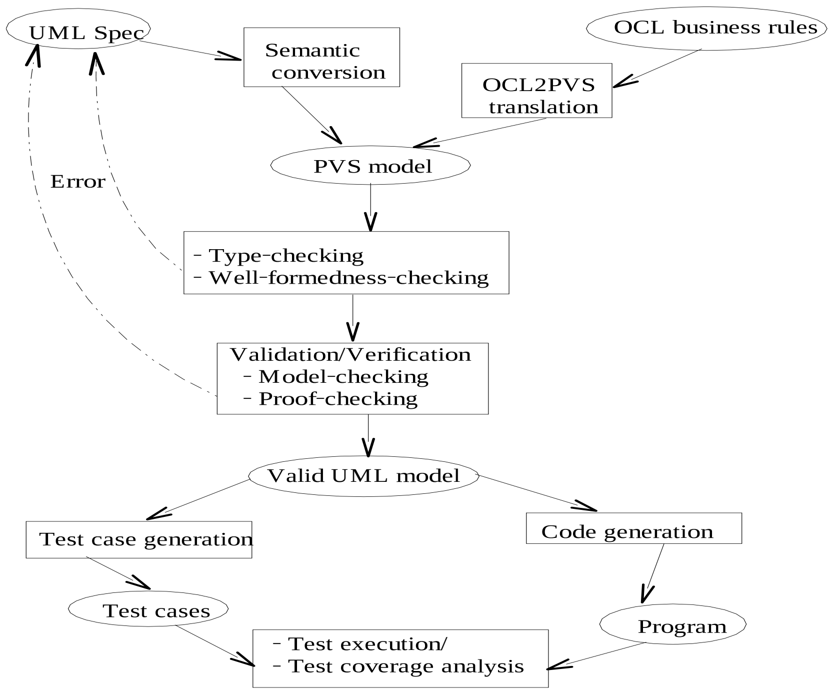

Figure 3: V&V Strategy Underlying the PrUDE Platform

If a verification process is successfully completed, i.e. a valid UML model is obtained, we proceed with the development process using the UML models. We may refine them to achieve an implementation of the system. The resulting program code can be tested using the PrUDE tool based on the UML specification. Test cases are generated from the valid UML model obtained after a series of V&V steps. The test cases are derived from various constraints related to the model, e.g. invariants, pre- and post-conditions. The current version of the PrUDE tool provides automatic test case generator and a test execution component for Java programs.

2.3.2 Known Limitations of the PrUDE Tool

The PrUDE tool is a research prototype developed to automate some aspect of the formal development framework we proposed. The PrUDE tool has some known limitations mainly with respect to implementation-related issues.

Firstly, the translation of system properties described in OCL expressions into PVS is done manually in the current version of PrUDE tool. Hence, developers are expected to be familiar with the OCL notation, and to be able to use it to express business rules. In the future, the PrUDE tool will be extended with a component that automatically translates and integrates OCL expressions into PVS specifications, which should be rather straightforward. Moreover, semantic definitions should be extended and more proof strategies should be developed for the verification of domain-specific properties.

Another shortcoming of the PrUDE tool is that feedback from the PVS theorem prover, in the case of a failed proof, is rendered as an error message embedded in a PVS message. By using the contextual vocabulary of the application domain in both the UML models and the PVS log messages, developers can trace the cause of an error message. But, the error message provides little support for automated tracing of the component in the UML model that contains the error. In the future, we will implement a parser that interprets the PVS error messages and translate them into a plain text understandable to the developers.

3 Case Study: a Banking System

In this section, we illustrate practical usability of the integrated framework we proposed [41] and the PrUDE tool by presenting an example of a formal development of a critical system - an electronic banking system. A typical banking system consists of the following main components: -

- a set of account numbers
- an *account master file* - a data structure for storing the current balance for each account;
- a list of *transactions* performed on the accounts during a given period of time;
- a set of *journals* for storing transactions that are received from teller stations but not yet entered into ledgers;
- a set of *ledgers* for tracking the flow of funds on their way through the system;
- a set of automatic teller machines (ATMs), usually known as cash machines;
- audit trails for recording actions of employees - essential information for verification of security requirements such as *non-repudiation*;
- a set of program modules for overnight batch-processing of transactions, i.e. for posting the transactions into appropriate ledgers, and for updating the account master file.
- several categories of actors - customers, employees, system administrators, auditors, etc.

Online processing includes a number of program modules for adding transactions to appropriate combinations of ledgers. For instance, if a customer has successfully deposited a certain amount of funds into an account, then a transaction is created and the same amount of funds is debited from the saving account ledger, and credited to the ledger recording the cash in the drawer. That means, a successfully completed *deposit* transaction involves modifications of both the *drawer* and the *debit* ledgers. This scenario is useful for monitoring the overall balance of the bank and activities of bank employees.

3.1 Summary of System Requirements

Functional requirement specification is a description of services that the system is expected to provide, how the system should react to a particular set of events, and how the system should behave in particular situations. The banking system is expected to provide the following list of functionalities. Note that the system requirements are significantly simplified and details are left out.

- The system must provide an authentication mechanism.
- Customers should be able to deposit, withdraw, or transfer funds, and inquire balances on their accounts.
- Customers should be provided with magnetic cards and PIN codes that will be used in the authentication process to use the ATM terminals. The ATM terminals should allow customers to choose a specific service, e.g. cash withdrawal, or balance enquiry by pressing an appropriate key on the terminal.
- Customers should be able to change PIN codes.
- Cancellation of a transaction should be allowed, if necessary, before its completion. A successfully completed transaction is kept in a journal until it is processed and posted to the appropriate ledgers and the account master file is updated.

Non-functional requirements are constraints put on the system, e.g. security requirements, and response time requirements. For an electronic banking system, a strong security mechanism is crucial to prevent customers from cheating each other and the bank, to prevent bank employees from cheating the customers and the bank, and to provide sufficient information for reconstruction of transactions and evidence to trace illegal actions. Different security models can be implemented to achieve the security requirements. In the Clark-Wilson model [7], for instance, security critical data items are constrained so that they can only be accessed or modified by users with appropriate level of security clearances. Data items are tagged with values specifying the level of access right required to access them, whereas actors are tagged with different levels of security clearances resulting in an access control matrix.

3.2 UML Models for the Application Domain

3.2.1 Functional and Structural Models

Using the UML modeling techniques, major components and aspects of the banking system and its business rules can be captured from different viewpoints. System functionalities and expected behaviors can be viewed as interactions between the system and its environment - actors such as customers, bank employees, and system administrators.

UML use case diagrams are description technique for specifying, at a high level of abstraction, *what* the system is supposed to do. Use cases are often used in the early stages of the design process to capture the intended system requirements. For instance, the use case diagram shown in Figure 4 describes major functionalities of the banking system. A possible realization of a use case can be modelled as an interaction and can be specified by a sequence diagram. Structural system properties

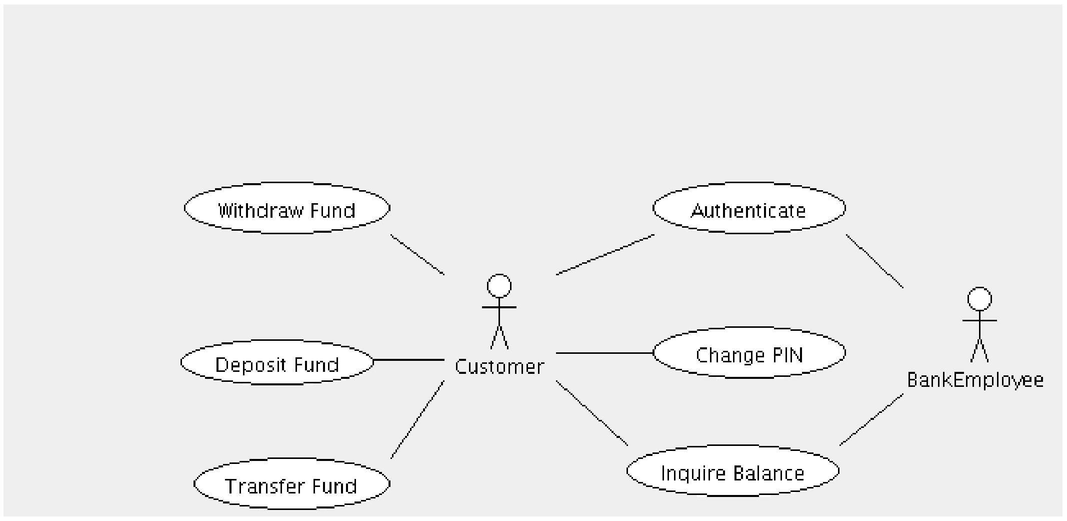

Figure 4: A Use Case Diagram Modeling System Functionalities

can be captured using class diagrams in terms of classifiers and relationships between them. This enables system developers to focus on design issues at a suitable level of abstraction by avoiding implementation details. The class diagram shown in Fig. 5, for example, models major components of the banking system: the classes *Bank, Person, Account, BankCard, Transaction, Ledger, Journal, ATM, CardReader, CashDispenser*, and *ATMSession* and relationships between them. The links connecting the classifiers model communication, containment, and dependency relationships. For example, the classes *Account* and *Bank* are connected by a *composition* relationship that specifies the fact that an instance of the class `Bank` contains one or more instances of the class `Account`, whereas an instance of the class `Account` is contained in exactly one bank.

