Unification of Verification and Validation Methods for Software Systems: Progress Report and Initial Case Study Formulation

James C. Browne, Calvin Lin, Kevin Kane
Department of Computer Sciences
University of Texas at Austin
Taylor Hall – MS C0500
Austin, Texas 78712 USA
{browne,lin, kane}@cs.utexas.edu

Yoonsik Cheon, Patricia Teller
Department of Computer Science
University of Texas at El Paso
500 W. University Avenue
El Paso, Texas 79968 US
{cheon,pteller}@cs.utep.edu

Abstract

This paper presents initial research on unification of methods for verification and validation (V&V)of software systems. The synergism among methods for V&V are described. The requirements for a unification are defined. The initial steps of a case study of application of the unified approach to V&V is sketched including definition of the problem domain, the approach and some details of a property specification language. An undergraduate course introducing the unified approach to V&V is described. The relationship of this research to other efforts toward unification of V&V are discussed.

1. Introduction

1.1 Status of Verification and Validation:

The verification and validation (V&V) practice for software systems is, with few exceptions, typically approached through informal methods and tools or occasionally approached through the application of formal proof methods. Neither approach alone is complete or satisfactory, and seldom are the methods integrated effectively. The situation is even less satisfactory in university curricula. Incredibly, few computer science or computer engineering programs have comprehensive coverage of methods and tools for V&V, and even fewer have attempted courses that present a unified perspective of V&V methods. Instead, the many aspects of V&V are typically taught in isolation using language-specific tools. Moreover, the role of software design in facilitating V&V is often not emphasized.

1.2 Synergism Among Verification and Validation Methods:

Testing, model checking, static analysis, theorem proving, and runtime monitoring are related in many ways. Model checking can be viewed as systematic testing of abstract states based on formal specifications. Indeed, much research in testing focuses on path coverage and the elimination of infeasible paths. Static analysis can be viewed as an approximation of model checking that is less precise but more scalable. Many of the pre- and post-conditions used in theorem proving can be expressed in temporal logic and verified by model checking or validated by runtime monitors. These techniques also have complementary strengths and weaknesses. Testing and runtime monitoring are necessarily unsound, while the other techniques can be sound, complete, or neither.[1]

Both testing and runtime monitoring can be guided by static analysis, and static analysis is a critical enabling technology for model checking, as it can project out states and behaviors that are irrelevant to the property being checked. Static analysis also can be used to generate code for runtime monitoring. On the other hand, runtime monitoring can be applied in cases where model checking is intractable and static analysis is imprecise. Finally, theorem proving has been used to improve the precision

[1] A technique is sound if it never verifies as correct a program that is incorrect, i.e., it reports all errors. A technique is complete if it always verifies as correct all programs that are correct, i.e., it never reports a false-positive.

of model checking. The relationship between model checking and static analysis is worth particular attention. Both model checking and static analysis are state space analysis techniques that can verify typestate properties [13], but the former explores each execution path in a depth-first manner while data-flow analysis produces an approximate answer by merging results at various control flow merge points. Thus, model checking can be viewed as path-sensitive data-flow analysis, and data-flow analysis can be viewed as a method of approximating model checking. In gross terms, the two approaches represent different tradeoffs between precision and scalability. Model checking and static analysis have traditionally operated on different program representations, but even these are converging as recent work in BDD-based pointer analysis has used symbolic representations to greatly improve scalability [15, 3]. A key aspect of this project is to further develop the synergies between these two approaches. Finally, theorem proving has also been integrated with other verification methods [9].

1.3 Opportunity:

Given the potential synergies of testing, program analysis, and model checking, and given the increasing maturation of these techniques, it is now feasible to intergrate these complementary approaches to V&V. It is therefore logical to organize, structure, and present a comprehensive approach to V&V that is language-independent, although tools will always be language-specific. This knowledge can then be used to select methods and tools for V&V of a particular programming paradigm or language.

It is also noteworthy that the hardware development community has begun to unify verification by integrating simulation and model checking techniques [1].

1.4 Definitions:

There are numerous definitions in the literature of Verification and Validation (V&V). The following are from the IEEE standards.

