A Visual Tradeoff Space for Formal Verification and Validation Techniques

Doron Drusinsky, James Bret Michael, *Senior Member*, *IEEE*,
and Man-Tak Shing, *Senior Member*, *IEEE*

***Abstract*—Numerous techniques exist for conducting computer-assisted formal verification and validation. The cost associated with these techniques varies, depending on factors such as ease of use, the effort required to construct correct requirement specifications for complex real-life properties, and the effort associated with instrumentation of the software under test. Likewise, existing techniques differ in their ability to effectively cover the system under test and its associated requirements. To aid software engineers in selecting the appropriate technique for the formal verification and validation task at hand, we introduce a three-dimension tradeoff space encompassing both cost and coverage.**

***Index Terms*—Software Verification and Validation, Formal Methods, Model Checking, Assertion Checkers**

I. INTRODUCTION

There are many real-world examples of the impact of software-related failures on our lives, such as the malfunctioning of the Miele G885 SC dishwasher, worldwide recall of the BMW 745i sedan, the temporary closure of Southern California's airspace while air traffic controllers had no access to digital displays of terrain and airspace boundaries, the loss of an Ariane 5 rocket and its payload of satellites, and the loss of life due to friendly fire by the Patriot missile system. Software is ubiquitous, and software errors affect everybody. A study conduct in 2001 and sponsored by the National Institute of Standards and Technology (NIST) found that the annual cost of software errors to the U.S. economy is approximately \$59.5 billion, which in 2001 was about 0.6 percent of the gross domestic product [1].

The ever increasing demand for highly automated high-integrity reactive systems, such as those used in defense (target-tracking system), healthcare (e.g., infusion pump), and transportation (e.g., traction-control system in an automobile), pushes the complexity of embedded systems to new heights. By high-integrity, we mean systems for which the customer or other stakeholder requires a high level of assurance be demonstrated before placing these systems into operation. Reactive systems, under the control of their embedded software, must interact closely with other embedded systems and adhere to tight constraints on both timing and control. Their close interaction with the environment makes the understanding and satisfaction of embedded systems' functional requirements (i.e., "what the software must do") and their safety requirements (i.e., "what the software must not do") a high priority.

The activities for assuring the correctness of reactive systems reside within the Verification and Validation (V&V) process. According to the Guide to the Software Engineering Body of Knowledge [2],

> The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose.

V&V traditionally relies on manual examination of software requirements and design artifacts in addition to the testing of target code. As software-intensive systems have become increasingly complex, traditional V&V techniques are now inadequate for use in locating subtle errors in the software. For example, there are intricate and abstruse system behaviors that are only observable at runtime and at such a fine level of granularity of time that human intervention at runtime is not practical; software automation holds the key to V&V of these types of system behaviors.

Lutz pointed out, in her study of the software errors discovered during the integration and testing phase of the Voyager and Galileo spacecraft, that the majority of the program faults were functional faults, and a large percentage

Manuscript received December 6, 2007. Manuscript revised September 19, 2008. This work was supported in part by the by the National Aeronautics and Space Administration under Grant No. NNG07LD01I. The views and conclusions in this talk are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

Doron Drusinsky is with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943 USA (telephone: 831-656-2397, e-mail: ddrusin@nps.edu).

James Bret Michael is with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943 USA (telephone: 831-656-2655, e-mail: bmichael@nps.edu).

Man-Tak Shing is with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943 USA (telephone: 831-656-2634, e-mail: shing@nps.edu).

of the functional faults were behavioral faults (50% of the safety-related, functional faults in Voyager and 38% of safety-related, functional faults in Galileo) [3]. Lutz's finding highlights the difficulties in understanding and implementing behavioral requirements correctly. We divide the system behaviors into two classes:

1) Logical behavior - This class describes the cause and effect of a computation, typically represented as functional requirements of a system. For example, given two positive numbers *x* and *e*, the output of the square root function *sqrt(x)* must satisfy the requirement: $| x - sqrt(x) * sqrt(x) | < e$.
2) Sequencing behavior – This class describes behaviors as sequences of events, conditions, and constraints on data values and timing. In its vanilla form, sequencing behavior specifies sets of legal and illegal sequences, such as the following automotive body-logic requirement:
   Once engine is turned off, compartment lights must be on until driver door is opened.