A class specifies the data structure of its instances in terms of attributes and their

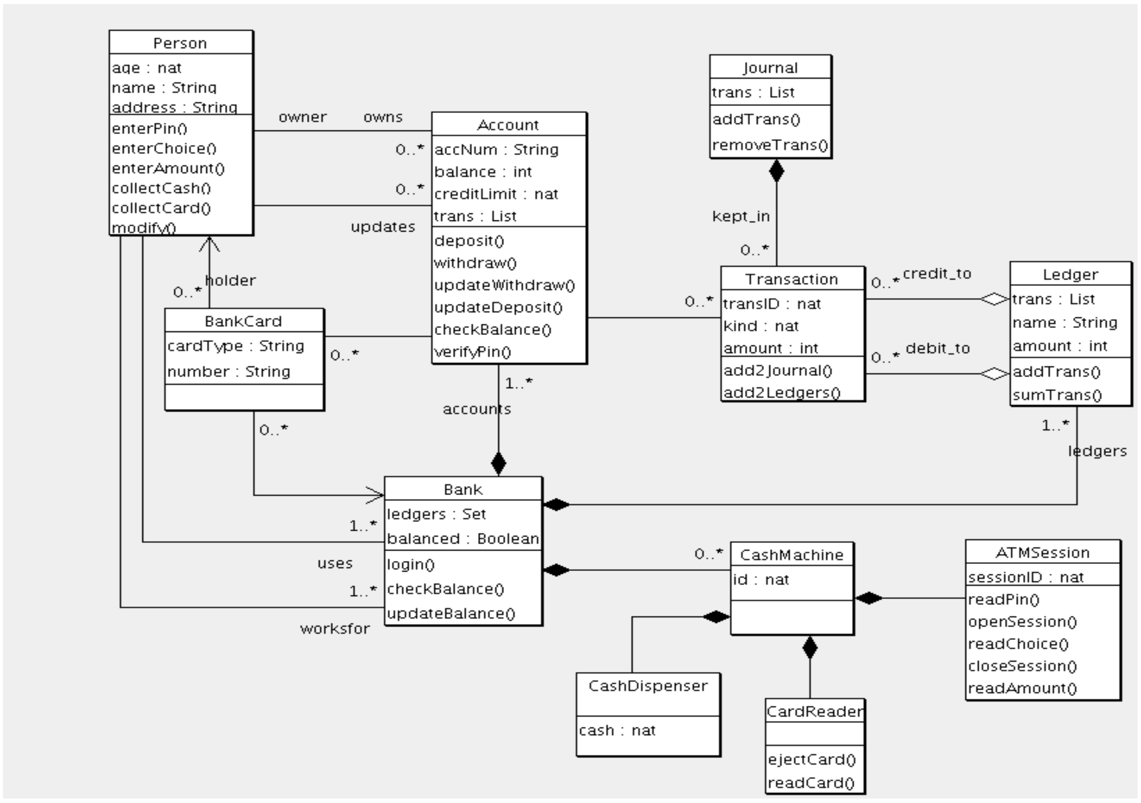

Figure 5: Class Diagram Describing Structure of the System

behaviors in terms of operations manipulating the data structures. The class *Account*, for instance, specifies a data structure that stores account number, current balance on an account, and a PIN code, and operations for manipulating them.

Remark 3.1 *The UML diagrams presented in the sequel are generated by using the ArgoUML [17] CASE tool. The stick arrowhead (→) on an association end in Figure 5 specifies the direction of navigation. The default multiplicity on an association end is* 1 *and association ends without explicit multiplicity assume the default value.*

The structural model of the banking system is shown in Figure 5 and briefly summarized below.

- An instance of `Bank` may contain one or more instances of the class `Account`, whereas an object of the class `Account` belongs to exactly one `Bank`. A bank may own zero or more cash machines, issue zero or more bank cards, have zero or more customers, etc.
- A cash machine contains exactly one cash dispenser, one card reader, and at most one ATM session at a time.
- A transaction is associated with exactly one account, whereas an account may contain several transactions that are temporally ordered based on their time of completion.

- We assume that an account is owned by exactly one customer, whereas a customer may own several accounts. This can easily be relaxed to accommodate the case where an account is owned by a set of customers.

- There are two associations between the `Transaction` and the `Ledger` classes. This is to capture the fact that every transaction is posted to a pair of ledgers; one recording credit to the bank and the other recording debit from the bank. This enables us to effectively record flow of funds and to monitor overall balance of the bank.

3.2.2 UML Sequence Diagrams

UML sequence diagrams are used to specify dynamic behavior of a system in terms of interactions between system components. They are useful for every stakeholder as they enable customers to visualize the specifics of their business processing; analysts to visualize the flow of processing; developers to visualize the objects that need to be developed and operations on those objects. An interaction is a possible realization of a use case described in terms of temporally ordered list of messages exchanged between the objects involved in the interaction.

Sequence diagrams exist in two variants, namely the *generic* and *instance* forms. The generic form of sequence diagram describes *must-interactions*, whereas the instance form describes *may-interactions* between objects. Damm *et al* [10] define a variant known as *Live Sequence Charts* (LSCs), the main addition being the ability to specify a *temperature* (hot or cold) to specify the *must* and *may* interactions respectively. A generic sequence diagram describes the interaction of classes, and documents all of the messages that can be exchanged between objects of the classes. An instance form of a sequence diagram describes a single possible scenario that may or may not occur. In the sequel, we consider the instance forms of UML sequence diagrams.

In an implementation of a behavior specified by a sequence diagram, a message corresponds to a method call on an object involved in the interaction. In a statechart diagram a message maps to an event that triggers a state transition. For example, the *withdraw Fund* use case shown in Figure 4 can be realized by the set of possible traces of events that lead to a successful withdrawal of funds, or to an unsuccessful attempt that is interrupted, for example, due to lack of sufficient funds in the account, or a wrong PIN code. For this discussion, we can assume that the authentication is successful. The sequence diagram shown in Figure 6 describes a scenario that leads to a successful withdrawal of funds from an ATM terminal. The interaction begins when a customer inserts a card into the card reader, which extracts information such as account number, balance on the account, PIN code, etc. and opens a session that interacts with the customer. The session prompts the user to enter a PIN code, and the ATM validates the PIN code. If the PIN code is valid, a list of the available services

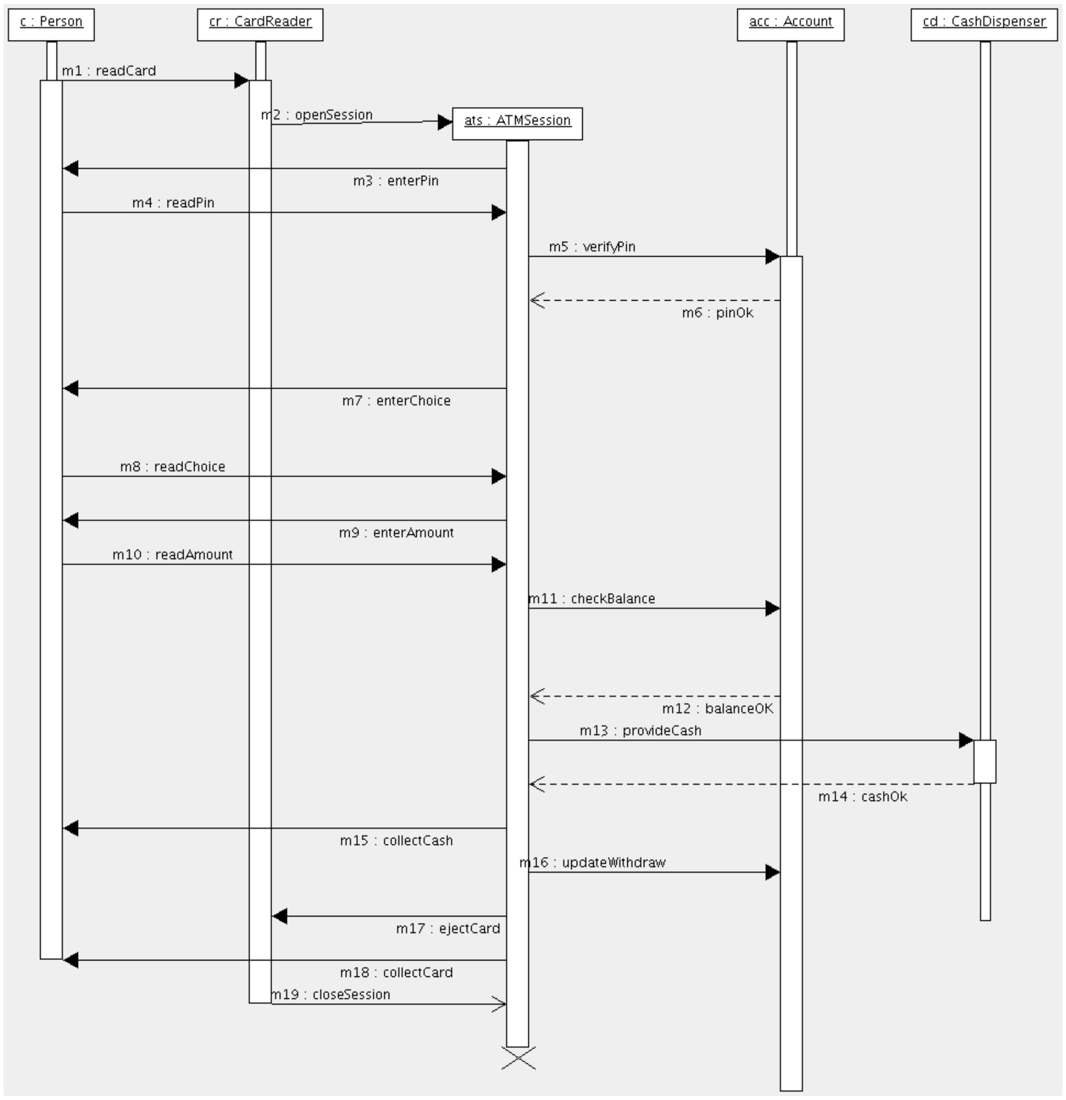

Figure 6: Sequence Diagram for a Successful *Withdraw Funds* Use Case

(deposit, withdraw, or transfer funds) is displayed. The customer selects a service, the *Withdraw* in this case, by pressing an appropriate key. The ATM session prompts the customer to enter the amount of funds to be withdrawn. When the customer enters the amount, availability of sufficient funds on the account, and sufficient cash in the dispenser are verified. If there is sufficient funds, the ATM deducts the amount from the balance of the account and updates the information on the card. The cash dispenser provides the cash and a receipt to the customer and the card reader ejects the magnetic card and closes the session. The ATM completes the transaction and sends it to the banking system. The system may keep the transaction in a journal for batch processing or add it to appropriate ledgers.

