2003-09-00

Evaluation of Program Specification and Verification Tools for High Assurance Development

Ubhayakar, Sonali; Bibighaus, David; Dinolt, George; Levin. Timothy E.

Naval Postgraduate School (U.S.)

Proceedings of the International Workshop on Requirements for High Assurance Systems, Monterey, CA, September 2003, pp. 43-47.
http://hdl.handle.net/10945/7117

Evaluation of Program Specification and Verification Tools for High Assurance Development

Sonali Ubhayakar
First author's affiliation
1st line of address
2nd line of address
Last line, including country
+1
1st author's email address

David Bibighaus
Naval Postgraduate School
833 Dyer Rd.
Spanagel Hall Rm. 401
Monterey, CA 93943 USA
+1 831 641 9391
bibighaus@acm.org

George Dinolt
Naval Postgraduate School
Mail Code: CS/
Department of CS
Monterey, CA 93943 USA
+1 831 656 3889
gwdinolt@nps.navy.mil

Tim Levin
Naval Postgraduate School
Mail Code: CS/
Department of CS
Monterey, CA 93943 USA
+1 831 656 2239
televin@nps.navy.mil

ABSTRACT
A key decision in the development of high assurance software is that of choosing a formal methods tool. This paper describes a methodology to select a formal methods tool for use in the development of high assurance software. Some of the factors that make a tool suitable to the task can be evaluated with a desk check, while others can only be appreciated by "hands on" testing. We describe the application of our methodology to a broad set of currently available formal methods tools, including a hands-on evaluation of one of the tools. The impact of the tools on the project development is also discussed.

Keywords
Formal specification, formal methods, high assurance

1 INTRODUCTION

A distinguishing feature of high assurance systems is that they are modeled mathematically using formal methods. This modeling enables formal reasoning about the system design, that can be used to prove that the system has certain properties. The model also assists in the verification of the code since all code must be shown to be an instance of a part of the model.

A key consideration in the design of high assurance systems is the choice of the software used to assist in the formal modeling. The question arises as to how to choose a tool to assist in the formal methods and on what factors should that decision be based. This paper describes an effort to identify discriminators that will allow practitioners to find the best fit of formal tools for a project.

Statement of Need

There are over a dozen freely available formal method tools that that can assist with formal methods of software design. Many of these were compared in an initial survey. The survey included ACL2, AutoFOCUS, Coq, Elf/Twelf, IMPS, HOL, Isabelle, Nuprl, Otter, PVS, SpecWare, STeP, TAME, TPS, Maude, Vienna and Z/Eves.

This study [7] was conducted at the Center for INFOSEC Studies and Research (CISR) at the Naval Postgraduate School, as part of the TCX project. [4] The goal of this project is to develop a high assurance separation kernel that can be evaluated under the Common Criteria at EAL-7. It is the hope of CISR that the code and supporting development documentation can be used as a concrete example for others in the development of high assurance systems. This project will be done using a spiral development method.

Thus the need was for a tool that would assist in the expression and validation of security properties. This tool must be easily obtainable and freely available so that reviewers can understand the role of formal methods in the development of high assurance systems.

2 METHODOLOGY

We developed a two-phase methodology for evaluating formal methods tools. The first phase is to conduct a desk check of the tools against a set of functional criteria or requirements. These requirements are used to determine a smaller set of tools that can be compared in-depth. The second phase, the in-depth evaluation, is to conduct a hands-on trial of the remaining tools. Each tool is used to specify and prove the security characteristics of a simple security model.

Evaluation Criteria

Before the evaluation began, a list of criteria was identified to evaluate the tools. While developing the set of criteria, it became apparent that some of the criteria could only be evaluated by using the tool. These criteria would be the basis for evaluation during the "hand's on" phase. *Table 1* summarizes the criteria used to evaluate the tools and the phases in which the criteria were examined.

3 SURVEY PHASE

A large effort was undertaken to compare the tools against the criteria. In the course of conducting the survey, we found that several of the criteria were more effective in distinguishing among the tools.

Some of the criteria appeared useful in establishing minimum functional requirements, but proved to be of little, if any, value in discriminating one tool from another, since all of the surveyed tools met these basic standards. For example, *Resource Requirements* and *Implementation Language* did not prove useful in reducing our tool set. All of the tools evaluated ran on at least one of the popular operating systems and used a well-known and common implementation language.