Sequencing behavior has two types of common constraints:

1) Timing constraints – describe the timely start and/or termination of successful or forbidden computations, such as the deadline of a periodic computation or the maximum response time of an event handler. For example,
   The sqrt() function must complete its computation and return an answer within 200 milliseconds from the time it is called.
2) Time-series constraints – describe the timely execution of a sequence of computations within a specific duration of time. For example,
   Whenever the system load (L) exceeds 75% of the MaxLoad, L must be reduced back to 50% of the MaxLoad within 1 minute and must remain at or below 60% of the MaxLoad for at least 10 minutes.

Sequencing behaviors with time-series constraints are the most difficult to understand, specify, and implement correctly. We need automated V&V techniques to assure the correctness of both logical and sequencing behaviors. Automated tools for conducting V&V take as input system behaviors and properties specified in a mechanically processable formal specification.

Webster's Dictionary defines formal as "definite, orderly, and methodical." The term "formal methods" refers to the software development activities (e.g., requirements analysis, software design, program transformation, and testing) that employ mathematically based techniques for describing, reasoning about, and realizing system properties, which are expressed using formal languages.

It is widely claimed that formal methods help improve the quality of software [4], [5]. Formal methods have received considerable academic attention during the last three decades, as reflected by the many technical papers published in the open literature. (For example, IEEE Software (Sept. 1990), IEEE Computer (Sept. 1990), and IEEE Transactions on Software Engineering (Sept. 1990, May 1997, Aug. 2000) all have published special issues on formal methods.) In the 1993 seminal study of industrial application of formal methods, Craigen, Gerhart and Ralston [6] reported that

Formal methods are maturing in terms of:

- The range of applications for exploratory, regulatory, and commercial use;
- The solution to technical problems inhibiting larger scale use;
- The understanding of nontechnical barriers to wider spread use; and
- The standardization of concepts and notations.

However, wide use of formal methods in industry and government, even for use in safety-critical commercial and defense applications, has failed to materialize in the past thirteen years [7]. One reason for this lackluster adoption of formal methods is that software development is a multi-facetted process, with each phase of the process having its own unique set of challenges, and there is a lack of a clear and common understanding about the effectiveness of the spectrum of formal methods in different phases of the software development process. In the past, people have been positioning and teaching different classes of formal V&V (FV&V) techniques in isolation, causing confusion in the market – people seeing a myriad of techniques with no uniform way to compare them.

In this article, we present a visual tradeoff space we developed for the NASA IV&V Facility, called the FV&V tradeoff cuboid, for software engineers to discuss the various tradeoffs (e.g. cost, coverage, etc.) between different FV&V approaches in order to select the appropriate techniques for the FV&V of high-integrity software-intensive systems, many of which are reactive systems with complex sequencing behaviors. The rest of the article is organized as follows. We first discuss the different needs for the FV&V techniques in the different phases of the software process in Section II, followed by a description of the three-dimensional FV&V tradeoff space in Section III. We then illustrate the use of the tradeoff space with a qualitative comparison of three classes of FV&V techniques for reactive system behaviors in Section IV and present a sample application of the tradeoff space in Section V. We conclude the paper with a discussion on how the tradeoff space can be used as an aid by software engineers for selecting the appropriate technique for the FV&V task at hand.

II. THE V&V REQUIREMENTS IN THE SOFTWARE LIFE CYCLE

One can view software development as a set of transformations via the following workflows: requirements specification, design, and implementation. Depending on the software process model, these transformations may be carried out in a sequential order (as in the Waterfall [8], or the Spiral processes [9]), or in an iterative and incremental fashion (as in the Unified process [10]). Table 1 shows the input/output of each transformation and the corresponding V&V activities.