- *Verification* – "Confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled."
- *Validation* – "Confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled."

In the context of software development, verification is the activity that ensures the work products of a given phase fully implement the inputs to that phase, or "the product was built right." Validation, in its simplest terms, is the demonstration that the software implements each of the software requirements correctly and completely, i.e., the "right product was built."

In the context of this proposed research, V&V means the determination that a software system meets its specifications beginning with analysis and design and continuing throughout implementation. The focus is on a comprehensive approach to determining that the specifications are met.

We believe that while in an ideal world software would be specified and developed with formal and rigorous methods, it is also important to enhance the effectiveness of V&V for conventional software development methods. Therefore, conventional software development methods as typically taught in computer sciences departments are the targets for the integrated approach to V&V proposed herein.

The following informal definitions give perspective on our approach to integration.

- *Testing* – Determines the correctness of the execution of a program for a given initial condition and input set.
- *Static Analysis* – Determines program properties such as data-flow paths and control flow paths that can be deduced from the static structure of the program.
- *Model Checking* – Determines the correctness of a temporal property for the executions of a program for all initial conditions and inputs.
- *Formal Proof* – Determines whether a program conforms to a specification of behavior, usually an input/output relation for all executions of the program.
- *Runtime Monitoring* – Dynamically checks whether the execution of a program conforms to a specified condition.

1.5 Motivation:

Computers are increasingly assuming central roles in safety- and security-critical systems, leading to dire consequences of viruses, worms, and software faults. Almost all of these viruses, security attacks, and equipment malfunctions are due to flaws in software design and implementation that could have been found by a truly comprehensive and well-structured process to verify and validate the properties and behaviors of the software. Additionally, there are specifications for information flow, which are sometimes called security policies, and the design and implementation of these security policies also must be verified and validated. The methods needed to verify and validate the security policies largely overlap with those needed to verify and validate other types of specifications.

This work is further motivated by the increasing role of concurrency and parallelism in embedded and control systems. Conventional testing of concurrent, parallel, and distributed systems remains relatively weak despite considerable research. Model checking, on the other hand, is naturally suited to the verification of such systems.

1.6 Paper Contents

The balance of the paper is organized as follows. Section 2 gives the requirements for a unification of V&V for software. The initial case study where the unification is being piloted is defined and described in Section 3. Section 4 mentions some related work that has influenced our approach. The content for an undergraduate course on the unified approach is given in Section 5. Future research, both short term and long term, are sketched in Section 6.

2. Requirements for Unification

The requirements for an effective unification include:

- A unified property specification language incorporating provision for representation of domain specific properties,
- A taxonomy for classification of properties with respect to applicability of verification and validation methods, and
- An abstraction/translation capability for translation of properties in the "universal" PSL to the property specification languages of method specific tools and generically specified system models into representations for language and method specific tools.
- A software design model which enables compositional reasoning over the entire system.

2.1 Unified Property Specification Language:

The unifying conceptual element for unification is a "universal" property specification language that spans properties that can be validated by testing or runtime monitoring or are verifiable by static analysis, model checking, or proof methods. This unification enables a systematic approach whereby properties can be verified by static analysis, testing, or model checking or be compiled to runtime monitors as appropriate or required. This comprehensive view of system behavior also serves as a powerful tool for documentation. The only example of a unified property specification language with which we are familiar is the Property Specification Language (PSL) [1] developed by the Acellera Consortium for simulation-based testing and model-checking-based verification of hardware systems. The Acellera PSL is a general version of a future time temporal logic which can be translated to several well-known forms of temporal logic. The Acellera PSL has domain specific operands and operators to facilitate formulation of properties of clocked hardware systems. For example, it incorporates a clock that can be modified for use in specification of performance properties.

2.2 Property/System Classification:

The appropriate choice of method and tool to verify or validate a given property is dependent upon both the property and the system. There currently exists no systematic method for mapping property/system characteristics to the most effective methods or tools for the property system pair. The classification will require characterizations of properties, systems and methods and tools in operational terms.