The balance on the account should be updated only after the transaction is completed and cash is delivered to the customer. In cases where a transaction is interrupted,

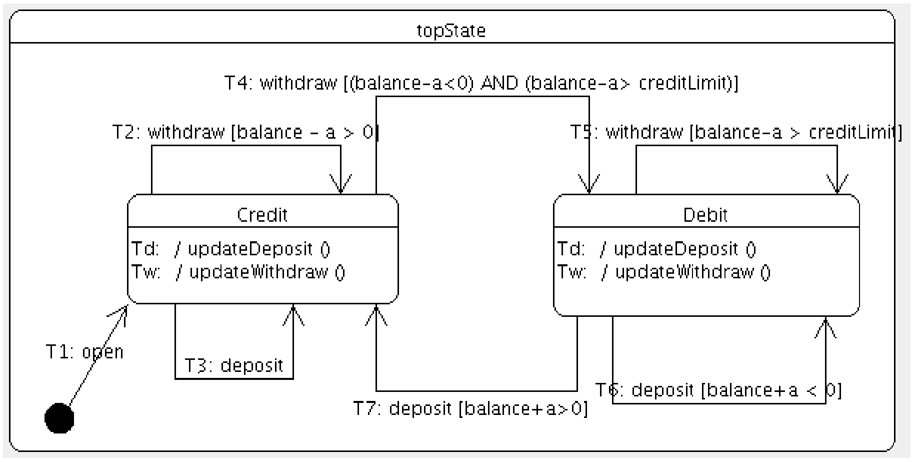
topState
T4: withdraw [(balance-a<0) AND (balance-a> creditLimit)]
T2: withdraw [balance - a > 0]
T5: withdraw [balance-a > creditLimit]
Credit
Td: / updateDeposit ()
Tw: / updateWithdraw ()
Debit
Td: / updateDeposit ()
Tw: / updateWithdraw ()
T1: open
T3: deposit
T6: deposit [balance+a < 0]
T7: deposit [balance+a>0]

Figure 7: Statechart Diagram for the *Account* Class

e.g. due to invalid PIN code, or insufficient funds in the account or in the cash dispenser, the system allows the customer, respectively, to reenter the PIN code a limited number of times, or to try a smaller amount of funds. If a transaction is interrupted, appropriate messages will be sent to the actors, e.g. a customer or an employee.

The sequence diagram shown in Figure 6 does not specify whether or not an account is updated before cash is successfully delivered to the user. It does not specify whether a successful authentication, i.e. correct PIN code, and availability of sufficient funds both in the account and the cash dispenser, are prerequisite for the delivery of cash either.

3.2.3 UML Statechart Diagrams

UML statecharts are used to model dynamic system properties as a complete life cycle of an individual object. This enables us to visualize interactions between the object and its environment. State machines are the basis for important security requirements specification [15]. To show that a given system property is fulfilled using a state machine, it suffices to identify some states satisfying that property and prove that all transitions preserve the property. In that case, if the initial state has this property, then by induction, the system property holds always. The essential features of a state machine are the notions of *state* and state *transitions* occurring at discrete points in time. A state is a representation of a behavior of an object, or the system as a whole, at a given point in time capturing exactly the aspects relevant to the problem. For example, an account can be either in the `Debit` state or the `Credit` state. The directed links connecting the states describe *transitions* between the states. The possible set of state transitions can be specified by a *next state* function, which defines, for every state, the set of next states depending on the present state and the triggering event.

A transition is labelled by a string of a general form `n:e[c]/sa`, where `n` is a transition name, `e` is a trigger event, `c` is a guard condition, and `sa` is a sequence of actions. For instance, in the statechart diagram shown in Fig. 7, which models complete life cycle of the class *Account*, `T1,T2,...,T7` denote transition names, `withdraw` and `deposit` are trigger events, and `balance - a > 0` is a guard on the transition `T2`. Sequences of actions are not explicitly shown in the statecharts diagram. For transitions triggered by event `deposit`, i.e. transitions `T3,T6,T7`, the list of actions includes updating of the balance with `balance:=balance + a`, whereas the `withdraw` event triggers transitions `T2,T4,T5`, leads to updating of the balance with `balance:=balance - a`. In the sequence diagram shown in Fig. 6, the later corresponds to the receiving and processing of the `updateWithdraw` event by an account object.

Assertions on states, guard conditions and actions in statechart diagrams are translated into PVS expressions and integrated into the semantic model using the PrUDE tool. A predicate on a state specifies a condition that must hold whenever the object to which the state machine is associated is in that state. For instance, properties of an account, when it is in the `Credit` and `Debit` states, can be captured by the following local predicates.

```
State :  TYPE+
acc:  VAR Account
Credit, Debit :  VAR State
pred(Debit) = balance(acc) < 0
pred(Credit) = balance(acc) ≥ 0
```

A guard condition on a transition is a predicate that specifies the condition that must hold for the transaction to fire. A guard condition can be viewed as a precondition for the operation associated with the event triggering the transition. Guard conditions on state transitions are translated into predicates in PVS specification language. For instance, the guard conditions on the transitions in Figure 7 can be translated into the following predicates in PVS, where the guards `g2,g4,g5,g6,g7` correspond to the transitions `T2,T4,T5,T6,T7`.

```
Guard :  TYPE+ :  [Account, nat → bool]
amount :  VAR nat
g2, g4, g5, g6, g7 :  VAR Guard
g2(acc,amount) = (balance(acc) - amount ≥ 0)
g4(acc,amount) = (creditLimit + amount ≤ balance(acc)) AND
                                (balance(acc) - amount < 0)
g5(acc,amount) = (creditLimit + amount ≤ balance(acc))
g6(acc,amount) = (balance(acc) + amount < 0)
g7(acc,amount) = (balance(acc) + amount ≥ 0)
```

The *creditLimit* is an attribute of the `Account` class, which specifies the maximum amount of funds a customer can withdraw in debt, i.e. a fixed value that shows how far the balance on the account can go below zero. The bank may change, through negotiation and agreement with the customer, the value of the *creditLimit* of an account.

3.2.4 Specification of Business Rules in OCL

UML diagrams are not detailed enough to address all the relevant aspects of system specification. Among other things, we need to describe additional constraints on elements in UML models that specify conditions and properties to be maintained, e.g. data invariants, pre- and post-conditions on operations, and complex multiplicity invariants. In this subsection, we describe some examples of constraints on the UML models given in previous sections using OCL [44, 28] expressions.
Rule 1: An instance of the class `BankCard`, and the `Account` with which it is associated must belong to the same bank. In reference to the class diagram shown in Figure 5, this property can be captured with the following invariant.
context BankCard **inv**:
$self.bank = self.account.bank$
Rule 2: For every instance of the class `BankCard`, the card holder must be the same as the owner of the account with which the card is associated.
context BankCard **inv**:
$self.holder = self.account.owner$ This rule can easily be modified to specify the case where an account is owned by several customers, e.g. a woman and her husband, by simply changing the type of the attribute *owner* to a set and the equality requirement to membership in a set.
Rule 3: The sum of the amounts of all transactions kept in the ledgers must be zero. This is equivalent to requiring that processing of every transaction preserves the overall balance of the banking system. Symbolically,

$$\sum_{l=1}^{n} amount(l) = 0 \tag{3.1}$$

where l is a ledger and n denotes the number of ledgers in the bank. This is a more complicated and important invariant that enables the banking system to prevent malicious acts by monitoring activities of its employees. For instance, if an employee wants to credit a given amount of funds to his own account, then he has to debit the same amount from another account, rather than just modifying the account's master file. This requirement can be expressed as an invariant in OCL.
context Bank **inv**:
$self.ledgers \rightarrow collect(trans.amount \rightarrow sum) \rightarrow sum = 0$
where *collect* is a predefined OCL operation on the collection type to return a subcollection of elements satisfying the predicate given as parameter. The relationships between the collections `ledgers`, `transactions`, etc. are as shown in Figure 5. This

invariant is translated to a conjecture in PVS specification (see Theorem 3.1) and checked directly using the PVS theorem prover.

This invariant is supposed to hold after completion of each transaction in an on-line processing, or daily in a batch processing. It significantly improves the security mechanism of the banking system by allowing monitoring of its overall balance. We specify a number of ledgers for recording different types of transactions. To simplify our discussion, we assume that the bank contains only three ledgers, namely:

- a *drawer* ledger for recording transactions affecting the amount of cash in the drawer;
- a *credit* ledger for recording transactions that affect the credit of the bank; and
- a *debit* ledger for recording transactions that affect the debit of the bank.

Note that the sets of transactions recorded in the ledgers are not mutually disjoint. When a transaction is successfully completed, it is processed and added to a pair of relevant ledgers. For instance, a *deposit* transaction is added to the *drawer* ledger to reflect the increment of cash in the drawer, and at the same time to the *debit* ledger to reflect the increment in the debit from the bank, i.e. the amount the bank must owe its customers.

Rule 4: The system must not allow withdrawal of an amount of funds that makes the balance on the account less than the pre-agreed `creditLimit` - a fixed amount of funds that the customer can withdraw in debt disregarding ongoing transactions. For customers without such an agreement, `creditLimit` is equal to zero. Moreover, if a withdrawal is successfully completed, the balance on the account must be updated. These requirements are specified as pre- and post-conditions on the `withdraw` operation as follows:

context $Account :: withdraw(amount : nat) : nat$

pre: $self.balance - amount \geq self.creditLimit$

post: $self.balance = self.balance@pre - amount$

where $balance@pre$ indicates the value of variable $balance$ at the start of the execution of the operation.