TABLE I
THE LIFE-CYCLE V&V ACTIVITIES

Development Activities	**Input**	**Output**	**V&V Activities**	
Requirements Specification	Clients' ideas	System/ software functional and non-functional requirements	Assure the adequacy, correctness, and consistency of requirements; develop acceptance test plan and test cases	Validation
Design	System/ software requirements	Architecture/ component specification	Assure the consistency of design with requirements, and the adequacy of design; develop integration and unit test plan and test cases	Verification
Implementation	Architecture/ component specification	Target Code	Assure the consistency of code with design, and the adequacy of the implementation, execute the tests as planned	

We need to separate the FV&V techniques into two categories: the FV&V for the Requirements phase and the FV&V for the Design/Implementation phase. The FV&V techniques for the Requirements phase are formal validation techniques. These techniques must allow stakeholders to capture and test the formal requirements (e.g., via simulations) to ensure that the developer's cognitive understanding of the requirements matches the formal specifications. The FV&V techniques for the Design/Implementation phase are formal verification techniques. These techniques must aid developers in demonstrating that their software satisfies the requirements (functional and non-functional), and should effectively locate and explain the cause of errors in faulty design and code.

III. THE FV&V DIMENSIONS

Let us return to our discussion of the dimensions of the FV&V tradeoff space, which is made up of the following three dimensions: specification/validation, program/ implementation, and verification.

A. *The specification/validation dimension*

The specification/validation dimension represents the cost, effort and effectiveness associated with formal specification. Formal requirements specification is the process of capturing requirements and properties for the domain of discourse (e.g., component, module, or system being designed or inspected) in a machine interpretable or executable form. Clark et al. reported in [4] that the process of specifying requirements formally enables developers to gain "a deeper understanding of the system being specified," and to "uncover requirements flaws, inconsistencies, ambiguities and incompletenesses." In addition, the artifacts produced by enacting the process "can itself be formally analyzed," thus allowing the possibility for some degree of automation of V&V tasks. The formal specifications describe what any system that solves the real-world problem ought to do. Typically, formal specifications are created from conceptual requirements as understood by the primary modeler. Regardless of what formal notations or formal methods were used, the system modelers always start their requirements-discovery process based on some scenarios involving the system and its environment, express their understanding of the expected behavior or properties of the system informally with natural languages, and then translate the natural language requirements into formal specifications.

The specification/validation dimension deals with the ease of writing formal specifications and getting them right, that is, getting them to represent the cognitive intent the human owner has for this requirement. This dimension measures cost and coverage. Cost is the fiscal cost of creating and validating correct representative formal specifications for desired properties. Coverage is the degree to which a given specification language can actually be used to capture certain properties; a weak formal specification language can only capture simple requirements. For example, the specification language known as Propositional Linear-time Temporal Logic (PLTL) is known to be star-free regular [11] and therefore cannot formally capture requirements that require a stronger formalism, such as requirements that require nontrivial counting. In addition PLTL cannot be used to capture requirements that contain real-time constraints.

B. *The program/implementation dimension*

The program/implementation dimension deals with the ease of the adaptation of a given real-life complex program to a specific FV&V technique. In an ideal world we would use an existing program verbatim for our FV&V technique of choice. In reality however this is often not the case, and a program needs to be modified, truncated, or abstracted to be considered for FV&V. For example, a model checker such as SPIN [12] cannot be used verbatim on a non-trivial C, C++, or Java program; rather, such a program needs to go through a process of abstraction before it can be used for verification, and hence has a low program coverage and a high program cost in the program/implementation dimension.

C. *The verification dimension*

The verification dimension bridges the specification and implementation dimensions. Verification ensures that the software implementation conforms to the specification. The verification dimension represents the cost, effort, and effectiveness of verification. For example, it is generally accepted that manual (i.e., human-based) testing is costly, slow, and error prone. Hence, human-based testing will be represented as a point in the cuboid whose verification dimension highlights high-cost and low-coverage.

IV. QUALITATIVE COMPARISON OF FV&V TECHNIQUES FOR REACTIVE SYSTEM BEHAVIORS

Let us illustrate the use of the tradeoff space with a discussion of cost and coverage tradeoffs among three categories of FV&V techniques: theorem proving, non-execution-based model checking, and execution-based model checking via the combination of runtime verification and automatic test generation. We choose these three categories of techniques because they are used in many hybrid methods. For example, the Software Cost Reduction (SRC) Toolset [13] allows the user to enter the required externally visible behavior of a software system using a tabular notation, and then translates the SRC specification either into Promela [14], a Process Meta Language of the SPIN model checker, for model checking, or into TAME (Timed Automata Modeling Environment) [15], a specialized interface to PVS (Prototype Verification System) [16], for theorem proving. Other tools offer translations of Z [17] specifications to PVS [18], Isabelle/HOL [19], EVES [20], or SAL [21] for theorem proving or modeling checking.