2.3 Unification of Method/Tool Implementations:

The unifying implementation technology of the proposed approach is integration of static analysis, testing including coverage analysis, translation/abstraction of the program to a model checkable representation, and generation and insertion of runtime-monitoring code based on the property specification language. Model checking is based upon translation/abstraction of the software representation to a model-checkable representation. The translation/abstraction infrastructure will be extended to verify static properties specified in the property specification language. Runtime monitoring will be accomplished by source-to-source transformation of the software representation based on the program analyses built into the translation/abstraction and program analysis infrastructure.

2.4 Software Design Model

Rigorous componentization is the software design model used in this study. The example system and the property specification language are based on self describing components. A self-describing component has its properties specified in its interface. This enables compositional reasoning by enabling representation of components by their properties in the verification of compositions of components.

3. Initial Case Study for UVV: Access Control for Distributed Systems of Services

The content of subsections 3.1, 3.2, 3.3, 3.4 and 3.7 are summarized from Kane and Browne [8]. More detail can be found in [8].

3.1

Access control systems are typically formulated in three stages: formulation of a policy, representation of that policy as a scheme, and finally realization of that scheme in an implementation. An access control policy is a definition of how a system should provide or deny access which can range from an abstract statement like, “only users on this list should have access,” or “only users who have given me service in the past should have access,” to programs in policy languages with executable semantics. An access control scheme, as defined by [14], is a state transition system in which access control decisions are specified as changes of state in an appropriate representation such as an access control matrix. A set of access control schemes is an access control model.

3.2 System Architecture

A distributed system of services is constructed from a set of nodes, each of which hosts one or more services. Services are active entities which provide their own access control as opposed to passive resources which do not. The nodes (services) are connected by a network that provides bi-directional communication and a mechanism for discovering other services. Each service interacts with other services through an interface, which is the portion of the service exposed to the network. An interface is a set of operations, which are individual units of functionality invoked by users. In the context of an interaction between two services, we refer to the user as the service invoking the operation, and still refer to the service as the service providing the operation. Human operators are abstracted as services which provide no operations, and so only ever play the part of users. We split the interface into the interface mediating requests made to the service, as well as requests made by the service.

We assume there are cryptographic primitives available for establishing private channels between two services, and also for guarding protected objects against tampering.

3.3 Formulation of Access Control for Distributed Systems of Services

The access control problem is then specified as a state transition system where:

- The states and state transitions are for an access matrix and an associated trust matrix.
- Trust matrix entries are trust values held by services about other services.
- Transitions in access control matrix result from granting and delegation of contractually limited capabilities which are defined following.
- Transitions in trust matrix result from yet to be defined trust computations.

Since the state of the system will be distributed across the set of services comprising the system, the information content of the access control matrix will be distributed across the services. This means that access control enforcement cannot always be by prevention but may be by detection and update of the trust matrix.

Access control is designed and implemented in terms of contractly limited capabilities (CLC’s). A basic capability [12] is a self-validating credential that provides access to a resource. It is both a reference to a resource and a set of access rights on it. Possession of a capability implies authorization, and so the access control decision is only validating the capability. For example, in an operating system on a single host, a user holds a set of capabilities to files in the file system in a special memory segment, and presents a capability to the kernel in order to execute a desired operation on a file. These capabilities can then be copied to other users to delegate all or part of a user’s authority to another. Capabilities are kept in tables, where they are mapped to particular functions or operations, so that validation is mapping it to a valid operation; if no such mapping exists, the capability is by definition invalid. In this way, verification is reduced to function/operation table look-up.

The properties to be established are that grants of capabilities conform to the stated policies in terms of values in the trust matrix and the access matrix and that access to the operation for which a capability is granted is accepted or declined in conformance to the stated policies in terms of the values in the access matrix and the trust matrix.

3.4 Contractually-Limited Capabilities

Definitions

A contract is a set of access rights on the invocation of a capability by the user[2]. It is represented as a (possibly stateful) function evaluated by the service at the point of invocation. Input to this function is operation-dependent, but consists of the capability and parameters used in the invocation, together with any state the operation is programmed to use in its decision. The function then returns true or false to indicate whether the invocation should be serviced.