A pre-condition on an operation corresponds to a guard condition on a state transition that must be fulfilled for the transition to be fired. State transitions must preserve local invariants, but a state transition may be undesirable globally. That is, when a transition is fired, the effect of actions associated with the transition may lead to undesirable behavior. For instance, transferring funds to a wrong account number is possible as far as the pre- and post-conditions are fulfilled. That is, the pre- and postcondition are necessary but not sufficient to enforce such requirements.

Rule 5: If a person is both a customer and an employee of a bank, then the person must not be allowed to modify his own account. This requirement is related to the *separation*

of duties security design principle. To enforce this requirement, every employee must be identified uniquely, for instance by a combination of social security number and a password, and a set of accounts that the employee can update must be specified. This requirement is expressed in OCL as follows:
contextPerson **inv**:
  $self.updates \rightarrow excludes(self.owns)$
where *excludes* is a predefined OCL operation, and the `updates` attribute contains the set of accounts an employee can modify (see section 3.4 for more discussion).
Rule 6: After a successful withdrawal transaction, the effect of the withdrawal must be reflected on the account by updating its balance before the cash is dispensed. What if the cash dispenser fails to deliver the cash after the balance is updated? This is an instance of the transaction integrity problem that can be handled by a new transaction that reestablishes the correct balance.

In general, transactions can be kept in a journal until they are processed and added to appropriate ledgers by batch processing modules during the night. In our example, however, we assume that a transaction is put into ledgers immediately after it is successfully completed. System properties described in OCL expressions are integrated into the PVS specifications generated from the UML models and verified using the PVS toolkit.
Rule 7: For any account, at most one ATM session can be associated with the account at any given time. This requirement prevents concurrent withdrawals from the same account by requiring uniqueness of an ATM session. This can be implemented by updating the balance on the account before a new ATM session can be started.
context ATMSession **inv**:
  $self.allInstances \rightarrow forall(s1, s2|s1 <> s2 \; implies \; s1.account <> s2.account)$
where the *allInstances* and the → are predefined OCL operations on types and object collections respectively.
Rule 8: The balance on an account is equal to the difference between the sum of deposited funds and the sum of withdrawn funds. This constraint can be specified as an invariant expressed in OCL, and translated into a conjecture in PVS and discharged.

context account **inv**:

$self.balance =$

$self.trans \rightarrow select(transKind = deposit)) \rightarrow collect(trans.amount) \rightarrow sum$

  - $(self.trans \rightarrow select(transKind = withdraw)) \rightarrow collect(trans.amount) \rightarrow sum$

where *select* and *collect* are OCL operations and `trans` is the list of transactions performed on the account object. The *select* operation returns a sub-list of *trans* for which the boolean expression is true. The *collect* operation derives a collection of objects of type different from the original collection. It returns a bag of natural numbers, i.e. amounts associated with the transactions selected. The *sum* operation returns the total sum of the amounts in the set of transactions to which it is applied.

3.3 Formal Analysis Using the PrUDE Tool

The main purpose of integrating semi-formal modeling techniques with formal methods (FMs) is to exploit the mathematical foundation underlying FMs in reasoning about correctness of the graphical models. This requires translation of graphical UML models, and OCL constraints to PVS specifications to make them amenable to rigorous analysis. The translation of UML models is based on the semantic definitions we proposed for UML notations [1, 3, 4, 38] and implemented in the PrUDE [5] tool to support automatic translation of UML models into formal specifications in PVS. The translation of OCL expressions into PVS is rather straightforward since OCL is based on first-order logic and PVS is based on higher-order logic.

The formal system development process using the PrUDE platform consists of the following major steps.

- Analysis and design of a system using UML modeling techniques. In this step, structural and behavioral properties of major system components, relationships between the components, and possible interactions between them are described using the UML modeling techniques and notations. Any UML CASE tool that supports model exchange in the XMI format can be used to automate this step. In the sequel, the ArgoUML [17] tool is used.

- PVS specifications are obtained by translating UML models and rigorously analyzed using the verification mechanisms and tools provided by the PVS environment in order to prove that the specifications satisfy the requirements. If an error is discovered during this step, e.g. if a type-checking fails, then the above steps are repeated until an error-free, UML model is obtained.

- When a valid, i.e. a well-formed, UML model is obtained the developer proceeds with the implementation and code generation in a language of interest. Most of the UML CASE tools support generation of skeletons of codes in programming languages such as Java, C++, etc.

Specifications of generic properties of UML models, e.g. the well-formedness constraints, can be captured by the semantic definitions for UML notations and obtained from the translation of UML models into PVS. The resulting PVS specifications are analyzed using the PVS verification tools such as the type-checker, theorem-prover and model-checker. The PVS specification shown in appendix B is, for instance, automatically generated from the sequence diagram shown in Figure 6 using the PrUDE tool.

The following are examples of generic properties of UML models. These properties follow from well-formedness constraints put on UML models.

- For every object involved in a given interaction that is specified by a sequence diagram, its class should be specified at least in one class diagram.
- For a given class and a statechart diagram describing its life cycle, an operation that triggers a state transition must be in the set of methods of the class.

As mentioned previously application-specific properties should be added directly into the PVS specification. For instance, the invariant stated as Theorem 3.1 specifies the requirement that the overall balance of the bank must be preserved by a *processing* of a transaction, i.e. the addition of the transaction into a pair of appropriate ledgers (see *Rule 3* in Section 3.2.4). In other words, for every transaction and a bank, processing of the transaction, i.e. its addition to a pair of appropriate ledgers, should preserve the overall balance of the bank.

To specify and verify this requirement, we start by declarations of transaction, ledger, bank, types. In fact these declarations are extracted from the PVS specification resulted from the translation of UML models. Note that the excerpt from the PVS specification contains the minimal information necessary for the following discussion.

```
TransactionKind : TYPE+ = {deposit, withdraw, transfer}

Transaction :TYPE+ = [# transId: int,
                        transKind: TransactionKind,
                        amount: nat #]
Ledger : TYPE+ = [# kind : LedgerKind,
                    trans : list[Transaction] #]
Bank : TYPE+ = [# accounts: setof[Account],
                  drawer : Ledger,
                  credit : Ledger,
                  debit : Ledger #]
```

A bank consists of a set of accounts, and three ledgers for recording cash in the drawer, the credit, and debit of the bank. A ledger consists of a list of transactions in the order of their occurrences. To every transaction there is an amount of funds.

The recursive function `sum_ledger` computes the sum of the amounts of funds associated with the list of transactions given as a parameter. When the PVS specification was typed, a TCC was generated in order to ensure termination of the recursion. The TCC was discharged automatically using the theorem-prover command `(grind)`.

```
sum_ledger(lt:list[Transaction]) : recursive nat = CASES lt OF
                           null : 0,
                           cons(t,lt1) : amount(t) + sum_ledger(lt1)
```

```
                         ENDCASES
                    MEASURE length(lt)
```

The predicate `balanced?()` defined on the `Bank` type states the condition that must hold when a bank is in the balanced state, i.e. the sum of all ledgers is equal to zero.

```
b : VAR Bank
balanced?(b): bool = sum_ledger(trans(drawer(b)))
                     + sum_ledger(trans(credit(b)))
                     + sum_ledger(trans(debit(b))) = 0
```

Processing of a transaction means addition of a successfully completed transaction into a pair of ledgers, depending on the kind of the transaction. More specifically, the transaction is appended to the sequence of transaction in the ledgers. It may be necessary to alter the amount associated with the transaction, for instance, when a withdrawal transaction is added to the drawer ledger. The auxiliary function `neg()` was defined for this purpose, whereas the function `processTrans()` specifies the processing of transactions.

```
t : VAR Transaction
neg(t) : Transaction = t WITH [amount:=-amount(t)]

processTrans(t,b) : Bank = IF transKind(t)=withdraw THEN
   b WITH [drawer:=drawer(b) WITH [trans:=cons(neg(t),trans(drawer(b)))],
           credit:=credit(b) WITH [trans:=cons(t,trans(credit(b)))]]
 ELSE IF transKind(t) = deposit THEN
      b WITH [drawer:=drawer(b) WITH [trans:=cons(t,trans(drawer(b)))],
        debit:=debit(b) WITH [trans:=cons(neg(t),trans(debit(b)))]]
      ELSE b
      ENDIF
 ENDIF
```

where `WITH` is a PVS construct for overriding values of fields of a record. Since the effect of processing a *transfer* transaction is the same as that of *withdraw* transaction, it is not considered in the definition of the *processTrans()* operation. The definition of the *processTrans()* operation is based on the assumption that a transaction is processed immediately after it is completed, otherwise the operation would have been recursive.

Now let us specify the requirement as a theorem and prove it by invoking the PVS theorem-prover.