The coverage cuboid, shown in Figure 1, represents the coverage-space tradeoff between three FV&V techniques. Each point in the solid represents the extent of coverage in the three dimensions (specification, verification and implementation) provided by a given FV&V technique. Hence, an FV&V technique with high coverage (e.g., high specification coverage) is better in that aspect than a technique with low coverage.

Figure 2 is the cost cuboid; it represents the cost-space tradeoff between the three FV&V techniques. Each point in the solid represents the cost in each dimension induced by a given FV&V technique. Clearly, an FV&V technique with high cost along some axis (e.g., high verification cost) is worse in that aspect than a technique with a low cost.

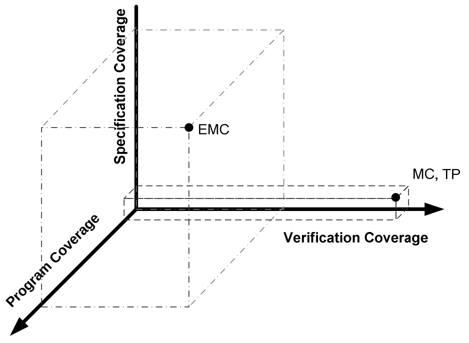

Fig. 1. The coverage space

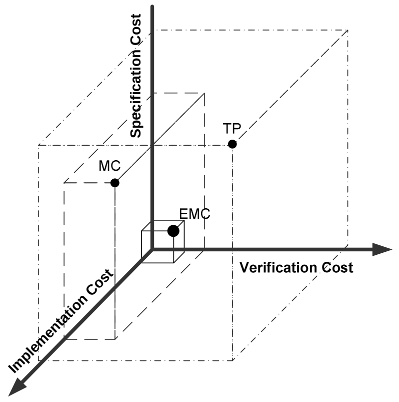

Fig. 2. The cost space

A. Theorem Proving

As its name suggests, Theorem Proving (TP) is a formal verification technique that uses mathematical proof techniques to make a convincing argument that a program conforms to a formal requirement. FV&V TP tools require a human driver because the underlying problem to be solved is typically undecidable. In addition, the choice of the specification language affects the skill level required by the driver. For example, ACL2 [22] uses Propositional-Logic (PL) specification that uses Lisp programming style notation for specification, whereas STeP (the Stanford Temporal Prover) [23] uses Propositional Linear-time Temporal Logic (PLTL) for specification [24], a language that requires more user expertise than PL. HOL theorem provers [25] are a family of interactive theorem proving systems that use higher order logic, which is theoretically more descriptive than PLTL but is arguably harder to use when it comes to specification of reactive system requirements. Examples of HOL TPs include the NQTHM [26], HOL4 [25], Isabelle [27], ProofPower [28] and PVS [16], as well as several efforts to embed temporal logic in HOL [29]-[32]. There are also a number of formal methods (e.g. methods using Floyd-Hoare Logic [33]-[35] and methods using the Type systems [36]) for the verification of target code via TP during the code-development phase. In the Floyd-Hoare Logic methods, every programming step has a pre-condition, post-condition and an invariant. The verifier is expected to use a proof system to check that the post condition follows from the precondition while the invariant is valid. In the Type systems methods, V&V can be moved to the design stage by formally stating the requirements in constructive logic. The software engineer then acts as a mathematician and proves that the requirement is a theorem that follows from the domain axioms. The system then extracts the code automatically from this proof. Therefore the generated code now automatically becomes correct, as the software engineer indirectly proved it to be so.

1) The specification/validation dimension of TP

This dimension is affected by the expressive power and ease of use of the formal specification languages used by TP tools. In general, the more automated the theorem prover, the more restrictive is its specification language. Existing theorem provers have rather weak specification languages - mostly based on some form of temporal logic. Such languages are also considered hard to use because they are considerably different than the languages used by programmers for whom a common practice is to model and program using UML-based visual languages. It is difficult for system designers who have a limited knowledge of formal logic to visualize the subtle meaning of temporal logic statements in order to validate the correctness of the formal specifications. Consequently, we ranked TP techniques as having low specification coverage and high specification cost.