- A contract can specify any condition expressible in the programming language in which the contract is written.

A contractually-limited capability is a 6-tuple (I, O, K, C, P, S), where:

- I is a pointer into the operation table of the service component, represented as an integer,
- O is a string containing the human-readable name of the operation as provided by the service,
- K is the certificate of the service,
- C is the contract,,
- P is the list of parameter types for the operation, and
- S is the cryptographic signature computed across the other five fields, and signed with the private key corresponding to K.

When invoked, a contractually-limited capability is satisfied if and only if:

- I is a valid pointer into the operation table,
- K is the certificate of the service where this capability is invoked,
- The arguments provided are of the correct types as listed by P ,
- S is the correct signature as computed across the other five fields, and
- Evaluation of C returns true.

These conditions are to be expressed in the property specification language and verified by the verification and validation process.

3.5 Formulation of a Unified Property Specification Language

The UPSL will be based on integration of BAN [4] logic for reasoning about beliefs held by one service concerning another service with past time and future time temporal logic [5] for reasoning about formulas over states of the access matrix and domain specific base terms for capabilities and services.

3.6 Verification and Validation Methods

Establishment of trust entries in the trust matrix will be by manual proofs on properties formulated in BAN logic. Verification of properties on the states of the access matrix will be of properties formulated in the integration of past time and future time temporal logic and will be accomplished either by model checking of the interactions of the state machines defined by the interaction protocols of the interacting services or by compiling the past time logic properties to runtime monitors to be validated on execution traces at services.

3.7 Implementation

Distributed systems of services are implemented in an extended version of the CoorSet [7] coordination language system where the compiler and runtime system provide for independent components to work with one another through associative interactions [7]. Associative interactions are interactions where messages are addressed to descriptions of receiver sets, and receivers are described not by an arbitrary name but by an application dependent descriptive name. They are so named due to their modeling after the addressing of associative memory. These interactions are mediated through two interfaces on each service: an accepts interface, describing the functionality provided by the service, and a requires interface, describing the functionality required by the service. We also add a third interface, the capability interface, which is the internal mapping of capabilities to internal functionality, and is consulted when an invocation is made with a capability. Entries are added to this interface most often by a

[2] A more complete implementation of contracts including obligations on the service and were as obligations on use of the capability will be incorporated in future research.

successful handshake, although application routines may create and extend capabilities as well. Space precludes a detailed presentation of the extended CoorSet [7] (CapCoorSet [8]) system. Details are given in the referenced papers.

3.8 Status

The logic upon which the property specification language is based has been formulated. The grammar for the translation system for the property specification language is being developed. An example distributed system of services has been coded and tested. Examples of properties for verification have been formulated.

4. Related Research

Each of the methods for verification and validation has an extensive (if not vast) literature. The research discussed here is confined to the relatively sparse literature on integration of multiple methods of V&V. There are numerous papers that address some degree of integration but literally none that we have been able to find that propose a unification and integration across all of static analysis, testing, model checking and runtime monitoring. For example, Richardson and Clarke addressed integration of testing and verification in 1985 [11]. There have been some research integrating various methods in pairs, for example Kuncak, et.al [9] discuss combining theorem proving with static analysis. There is considerable literature combining testing and runtime monitoring. McHugh in his dissertation [10], combined runtime monitoring with formal proofs. The Java PathFinder [6] project integrates formal specification of properties with runtime monitoring. One of the more interesting but peripherally related papers compares the effectiveness of different methods of V&V in experiments reported by Brat, et. al. [2].