Theorem 3.1 *For any transaction* `t` *and a bank* `b`*, processing of the transaction preserves the overall balance of the bank. In other words, if the bank is in a balanced state, and a transaction is successfully processed, then the bank remains balanced. Symbolically,*

```
thm2: THEOREM FORALL t,b: balanced?(b) => balanced?(processTrans(t,b))
```

The following is a slightly reformatted excerpt from a proof of the theorem generated by the PVS toolkit.

```
thm2 :

  |
  |--------------------------------------------------------------------
  | {1}  FORALL t, b: (balanced?(b) => balanced?(processTrans(t,b)))
  |
```

Trying repeated skolemization, instantiation, and if-lifting, then Expanding the definition of sum_ledger, and then Expanding the definition of `processTrans`, this simplifies to:

```
thm2 :

  | {-1}   (CASES trans(credit(b!1)) OF
  |           null: 0,
  |           cons(t, lt1): amount(t) + sum_ledger(lt1)
  |         ENDCASES)
  |       +(CASES trans(debit(b!1)) OF
  |           null: 0,
  |           cons(t, lt1): amount(t) + sum_ledger(lt1)
  |         ENDCASES)
  |       +(CASES trans(drawer(b!1)) OF
  |           null: 0,
  |           cons(t, lt1): amount(t) + sum_ledger(lt1)
  |         ENDCASES) = 0
  |--------------------------------------------------------
  | {1}    (CASES (IF transKind(t!1) = withdraw THEN
  |                  cons(t!1,trans(credit(b!1)))
  |               ELSE b!1`credit`trans ENDIF) OF
  |           null: 0,
  |           cons(t,lt1): amount(t) + sum_ledger(lt1)
  |         ENDCASES)
  |       +(CASES (IF transKind(t!1)=withdraw THEN
  |                  b!1`debit`trans
  |                ELSE cons(neg(t!1), trans(debit(b!1))) ENDIF) OF
  |           null: 0,
  |           cons(t,lt1): amount(t)+sum_ledger(lt1)
  |         ENDCASES)
  |       + (CASES (IF transKind(t!1) = withdraw THEN
  |                  cons(neg(t!1), trans(drawer(b!1)))
  |                 ELSE cons(t!1, trans(drawer(b!1))) ENDIF) OF
  |           null: 0,
  |           cons(t,lt1): amount(t)+sum_ledger(lt1)
  |         ENDCASES) = 0
```

Lifting IF-conditions to the top level,

```
thm2 :
```

```
{-1}          IF null?(trans(credit(b!1)) THEN
                 (0 + (CASES trans(debit(b!1)) OF
                          null: 0,
                          cons(t, lt1): amount(t) + sum_ledger(lt1)
                          ENDCASES)
                 + (CASES trans(drawer(b!1)) OF
                          null: 0,
                          cons(t, lt1): amount(t) + sum_ledger(lt1)
                          ENDCASES)) = 0
              ELSE amount(car(trans(credit(b!1))))
                 + sum_ledger(cdr(trans(credit(b!1))))
                 + (CASES trans(debit(b!1)) OF
                    null: 0,
                    cons(t, lt1): amount(t) + sum_ledger(lt1)
                    ENDCASES)
                 + (CASES trans(drawer(b!1)) OF
                    null: 0,
                    cons(t, lt1): amount(t) + sum_ledger(lt1)
                    ENDCASES) = 0
              ENDIF
-----------------------------------------------------------------
{1}           IF transKind(t!1) = withdraw THEN
                  (CASES cons(t!1,trans(credit(b!1))) OF
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                   ENDCASES)
                + (CASES b!1‘debit‘trans OF
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                   ENDCASES)
                + (CASES cons(neg(t!1), trans(drawer(b!1))) OF
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                   ENDCASES) = 0
              ELSE
                  (CASES b!1‘credit‘trans OF
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                   ENDCASES)
                + (CASES cons(neg(t!1), trans(debit(b!1))) OF
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                    ENDCASES)
                + (CASES cons(t!1, trans(drawer(b!1)))
                          null: 0,
                          cons(t,lt1): amount(t) + sum_ledger(lt1)
                  ENDCASES) = 0
            ENDIF
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of thm2.

```
Q.E.D.
```

3.4 Model-based V&V in Making Design Decisions

In the UML standard document [28] it is stated that associations on base classes are inherited by its subclasses. We briefly discuss this issue, present a concrete example of a deviation of designers' understanding of the issue, and illustrate how the proposed development framework may assist developers in making design decisions in cases when the semantics of the UML notations is ambiguous and/or inconsistent with intuitive informal semantics.

In UML, the semantics of *specialization/generalization* relationship between classifiers satisfies *Liskov's substitutability principle* [25] stated as follows:

> *If S is a subtype of type T, then objects of T in a program may be substituted with objects of type S without altering the desired properties of the program, e.g. its correctness. In other words, if $p(x)$ is a property provable about an element x of type T, then $p(y)$ should be true for an element y of type S.*

Let us consider the specialization/generalization hierarchy of the classes extracted from the class diagram shown in Figure 5, modified/refined and shown in Figures 8 and 9 so that they suit the discussion in this section. When applied to the inheritance hierarchy shown in Figure 8, Liskov's substitutability principle states that objects of specialized classes, namely the `Employee` and `Customer` classes, are substitutable for objects of the base class `Person`. In other words, the associations between classes `Person` and `Account` are inherited by the subclasses `Customer` and `Employee` of the class `Person`. Thiat means, both subclasses are associated with the class `Account` by the two associations they inherit from the base class.

In PVS semantic models, we specify the inheritance hierarchy by representing classes and subclasses as PVS types and subtypes, respectively. Subtyping satisfies Liskov's substitutability principle.

```
Person :  TYPE+
Employee :  TYPE+ FROM Person
Customer :  TYPE+ FROM Person
p :  VAR Person
b :  VAR BAnk
acc :  VAR Account
```

Moreover, semantics of inheritance relationship requires that sets of objects of specialized classes are mutually disjoint in the sense that they cannot have a common subclass. This property does not automatically follow from the specification of subclasses as uninterpreted subtypes declared above. Hence, we need to explicitly specify this property as a constraint on the metamodel (see axiom `disjoint_ax` in the `corePackage`

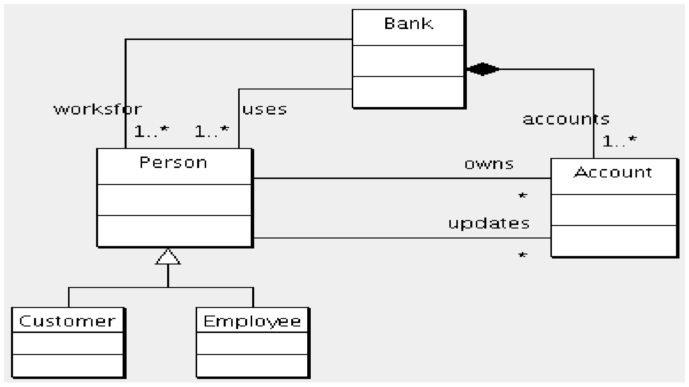

Figure 8: Associations in Inheritance Hierarchy

theory in the appendix A). There are two associations between the classes `Person` and `Account` (see Fig. 8: the *updates* association that captures the relationship between an account and a bank employee; and the *owns* association that specifies a relationship between an account and a bank customer. Specialized classes inherit both the structure and behavior of the base class. Note that the two associations may not be mutually disjoint, i.e. a single person can be associated to an account both as a customer and an employee (at least at this point) in which case additional restriction may apply to the set of accounts such a person may update. More specifically, a person should not be allowed to modify his own account.

According to the semantics of inheritance in UML notations, an association involving a base class is inherited by all its subclasses. This means, referring to Figure 8, that the subclasses `Employee` and `Customer` inherit the two associations `owns` and `updates` from the base class `Person`. A person is said to be associated with a bank as an employee if there exists an account in the bank, which the person may updates. A person is said to be associated with a bank as a customer if there exists an account in the bank, which the person owns. We specify the associations and their properties as follows.

```
owns :  [Person -> set[Account]]
updates :  [Person -> set[Account]]
uses :  [Bank -> set[Person]]
worksfor :  [Bank -> set[Person]]

worksfor_ax:AXIOM (FORALL p,b:  worksfor(b)(p) IFF
            (EXISTS acc:  accounts(b)(acc) AND updates(p)(acc)))

uses_ax:  AXIOM(FORALL p,b:  uses(b)(p) IFF
               (EXISTS acc:  accounts(b)(acc) AND owns(p)(acc)))
```

Based on the above axioms, let us specify and verify the property stated as business ***Rule 5*** in section 3.2.4.

Theorem 3.2 *If a person p is an employee and a customer of a bank b, then the person must not be allowed to update an account acc which (s)he owns. Symbolically,*

```
thm6: THEOREM (FORALL p,b,acc: (worksfor(b)(p) AND uses(b)(p)) IMPLIES
                NOT (owns(p)(acc) IFF updates(p)(acc)))
```

An attempt to prove the above theorem by invoking the PVS theorem prover, turned out to be unsuccessful by resulting in two unprovable subgoals: *thm6.1* expressed as unproved sequent with several antecedents and no consequents; and *thm6.2* expressed as a sequent with consequent contradicting the consequent of the original goal. The counter examples are given as PVS debugging messages, which indicate that either the antecedents are inconsistent, or they are insufficient to prove the sequent.

```
thm6 :
  |--------------
  {1} (FORALL p,b,acc: (worksfor(b)(p) AND uses(b)(p)) IMPLIES
                NOT (owns(p)(acc) IFF updates(p)(acc)))

Rule? (grind :theories ("inheritance"))

Trying repeated skolemization, instantiation, and if-lifting, this
yields 2 subgoals:

thm6.1 :
    {-1} GeneralizableElement_pred(p!1)
    {-2} Classifier_pred(p!1)
    {-3} Class_pred(p!1)
    {-4} Person_pred(p!1)
    {-5} owns(p!1)(acc!1)
    {-6} updates(p!1)(acc!1)
   |--------------

Rule? (postpone) Postponing thm6.1.

thm6.2 :
    {-1} GeneralizableElement_pred(p!1)
    {-2} Classifier_pred(p!1)
    {-3} Class_pred(p!1)
    {-4} Person_pred(p!1)
   |--------------
    {1} owns(p!1)(acc!1)
    {2} updates(p!1)(acc!1)