	Evaluation Criteria	Qualifications
Survey Phase	Maturity	Ideal tool should be mature enough to be trusted, and actively supported.
	Documentation	Ideal tool should have a large body of resources and documentation.
	Purpose	Tool must be able to reason about security properties. Examples of using the tool in a software development process must exist.
	Implementation Language	Ideal tool will use a language to describe the model that is portable and "well known" with ample documentation
	Resource Requirements	Tool should run on either Windows or a version of Unix (Linux, Solaris, OS X.). Ideal tool will work on multiple platforms.
	Expressiveness	Tool must be able to express software properties and be able to prove that the specification meets those properties.
	User Interface	Ideal tool should not require the memorization of a large body of commands. A simple GUI is preferred.
	Consistency of Specification	Logic of the tool must be shown to be consistent. Tool must guarantee that a specification is consistent ether by construction or by consistency checking.
	Executable Specification	Ideal tool should allow a specification to be executed to demonstrate certain cases and allow the developer to gain an interactive "feel" for the program
"Hands On" Phase	Multiple Levels of Abstraction	Tool must support a "top down" design process where a specification can become less abstract. Tool must support inter-level mapping or some other way of demonstrating correct traceablility of requirements between levels.
	User Friendliness	Ideal tool should be useable without first acquiring a detailed understanding of the mechanics of the tool.

Table 1: Evaluation Criteria

Factors that were useful in discriminating one tool from another in the survey phase were further divided into primary and secondary groups. The primary discriminators are *Support*, *Maturity*, *Expressiveness* and *Purpose*; the

remaining factors are deemed "*secondary*." The rational for this division was that, since our project was focused on developing very high assurance software, the correctness of the proofs was paramount and therefore it would be unwise to choose an immature tool that had not been thoroughly scrutinized for potential errors in its internal logic. Since the ultimate purpose of the TCX project was not to advertise a tool, but rather to construct software, the availability of supporting documentation was critical. Expressiveness was desirable to support the maintainability of the specifications and the mapping of them to actual code. Finally, we sought tools that were developed as general-purpose theorem provers since they were the most flexible and imposed the fewest restrictions on development.

Survey Conclusions

Out of the sixteen tools surveyed, two were selected for an in-depth analysis for their suitability to our project. The two tools were ACL2 and PVS. A comparison of the two tools follows in *Table 2*.

4 HANDS ON PHASE

One of the major goals of the TCX project is to show, with high assurance, how an abstract security policy is implemented in the software and hardware. To achieve this, we plan to use the following approach:

> A model of the security policy is created using a formal specification language. At this level, the model is checked for internal consistency and for its ability to capture the salient properties of the security policy. The model of the security policy then is "refined" to capture more details of the implementation. This refinement is often called an FTLS (Formal Top Level Specification). The refinement should be expressible in the terms of the specification language, there should be support for the mapping between the refined specification and the security policy and there should be support for proving that the refined specification is consistent with the security policy and does not implement anything that would "violate" the properties of the model.

Having narrowed down the field of tools, it is possible to begin using the tools to determine their applicability to the project. For the tool to support our approach, it should be

Criteria		ACL2 Adapted Common LISP 2	PVS Prototype Verification System
Primary Criteria	Maturity	Editors Kaufmann and Moore. Current version 2.7 released in 2002. First version developed in 1994. Based on the Boyer Moore Theorem Prover that dates to the 1970's.	Developed by SRI International Current version 3.1 released in 2003. First version developed in 1992.
	Support	Since ACL2 is a subset of Common LISP, there is extensive supporting documentation available	One of the oldest and best documented of the theorem provers that is in use today.
	Expressiveness	Blends arithmetic decision procedures with rewriting techniques. Uses first-order quantifier-free logic. Has Powerful type-like mechanism called "guards" that can be used to ensure functions are well typed. Logic is quantifier free semi-automatic and uses lemmas as guidance. [3]	Supports classic higher-order logic with functions, sets records, tuples, predicate sub-types, dependent typing and theories. Axioms may be introduced freely. Allows concise and natural specifications as well as the ability to generate human readable proofs [3][6]
	Purpose	General purpose theorem prover	General purpose theorem prover
Secondary Criteria	User Interface	Emacs	Emacs with X window tools.
	Consistency of Specifications	Yes, through built-in interactive "proof checker" [3]	Yes, Automated consistency checking of specifications. Type checking system that allows the use of arbitrary axioms during development.
	Executable Specifications	Yes, unless there exist undefined functions.	No
Non-discriminator	Resource Requirements	Any environment that supports one of the standard Common Lisp Implementations, Unix, Linux, Windows, etc.	Sparc machines with Solaris 2 or greater and x86 machines with Linux distributions compatible with Redhat 5 or later.
	Implementation Language	Untyped Common Lisp	Allegro Common Lisp (Commercial Version of the Language)

Table 2: Candidate Tools for In-Depth Evaluation

able to provide the mappings between the security model and the FTLS.