5. Course Development

Bringing a comprehensive approach to V&V that exploits both synergisms and complementarities into the standard curriculum of computer science and computer engineering is essential to enable effective application of the unification. Development of graduate and undergraduate courses are integral to this research project. The classes will be the laboratories in which the concepts and their implementations are evaluated. A graduate seminar was offered in the Spring of 2005. The first offering of the undergraduate course will be in Fall 2006. The content for the undergraduate course will include:

a. Design for test and verification.
b. Unified Property Specification
c. Introduction to program analysis (static analysis methods).
d. Formal and complete approaches to testing:
   Specification of properties, behaviors and assertion
   Test coverage algorithms based on static analysis processes
   Testing as a continuous process integrating runtime monitoring with conventional testing, model checking and proof-based verification.
a. Applied model checking:
   Model checking as the endpoint of testing
   Property formulation
   Compositional reasoning
b. Classical Dijkstra/Hoare and other proof-based verification.
   This material is already covered in other courses and will not be repeated but the role of this material in a comprehensive approach to verification and validation will be covered.
g. Run-Time Monitoring
   Methods and Tools
   Automated compilation of property monitors.
h. Integration of all the methods in a coherent, complete structure for validation and verification.
i. Extension of verification and validation to security policy issues such as information flow.
j. Failure analysis, fault-tolerance, practical self-stabilization, etc.
k. Verification and validation of non-functional properties such as performance.

6. Future Research

A comprehensive unification of verification and validation methods for software is a generation-long process. The immediate steps are to formalize the logic and the domain specific UPSL for access control in distributed systems of services, to formulate a comprehensive set of properties to be established, to map verification of the properties to methods for verification and validation and to then begin implementing a translation infrastructure which will support verification and validation process. But this example is merely a partial trial run for the unification approach.

The longer term goals for the project include:

a. A uniform approach to specifying properties and behaviors that provides a systematic basis for verification and validation. Currently tests typically are specified as input and output relations (pre-condition and post-condition pairs); properties for model checking are specified as formulas in some temporal logic; properties for formal proofs are specified as equivalence relations, invariants, constraints, or pre-condition and post-condition pairs.
b. A systematic formulation of design principles that enables comprehensive and effective verification and validation, and establishes a process for developing software in which the unified verification approach can be applied.
c. A systematic exploration of the applicability of each method and tool.
d. Development of an implementation that coordinates application of the methods and tools for Java.
e. Evaluation of the unified approach through experimental applications. The experiments will be conducted in the context of the offerings of classes based on the unified methodology and its prototype implementations.
f. Extension of the property specification language and support tools to non-functional properties and performance in particular.

7. Acknowledgements

This research has been supported by NSF grant number 0509354 "Collaborative Research: CSR---AES: Unification of Validation and Verification Methods."

8. References

[1] http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[2] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software," *Formal Methods in Systems Design*, Sept./Nov. 2004.

[3] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee, "Points-to Analysis using BDDs," *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 103-114, 2003.

[4] M. Burrows, M. Abadí, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb 1990.

[5] E. A. Emerson. *Temporal and modal logic*, volume B, pages 955–1072. MIT Press, 1990.

[6] Klaus Havelund and Grigore Ros "An Overview of the Runtime Verification Tool Java PathExplorer" *Formal Methods in System Design*, 24, 189–215, 2004

[7] K. Kane and J. C. Browne. CoorSet: A development environment for associatively coordinated components. In *Coordination Models and Languages, Proceedings of COORDINATION 2004*, pages 216–231, Feb 2004.

[8] K. Kane and J. C. Browne Capability-Based Access Control for Distributed Service-Oriented Systems (Submitted to SACMAT'06)

[9] V. Kuncak, P. Lam, K. Zee, and M. Rinard, "Combining Theorem Proving with Static Analysis for Data Structure Consistency," *Proceedings of SVV'04*, Seattle, WA, Nov. 2004.

[10] J. McHugh, *Towards the Generation of Efficient Code from Verified Programs*, PhD Dissertation, University of Texas at Austin, 1983.

[11] D. J. Richardson and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification," *IEEE Transactions on Software Engineering*, **11**(12):1477-1490, 1985.

[12] J. Shapiro. What *is* a capability, anyway? http://www.eros-os.org/essays/capintro.html.

[13] R. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering*, **12**(1):157-171, 1986.

[14] M. V. Tripunitara and N. Li. Comparing the expressive power of access control models. In Proceedings of the ACM Conference on Computer and Communications Security (CCS), Oct 2004.

[15] J. Zhu, "Symbolic Pointer Analysis," *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 150-157, 2002.