Rule? quit
```

```
Run time  = 1.45 secs.
Real time = 50.58 secs.
```

A closer investigation of the axioms reveals that the antecedents are insufficient to prove the sequent. That means, it is inconclusive from the specified axioms, whether or not a person who can update an account is different from the one who owns it. Hence, we need to analyze the UML class diagram since this contradicts the intended/required property of the system.

A solution is to specify the two associations *owns* and *updates* between the specialized classes `Customer` and `Employee`, and the class `Account`, respectively. We capture the desired property by specifying an {`xor`} (*exclusive or*) – a predefined constraint in UML – on the two associations (see Figure 9). The {`xor`} constraint specifies that for any instance of the class `Account`, either it is associated with an instance of the class `Customer` by the association *owns* or with an instance of the class `Employee` by the association *updates*, but not both. The {`xor`} constraint is translated to the following axiom in the PVS specification.

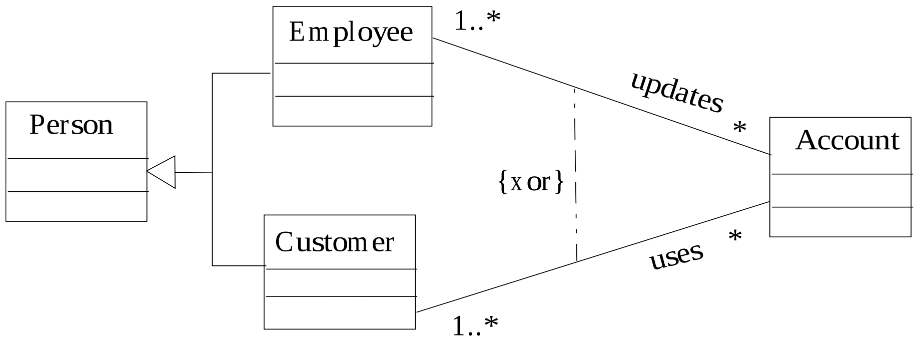

Figure 9: Associations in Inheritance Hierarchy

```
xor_ax:  AXIOM (FORALL acc:  (owns(c)(acc) XOR updates(e)(acc)))
```

By including axiom `xor_ax` in the PVS specification (see appendix E), theorem `thm6` was discharged automatically by invoking the PVS prover, with the single command *(grind :theories ("inheritance")).*

```
thm6 :
  |-------
  {1} (FORALL p,b,acc: (worksfor(b)(p) AND uses(b)(p)) IMPLIES
                NOT (owns(p)(acc) IFF updates(p)(acc)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of thm6.

Q.E.D.
```

This example shows how formal V&V can reveal subtle errors (omissions, inconsistencies, etc) in UML models, which may not be discovered otherwise, and how log messages can help us to reconsider our design decisions. Although the detected error might seem trivial, it is an example of typical errors that can easily be overlooked during design phase, until its it is too late and costly to fix them.

3.5 Discussions

Generic correctness requirements on UML models are specified and automatically verified by implementing the well-formedness rules (WFRs) defining the UML static semantics in the PrUDE tool. Application-specific requirements should, however, be specified during the development process and this requires certain amount of developers' interaction with the PrUDE platform, thus full automation of the verification process is not realistic. System models are expressed in UML notations, whereas additional constraints on models are captured either by OCL or OUN expressions. The ADAPT-FT project integrates UML, OUN and PVS into a platform for the formal development of open distributed systems (ODS). In the PrUDE tool, however, OCL is used instead of OUN to enhance the UML notations. The UML models, and the constraints expressed in OUN or OCL are translated to PVS to take advantage of the PVS theorem proving facilities in verifying correctness of the UML models [1, 3, 4].

The PrUDE platform relies on UML for modeling, and on OCL for specifying constraints on the models, and on PVS [30] for consistency checking and verification of the specifications. It allows developers to interactively insert assertions directly using the PVS editor. This seems to be in contrary to the main purpose of integrating formal methods with graphical modeling techniques, namely, hiding the processing of formal software artifacts from practitioners. However, as stated in [6], complete automation of the translation of semi-informal models into formal specifications is unlikely, since the informal descriptions are inherently incomplete. Most of the generative translations results in only skeletons of formal specifications and require the specifiers to provide additional details to complete the semantic models.

Hence, translation of UML models into PVS results in a skeleton of formal specification that is neither 'complete' nor detailed enough to perform a meaningful verification of the properties of the system in question. The level of details of the formal specifications generated from the UML models directly depends on the information available in the UML models and the detail of semantic definitions implemented in the CASE tool automating the translation.

The PrUDE tool is developed based on the formal semantic definitions we proposed for a subset of the UML notations. Even if semantics for the whole UML notations is defined and implemented in the platform, it is impossible to capture all application specific properties although some generic properties can be implemented in the platform and instantiated in applications. Hence, allowing users to add system properties is essential for performing a meaningful verification and makes the PrUDE platform more flexible. This feature seems to contradict with the very purpose of developing the integrated platform and the supporting tool. This issue can be addressed in one or more of the following ways:

- Formalize generic domain-specific properties and implement them;
- Use more user friendly and intuitively understandable specification languages such as the tabular notation; and [32, 19] that have semantic definitions in PVS.
- Define and implement suitable proof strategies that capture domain-specific properties.

The separation of generic semantic theory and model-specific definitions allows the development of a meta-theory and proof strategies for UML models, which are useful to reduce users' interaction with verification tools.

Another issue that needs further consideration is communication of results of formal verifications using PVS tools to developers who may not have knowledge about the PVS environment. In the current version of the PrUDE tool, results from PVS verification tools are reported as plain texts. The main challenge is, to present the feedback from the PVS tool, e.g. an error message from type-checking or the theorem-proving, in such a way that it enables the developers to trace the cause of errors back to the UML models they have created and identifying the model elements containing the errors. Such a mechanism is very crucial for practical usability of the proposed development framework and its tool.

A preliminary investigation shows that it is feasible to achieve this by recording a sufficient amount of information that is necessary to re-engineer the UML models from the PVS specifications. For instance, preserving the system vocabulary across the graphical models and formal specifications significantly contributes to the improvement of practitioners understanding of feedbacks from the verification step. Moreover, encoding model information in a notation that preserves the structure of UML models can improve understanding of the developers, and at the same time represent sufficient information about model elements.

An alternative approach is to implement an 'intelligent' parser that can interpret the log file generated by the PVS verification tools. Even though the error messages might indicate the cause of errors in the UML models, they are not sufficiently detailed. In the future we implement an *"intelligent"* parser that will extract textual "English-only" messages from the raw PVS log messages.

4 Conclusion and Future Work

Our framework relies on PVS [30] as a formalism for verification of specifications. Basic modeling constructs and constraints on UML diagrams can be expressed formally in the PVS specification language in terms of functions and abstract data types [2]. Our approach to consistency checking was described in [40] where software specification is done in a development framework, which integrates UML and PVS toolkit. A combined use of the different UML viewpoints improves integrity and completeness of system models, which in turn provides a firm foundation for a better design and implementation decisions.

By integrating semi-formal modeling notations with formal methods (FMs), we have taken a step towards exploiting the mathematical foundation underlying the FMs for rigorous analysis. This requires translation of UML models into PVS specifications that are amenable to rigorous analysis. The translation is based on semantic definitions we proposed in [1, 3, 4, 38] and provides the necessary link for reasoning about the UML models. The PrUDE tool automates most of the translation of UML models developed by using UML tools supporting data exchange in the XMI format into PVS specifications. The PVS toolkit allows us to perform conformance checks of the semantic models as illustrated in section 3.

It is not feasible to implement all application-specific properties in a CASE tool as such properties will not be available before the development process starts. Generic properties, however, can be implemented in CASE tools. Hence, allowing users to add domain-specific properties is essential to perform a meaningful verification possibly guided by users. Moreover, this feature makes the PrUDE tool flexible and useful to a wider group of users. The fact that system designers are allowed to specify system properties in PVS, seems to contradict with the very purpose of developing the integrated framework and the supporting tools: minimizing user's interaction with verification tools. This issue can be addressed by using a user friendly specification language such as the tabular notation [32] and by identifying a number of proof strategies for application-specific properties, to minimize user's interaction with the theorem-prover.

Another issue that needs further consideration is how to communicate feedbacks from PVS toolkit to developers who may not be expert in the PVS environment. One possible approach is to implement an 'intelligent' parser that interprets the output from the PVS verification tools, and enables the developer to navigate the model to identify source of errors.

We presented an integrated development framework and a supporting tool and illustrated how it can be used in the development of critical applications. We strongly believe that integrating formal methods with a well-accepted visual modeling language like the UML into a development process improves system reliability and clarity of the meaning of the modeling elements.

The main contribution of our work is precise representation of UML models by translating them into PVS specifications and performing rigorous analysis. The interpretation of the feedbacks from the PVS verification tools into UML model needs to be addressed. This transformation is crucial for communicating results of formal analysis to software practitioners that may not be familiar with the PVS environment. A significant limitation of our framework is that when a proof fails there is no real explanation of the cause in the context of the UML models.

Acknowledgements

We would like to thank Dr. Issa Traoré for reviewing earlier versions of this report and for his invaluable comments.

References

[1] D. Aredo, I. Traoré, and K. Stølen. An Outline of PVS Semantics for UML Class Diagrams (extended abstract). In *the Proc. of The 11th Nordic Workshop on Programming Theory NWPT'99*, Uppsala, Sweden, October 6-8, 1999.

[2] D. B Aredo. Formalization of UML class Diagrams in PVS (Extended Abstract). In *the Proc. of Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, at OOPSLA99.*, Denver, Colorado, USA, November 2, 1999.

[3] 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*, 8(7):674–697, July 2002.

[4] D. B. Aredo. Semantics of UML Statecharts in PVS. In *the Proc. of 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI2003)*, Orlando, Florida, USA, July 27-30, 2003.

[5] M. Belaid and I. Traoré. The Precise UML Development Environment (PrUDE) Reference Guide. Technical Report ECE01-2, Department of Electrical and Computer Eng., University of Victoria, April 2001.

[6] J.-M. Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *Overview of Panel discussion on International Workshop on Industrial Strength Formal Techniques*, Vancouver, Canada, October 22, 1998. panalists: B. Cheng and S. Easterbrook and R. B. France and B. Rumpe.

[7] D. D. Clark and D. R. Wilson. Comparison of Commercial and Military Computer Security Policies. In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, pages 184–195, Oakland, California, USA, April 27-29, 1987.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, August 2002.

[9] O.-J. Dahl and O. Owe. Formal Methods and the RM-ODP. Research report No. 261, March 1998. Department of Informatics, University of Oslo, Norway.

[10] W. Damm and D. Harel. LSC's: Breathing Life into Message Sequence Charts. In *Formal Methods for Open Distributed Systems (FMOODS'99)*, Florence, Italy, February 15-18, 1999.

[11] S. Easterbrook, J. Callahan, and V. Wiels. V&V Through Inconsistency Tracking and Analysis. In *the Proc. of International Workshop on Software Specification and Design*, Ise-Shima, Japan, April 16-18 1998.

[12] S. Flake and W. Mueller. Expressing Property Specification Patterns with OCL. In *The 2003 International Conference on Software Engineering Research and Practice (SERP'03)*, pages 595–601, Las Vegas, NV, USA, June 2003. CSREA Press, Las Vegas, NV, USA.

[13] S. Flake and W. Mueller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Journal on Software and System Modeling (SoSyM)*, 2(3):164–186, October 2003.

[14] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Proc. of Abstract State Machines, Workshop, ASM 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322, Monte Verità, Switzerland, March 19-24, 2000. Springer.

[15] D. Gollmann. *Computer Security.* John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1999.

[16] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[17] CollabNet Inc. ArgoUML: A modelling tool for design using UML, 1999-2002. URL address, http://argouml.tigris.org/.

[18] ISO. A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, September 1988. "ISO Standard 8807".

[19] R. Janicki, D. Parnas, and J. Zucker. Tabular representations in relational documents. In *Relational Methods in Computer Science*, pages 184–196. Springer-Verlag, 1996.

[20] E. B. Johnsen and O. Owe. A Compositional Formalism for Object Viewpoints. In A. Rensink and B. Jacobs, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 45–60. Kluwer Academic Publisher, March 2002.

[21] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods: Dedicated to the Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science.* Springer-Verlag, 2003.

[22] 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.

[23] Anneke Kleppe and Jos Warmer. Extending OCL to include Actions. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

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

[25] B. Liskov and J. Wing. A Behavioral Notation of Subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[26] Klasse Objecten. Octopus: OCL Tool for Precise Uml Specifications.

[27] Dresden University of Technology. Dresden ocl toolset.

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

[29] O. Owe and I. Ryl. The Oslo University Notation: A Formalism for Open, Object-Oriented, Distributed Systems. Report No. 270, August 1999. Department of Informatics, University of Oslo, Norway.

[30] S. Owre, J. Rushby, N. Shankar, and F.V. Henke. Formal Verification for Fault-tolerant Architectures: Prolegomena to the design of PVS. *IEEE Transactions On Software Engineering*, 21(2):107–125, February 1995.

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

[32] D. L. Parnas. Tabular Representation of Relations. Technical Report 260, Department of Electrical and Computer Engineering, Telecommunications Research Institute of Ontario, Communications Research Laboratory, 1992.

[33] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language (OCL) . In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[34] J. Rumbaugh, I. Jacobson, and G. Booch. *The Umified Modeling Language, Reference Manual.* Addison Wesley Longman Inc., 1999.

[35] J. Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. In *FORTE X/PSTV XVII '97: Formal Description Techniques and Protocol Specification, Testing and Verification*, November 1997.

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

[37] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, 2nd edition, 1992.

[38] I. Traoré. An Outline of PVS Semantics for UML Statecharts. *Jounal of Universal Computer Science*, 6(11):1088–1108, 2000.

[39] I. Traoré and D. B. Aredo. Enhancing Structured Review with Model-based Verification. *IEEE Transaction on Software Engineering (to appear)*, April 2004.

[40] I. Traoré, D. B. Aredo, and K. Stølen. Tracking Inconsistencies in an Integrated Platform. Research report No. 274, August 1999. Department of Informatics, University of Oslo, Norway.

[41] I. Traoré, D. B. Aredo, and H. Ye. An Integrated Framework for Formal Development of Distributed Systems. *Journal of Information and Software Technology, Elsevier Science*, 46(5):281–286, April 2004.

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

[43] J. B. Warmer and et al. Response to the UML2.0 OCL RfP, ver. 1.6, OMG Document ad/2003-01-07, January 2003.

[44] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley Longman Inc., 1999.

[45] J. Whittle. Formal Approach to Systems Analysis Using UML: An Overview. *Journal of Database Management*, 11(4):4–13, 2000.

[46] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23:8–24, September 1990.

A Representation of UML Core Package

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Representation of UML Core Package-(Backbone and Relationships)
%% UML v1.3 standard pp. 2-14 and 2-15
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
corePackage : THEORY

BEGIN
  %%%% TYPE DECLARATIONS %%%%%%%%%%
  ModelElement: TYPE+

  Feature, GeneralizableElement, Parameter: TYPE+ FROM ModelElement

  Classifier: TYPE+ FROM GeneralizableElement

  Class: TYPE+ FROM Classifier

  StructFeature, BehavoralFeature: TYPE+ FROM Feature

  Attribute: TYPE+ FROM StructFeature

  Operation: TYPE+ FROM BehavoralFeature

  name: [Feature -> string]

  %%%% TYPE DECLARATIONS Core Package - Relationships
  Relationship, AssociationEnd: TYPE+ FROM ModelElement

  Association, Aggregation: TYPE+ FROM Relationship

  Generalization: TYPE+ FROM Relationship

  source, target: [Relationship -> Classifier]

  acyclic_ax: AXIOM (FORALL (r: Relationship): source(r) /= target(r))

  parameters: [BehavoralFeature -> finite_sequence[Parameter]]

  typeof: [StructFeature -> Classifier]

  precondition, postcondition: [Operation -> bool]

  connection: [Association -> finite_sequence[AssociationEnd]]
  connection_ax: AXIOM
    (FORALL (assoc: Association): length(connection(assoc)) >= 2)

  class_attributes: [Class -> set[Attribute]]

  class_features: [Class -> set[Operation]]

  children: [Classifier -> set[Classifier]]

  parents: [Classifier -> set[Classifier]]

  %%%% TYPE DECLARATIONS: Common Behaviour - Instances and Links
  Object: TYPE+ FROM ModelElement

  null: ModelElement

  classifier: [Object -> Class]

  instance_ax: AXIOM (FORALL (o: Object): classifier(o) /= null)

  class_objects: [Classifier -> set[Object]]

  %%%% VARIABLE DECLARATIONS
  c, c1, c2:    VAR Class
  f1, f2:       VAR Operation

  isActive: [Class -> bool]
  isRoot?(c): bool = (parents(c) = emptyset)
  isLeaf?(c): bool = (children(c) = emptyset)
  isAbstract(c): bool = (class_objects(c) = emptyset)

%% Sets of instances of subclasses are mutually disjoint
  disjoint_ax: AXIOM (FORALL c, c1, c2:
       (children(c)(c1) AND children(c)(c2)) IMPLIES
        empty?(intersection(class_objects(c1), class_objects(c2))))

  unique_names_ax: AXIOM (FORALL c, f1, f2:
       class_features(c)(f1) AND class_features(c)(f2) IMPLIES
        (name(f1) = name(f2) IMPLIES f1 = f2))

  no_mult_parent_ax: AXIOM (FORALL c: singleton?(parents(c)) OR
                                        empty?(parents(c)))
 END corePackage
```