Approach

The model we used is a simplified instance of the Bell and LaPadula Model [1]. The top level represents the Security Policy, and the second is a Formal Top-Level Specification. The security policy describes a hypothetical state machine with some basic state transitions. Within the machine there are subjects (actors that use or manipulate data) and objects (representing the data itself). Every subject and every object had an associated security label. These labels do not change and are ordered so that label $a \geq b$, implies that a is more sensitive than b. A real world example would be the U.S. military labels *Secret* and *TopSecret* where *TopSecret* ≥ *Secret*. In plain English, the security policy states that a subject cannot read an object unless the subject's security label is greater-than or equal-to the security label of the object. Similarly, a subject can only write to an object if the subject's security label is equal to the security label of the object.

The fundamental security property is that that the system is in a secure state if and only if subjects can only access objects in accordance with the policy described above. The security theorem we need to prove is that if the system starts in a secure state and only makes transitions according to our restrictions then it will always be in a secure state.

This policy was chosen because it is well known, simple enough that it can be proven easily and has enough elements in it to provide a glimpse as to how well the tool would support a more complicated policy.

The Security Policy states very few details about the machine, subjects, objects, security labels or transitions. The second level of the model, the Formal Top-Level Specification describes some of these in more detail. The Formal Top-Level Specification represents a particular implementation of the security policy. The FTLS shows the actions of the system including inputs, outputs and effects (including errors). In the FTLS, we assign more concrete descriptions to the elements of our policy. Subjects, for example, are described as processes, while objects are represented as memory blocks.

Implementation of the Model

We implemented the two-level model in PVS. PVS is a

type-theoretic specification language. An element is of a specific *type* if it satisfies some condition (for example a being may be of type *Human* if it walks on two legs and does not have fur). Type theory is similar to set theory in many respects but differs primarily in the sense that two sets are considered equal if every element is identical, whereas stating the types of two elements are equal says nothing about elements themselves but does describe their structure [2]. A SubType is a type with additional constraints imposed on the type (For example a Type *Man* may be described as a predicate subtype of *Human* that meets the additional predicate male(x)). Within PVS and other type-theoretic proof checking systems, checking that types are used appropriately becomes an important obligation in the overall proof (for example, assuming a proof exists that being a *Man* implies being mortal, proving that Socrates is mortal is reduces to proving that Socrates was of type *Man*). For our two level model, the primary challenge of the inter-level mapping was type-checking that the appropriate FTLS elements were a sub-type of the Security Policy elements.

PVS provides three important features that were key to the development of the specification: TYPE+, Sequences and IMPORTING. The TYPE+ notation indicates that an abstract type is non-empty, with no additional constraints on it. Sequences are a type in PVS, and are used to talk about series of events for inputs or outputs. The IMPORTING clause allows the use of theories from other (presumably higher level) specifications. Our implementation made use of all of these items. After much work, we were able to successfully implement our model and prove the security properties. A summary of the elements and their mapping is shown in *Figure 1*.