B UML Sequence Diagrams in PVS

The following PVS specification is automatically generated from the UML sequence diagram shown in Figure 6 by using the PrUDE tool. The transformation is based on semantic definitions of UML notations provided in the PVS specification language and implemented in the PrUDE tool. In the current version of the PrUDE tool, application-specific properties are added interactively using the PVS property editor. In the future, we implement several domain specific properties, and proof strategies.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Semantic definition for a partial UML sequence disgram,
%% generated from ArgoUML model using the PrUDE tool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sequenceDiagram[T:TYPE+]: THEORY
BEGIN

s: VAR set[T];
t1,y: VAR T

optional?(s):bool = empty?(s) OR singleton?(s)

optional: TYPE+ = (optional?)
Event : TYPE+
AccessEvent : TYPE+ FROM
Event e,x : VAR Event

Attribute, Operation, Object: TYPE+
Trace: TYPE+ = list[Event]

readCard,openSession,enterPin,readPin,verifyPin,pinOk,
enterChoice,readChoice,enterAmount,readAmount,checkBalance,
balanceOK,provideCash,cashOk,collectCash,updateWithdraw,
ejectCard,collectCard,closeSession,auth: Event

Class:TYPE = [# classID: string,
                attributes:setof[Attribute],
                operations:setof[Operation] #]

t1,t2, t: VAR Trace
n: VAR nat
ae: VAR AccessEvent

prefix_upto(n,t): RECURSIVE Trace =
                  CASES t OF
```

```
                    null: null,
                    cons(e, t2) : IF n=0 THEN null
                                ELSE cons(e,prefix_upto(n-1,t2))
                                ENDIF
                  ENDCASES
                MEASURE length(t)

rank(e,t): RECURSIVE nat = IF NOT member(e,t) THEN 0
                           ELSE CASES t OF
                                null:0,
                                cons(x,t2): IF x=e THEN 1
                                            ELSE 1+rank(e,t2)
                                            ENDIF
                                ENDCASES
                           ENDIF
           MEASURE  length(t)
ax: AXIOM FORALL t,e: member(e,t) IMPLIES
            member(auth, prefix_upto(rank(e,t), t))

SeqDiag : TYPE = [# seqDiagramID : string,
                    objects: setof[Object],
                    traces: setof[Trace] #]
tr: VAR Trace
y: Event
sq: VAR SeqDiag

Message : TYPE = [# name : string,
                    source : Object,
                    target : Object #]

pin_cash_OK(t) : bool = FORALL e : (e = updateWithdraw AND member(e,t))
        IMPLIES (LET prefix = prefix_upto(rank(e,t),t) IN
                    member(pinOk,prefix) AND member(cashOk,prefix))

b, a : VAR nat       %% balance and amount, respectively
cl : nat = 1000      %% a constant Credit Limit
balance_OK(b,a) : bool = b-a >= 0 OR (b-a < 0 AND b-a >= -cl)

thm1: THEOREM FORALL (e:Event, t:Trace):
    (e=collectCash OR e=updateWithdraw) IMPLIES
        ((member(t,traces(withdrawSq)) AND member(e,t)) IMPLIES
         subset({pinOk,balanceOk,cashOk}, prefix_upto(rank(e,t),t)))

END sequenceDiagram
```

C Partial Specification of the Banking System

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  PVS specification for the Banking system
%% generated from ArgoUML model using the PrUDE tool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

bank: THEORY
BEGIN

IMPORTING sequenceDiagram

%%%%%%% DECLARATIONS OF TYPES %%%%%%%
ValueType: TYPE+
ClassID : TYPE+ = string
Event : TYPE+
Trace : TYPE = list[Event]
TransactionKind: TYPE+ = {deposit, withdraw}
LedgerKind : TYPE+ = {drawerLedger, creditLedger, debitLedger}

%%%%%%%%% DECLARATIONS OF CLASSES as TYPES %%%%%%
Transaction: TYPE+ = [# transId: int,
                        transKind: TransactionKind,
                        amount: int #]

Account: TYPE+ = [# accountNum : string,
                    balance : nat,
                    pin : int,
                    trans: list[Transaction],
                    trace : list[Event] #]

Ledger: TYPE+ = [# kind : LedgerKind,
                   trans : list[Transaction],
                   amount : int #]

Bank: TYPE+ = [# accounts: setof[Account],
                 drawer : Ledger,
                 credit : Ledger,
                 debit : Ledger #]

%%%%%%% DECLARATIONS OF VARIABLES %%%%%%
acc, acc1:      VAR Account
tr :            VAR Trace
t, t2:          VAR Transaction
```

```
b, b1, b2:      VAR Bank
l, l1, l2:      VAR Ledger
lt :            VAR list[Transaction]

%%%%%% CONSTRUCTIVE DEFINITIONS OF OPERATIONS %%%%%
acc_bank_ax: AXIOM (FORALL acc,b1,b2:
        accounts(b1)(acc) AND accounts(b2)(acc) IMPLIES b1=b2)

trans_ledger_ax: AXIOM (FORALL l1,l2:
    member(t,trans(l1)) AND member(t,trans(l2)) IMPLIES  l1=l2)

neg(t): Transaction = t WITH [amount:= -amount(t)]

sum_ledger(lt): recursive int = CASES lt OF
                          null: 0,
                          cons(t,lt1): amount(t)+sum_ledger(lt1)
                         ENDCASES
                MEASURE length(lt)

 balanced?(b): bool = sum_ledger(trans(drawer(b)))
                      + sum_ledger(trans(credit(b)))
                      + sum_ledger(trans(debit(b)))= 0

processTrans(t,b): Bank =
   IF transKind(t) = withdraw THEN
    b WITH [drawer:=drawer(b) WITH [trans:=cons(neg(t),trans(drawer(b)))],
      credit:=credit(b) WITH [trans:=cons(t,trans(credit(b)))]]
   ELSE IF transKind(t)=deposit THEN
        b WITH [drawer:=drawer(b) WITH [trans:=cons(t,trans(drawer(b)))],
          debit:=debit(b) WITH [trans:=cons(neg(t),trans(debit(b)))]]
        ELSE b
        ENDIF
   ENDIF

thm1: THEOREM (FORALL t,l: (member(t,trans(l)) AND
                (transKind(t)=deposit OR transKind(t)=withdraw)) IMPLIES
                (EXISTS t2, l2: member(t2,trans(l2)) AND
                                (t2=t WITH [amount:= -amount(t)])))

thm2: THEOREM (FORALL t,b: balanced?(b)=> balanced?(processTrans(t,b)))

END bank
```

D Proof of Theorem thm2

```
 thm2 :

  |-------------------------------------------------------
{1}   FORALL (t, b): balanced?(b) => balanced?(processTrans(t, b))

Trying repeated skolemization, instantiation, and if-lifting, then
Expanding the definition of sum_ledger, and then Expanding the
definition of processTrans(), this simplifies to: thm2 :

{-1}  CASES trans(credit(b!1))
        OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
        ENDCASES
       +
       CASES trans(debit(b!1))
         OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
         ENDCASES
       +
       CASES trans(drawer(b!1))
         OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
         ENDCASES
       = 0
  |-------------------------------------------------------
{1}   CASES IF transKind(t!1) = withdraw THEN cons(t!1, trans(credit(b!1)))
            ELSE b!1`credit`trans
            ENDIF
        OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
        ENDCASES
       +
       CASES IF transKind(t!1) = withdraw THEN b!1`debit`trans
             ELSE cons(neg(t!1), trans(debit(b!1)))
             ENDIF
         OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
         ENDCASES
       +
       CASES IF transKind(t!1) = withdraw
               THEN cons(neg(t!1), trans(drawer(b!1)))
             ELSE cons(t!1, trans(drawer(b!1)))
             ENDIF
         OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
         ENDCASES
       = 0

Lifting IF-conditions to the top level,
thm2 :
```

```
{-1}  IF null?(trans(credit(b!1)) THEN
           (0 + (CASES trans(debit(b!1))
                OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
                ENDCASES)
              +
              (CASES trans(drawer(b!1))
                OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
                ENDCASES))
              = 0
      ELSE amount(car(trans(credit(b!1)))) +
            sum_ledger(cdr(trans(credit(b!1))))
            +
            CASES trans(debit(b!1))
              OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
              ENDCASES
            +
            CASES trans(drawer(b!1))
              OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
              ENDCASES
            = 0
      ENDIF
  |-------------------------------------------------------
{1}   IF transKind(t!1) = withdraw
        THEN CASES cons(t!1, trans(credit(b!1)))
               OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
               ENDCASES
              +
              CASES b!1`debit`trans
                OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
                ENDCASES
              +
              CASES cons(neg(t!1), trans(drawer(b!1)))
                OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
                ENDCASES
              = 0
      ELSE CASES b!1`credit`trans
             OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
             ENDCASES
            +
            CASES cons(neg(t!1), trans(debit(b!1)))
              OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
              ENDCASES
            +
            CASES cons(t!1, trans(drawer(b!1)))
              OF null: 0, cons(t, lt1): amount(t) + sum_ledger(lt1)
              ENDCASES
            = 0
      ENDIF

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of thm2.

Q.E.D.
```

E Association and Inheritance in UML

```
inheritance : THEORY

BEGIN
 %% IMPORTING
   IMPORTING bank
   IMPORTING corepackage

   %% TYPE DECLARATIONS - Inheritance
   Inheritance : TYPE+ FROM Relationship

   c1, c2 : VAR Class
   i: VAR Inheritance

   inh_ax: AXIOM (source(i)= c1 AND target(i)= c2 IFF
                children(c2)(c1) AND parents(c1)(c2))

   %%% DECLARATION CLASS Person AND ITS SUBCLASSES

   Person: TYPE+ FROM Class
   Customer : TYPE+ FROM Person
   Employee : TYPE+ FROM Person

   %%%% SOME VARIABLE DECLARATIONS %%%%%%%
   b :                     VAR Bank
   acc, acc1, acc2 :       VAR Account
   p, p1, p2 :             VAR Person
   c :                     VAR Customer
   e:                      VAR Employee

   %%%%% DECLARATION OF ASSOCIATIONS %%%%%%%%%%%
   owns : [Person -> set[Account]]
   updates : [Person -> set[Account]]

   uses : [Bank -> set[Person]]
   worksfor : [Bank -> set[Person]]

   %%%%% AXIOMS %%%%%%%%%
   uses_ax: AXIOM (FORALL p,b: uses(b)(p) IFF
       (EXISTS acc:  accounts(b)(acc) AND (owns(p)(acc) IMPLIES
                           NOT updates(p)(acc))))

   worksfor_ax: AXIOM (FORALL p,b: worksfor(b)(p) IFF
       (EXISTS acc:  accounts(b)(acc) AND (updates(p)(acc) IMPLIES
                           NOT owns(p)(acc))))
   %%% An employee is not allowed to update his owns account
   emp_cust_ax: AXIOM (FORALL e,b,acc: (uses(b)(e) AND worksfor(b)(e))
        IMPLIES intersection(owns(e), updates(e)) = emptyset)

   %%% Declaration of {xor} constraint as an axiom
   xor_ax: AXIOM (FORALL p,acc: NOT (owns(p)(acc) IFF updates(p)(acc)))

   thm6: THEOREM (FORALL p,b,acc: (worksfor(b)(p) AND uses(b)(p))
            IMPLIES NOT (owns(p)(acc) IFF updates(p)(acc)))

END inheritance
```

F Proofs of Theorem thm6

```
thm6 :
  |--------------
  {1} (FORALL p,b,acc: (worksfor(b)(p) AND uses(b)(p)) IMPLIES
                NOT (owns(p)(acc) IFF updates(p)(acc)))

Rule? (grind :theories ("inheritance"))

Trying repeated skolemization, instantiation, and if-lifting, this
yields  2 subgoals:
thm6.1 :
    {-1} GeneralizableElement_pred(p!1)
    {-2} Classifier_pred(p!1)
    {-3} Class_pred(p!1)
    {-4} Person_pred(p!1)
    {-5} owns(p!1)(acc!1)
    {-6} updates(p!1)(acc!1)
   |--------------
Rule? (postpone) Postponing thm6.1

thm6.2 :
    {-1} GeneralizableElement_pred(p!1)
    {-2} Classifier_pred(p!1)
    {-3} Class_pred(p!1)
    {-4} Person_pred(p!1)
   |--------------
    {1} owns(p!1)(acc!1)
    {2} updates(p!1)(acc!1)
Rule? quit

Run time  = 1.45 secs.
Real time = 50.58 secs.
```

The two subgoals `thm6.1` and `thm6.2` generated are not provable. Hence, to prove the theorem we need to add an axiom (see section 3.4 for details). The following is a successful proof of theorem `thm6`.

```
thm6 :
  |-------
  {1}   (FORALL p, b, acc:
          (workers(b)(p) AND workers(b)(p)) IMPLIES
           NOT (owns(p)(acc) IFF updates(p)(acc)))

Trying repeated skolemization, instantiation, and if-lifting, this
completes the proof of thm6.

Q.E.D.
```