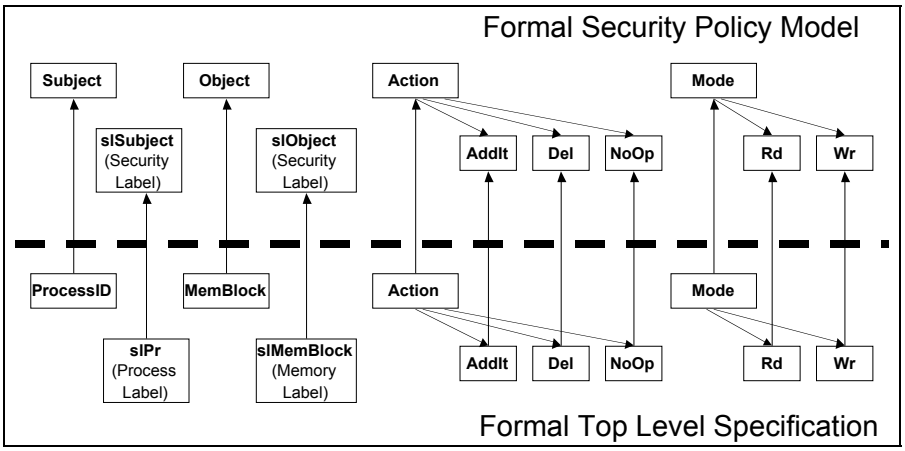

Figure 1: Intra-Level Mapping

Evaluation of the PVS Implementation of the Model

After specification and proof of the model, we were able to draw several conclusions about PVS. The first was that the specification could be written concisely. The use of higher order logic and the type system enabled a specification that followed the original model closely and was comprehensible by a human reader. As is usual in any system that has "strong" typing, it took a considerable amount of effort to ensure that the various types had the appropriate relationships with each other. The automated type checking and the verification system were very useful in identifying when there were mismatches and validating the final results. For example, the type checking system generated several proof obligations that were missed in the initial construction of the specification; such as the fact that Read _ Write and that Memory could not be empty.

One problem that was encountered was the use of the IMPORTING clause. PVS currently does not allow embedded assumptions in the importing parameter list. This prevented automated generation and proving of some of the Type-Checking obligations. Therefore the assumptions at the Policy had to be repeated at the FTLS level. This made the FTLS proof more complicated and reduced the benefits of the layering.

Evaluation of ACL2

ACL2 is a subset of common LISP. The goal in the construction of ACL2 was to retain the power of LISP while removing the commands that introduced side effects that could harm the integrity of proofs[5]. It can be used not just as a specification or modeling language but also as a programming language. ACL2 is based on set theory, however it attempts to derive some of the benefits of a type-based system through the concept of guards. A guard prevents a function from accepting a parameter unless it meets a condition.

We have yet to implement the demonstration security model and FTLS in ACL2. We will be able to do a more complete analysis after that has been completed. It appears that since ACL2 is not strongly typed, it may require more effort than PVS to express the same concepts. A simplified proof process may offset this additional effort[8]. Finally, ACL2's logic provides several automatic proof techniques. The style of these proofs is very robust and dramatic changes to the specification may not imply a dramatic change to the proof.

5 CONCLUSIONS

The selection of a formal methods tool is an important decision in the development of a high assurance system. We have defined a useful methodology for the assessment of formal methods tools. Included in this methodology is a set of evaluation criteria, or requirements. Several factors must be considered including the system's maturity,

support, expressiveness and multi-level mapping. While several tools may meet the general requirements, differences in the foundational logic will greatly shape how the specification is developed, how the theories are proved and how the specifications are layered, and will thus effect the overall assessment of a given tool.

REFERENCES

1. Bell, D.E. and LaPadula, L.J. *Secure Computer Systems: Mathematical Foundations and Model.* M74-244, The MITRE Corp., Bedford MA, May 1973.

2. Constable R. L., Allen S. F., Bromley H. M., Cleaveland W. R., Cremer J. F., Harper R. W., Howe D. J., Knoblock T. B., Mendler N. P., Panangaden P., Sasaki J. T., Smith S. F. *Implementing Mathematics with The Nuprl Proof Development System*. Cornell University Ithaca NY, 1986.

3. Kolhase, M. Database of Existing Mechanized Reasoning Systems. <http://www-formal.stanford.edu/clt/ARD/systems.html>. June 1999.

4. Irvine, Cynthia E., Levin, Timothy E., Dinolt, George W. "HASP Trusted Computing Exemplar", Naval Postgraduate School Technical Report NPS-CS-02-004, September 2002.

5. Moore and Kaufmann. *ACL2 Version 2.7 Homepage*. <www.cs.utexas.edu/users/moore/acl2/acl2-doc.html>.

6. SRI International. *PVS Homepage*. <pvs.csl.sri.com>.

7. Ubhayakar, S. *Evaluation of Program Specification and Verification Systems* Masters Thesis, Naval Postgraduate School, Monterey, California June 2003.

8. Young, William *Comparing Verification Systems: Interactive Consistency in ACL2*. <www.cs.utexas.edu/users/moore/publications/others/interactive-consistency-young.ps> 1996.

9. Zhang, Wenhui. *Evaluation of Verification Tools* <www.ifi.uio.np/~adapt/adapt-ft-05.ps.gz>