2) The program/implementation dimension of TP

TP techniques rely on special programming languages tailored specifically for the TP process. Consequently, it is not possible to perform TP on an existing Java or C++ application verbatim, i.e., an existing complex application needs to be first translated into a new representation using the TP tool's language of choice. In most safety-critical application, such as NASA flight-code, or complex defense applications (e.g., the AEGIS weapon system), the new representation will not cover all aspects of the original program; for example, STeP does not have nearly the same library support as Java or C++. Consequently, we ranked TP techniques as having low program coverage and high program cost.

3) The verification dimension of TP

As discussed above, TP is never automatic, and requires a high level of expertise on the part of the user in automated reasoning. Even with such expertise, it is not guaranteed that the human driver will be able to navigate the TP process (e.g., selecting the inference rules or managing the set of support axioms) to completion. Nevertheless, when the process does complete it provides 100% coverage, that is, no more testing is required for that specific requirement. Hence, we ranked TP techniques as having good verification coverage but high verification cost.

B. Model Checking

Classical, or non-execution-based, Model Checking (MC) is an algorithmic formal verification technique. MC is a push-button verification technique in that once a program is set-up for MC and a property (e.g., reachability, safety, liveness, and fairness) is formally captured using the formal specification language of choice, the process does not require an expert human driver.

1) The specification/validation dimension of MC

Contemporary MC techniques are limited in the specification dimension. For example, SPIN [12] uses PLTL or Büchi-automata [37] for requirement specification, resulting is the similar specification coverage and cost limitations as TP techniques. Kronos [38] and Uppall [39], on the other hand, use timed automata to verify real-time properties specified in computation tree logic (CTL) [40]. Both CTL and PLTL are rather weak subsets of full branching time logic (CTL*) [41]. Both CTL and CTL* use path operators, making it challenging to formulate correct specifications. Like the formal specifications in the TP techniques, specifications for the MC techniques are text-based and difficult to visualize and validate by system designers. Unlike TP, MC does not require the detailed assertions (e.g. invariants) to help guide the intermediate steps of the proof processes. Hence, we rank MC as having low specification coverage, yet with a specification cost somewhat lower than that assigned to the TP category.

2) The program/implementation dimension of MC

Model checking's Achilles heel is typically considered to be the state-space explosion problem, where the size of the problem space as seen by the MC grows exponentially as the program under verification grows. Consequently, MC is limited to finite-state components and is performance-constrained by the number of states in that component. For example, a single 32-bit integer variable induces effectively 2^{32} states. For FV&V of large real-life systems there are two options available to MC users: (i) to ignore large parts of the system using a process known as abstraction [42], where MC is performed on a small abstract model of the original system, or (ii) to carve out limited, small, parts of the system and perform MC only on those parts. In either case there is a non-trivial effort involved. In addition, the artifact that is eventually model-checked differs significantly from the original system, being either an abstract version or limited portion of the original system. We therefore rank MC as having low program coverage and high program cost.

3) The verification dimension of MC

The premise of MC is automatic, "push-button", verification with no special driver required. Also, MC results in 100% verification coverage of the component being verified (for components small enough to allow MC to complete without running into the state explosion problem). Hence, we rank MC as having high verification coverage and low verification cost.

C. Execution-based Model Checking

Runtime Verification (RV) involves monitoring the runtime execution of a system and checking the observed runtime behavior against the system's formal specification. Hence, RV behaves as an automated observer of the program's behavior and compares that behavior with the expected behavior per the formal specification.

Some published RV tools are the TemporalRover/DBRover [43], PaX[44] and RT-Mac [45], all of which use extensions and variants of PLTL as the specification language of choice, and the StateRover [46] that uses deterministic and non-deterministic statechart diagrams as its specification language.

Execution-based Model Checking (EMC) is a combination of RV and Automatic Test Generation (ATG). With EMC, a large volume of automatically generated tests are used to exercise the program or system under test (SUT), using RV on the other end to check the SUT's conformance to the formal specification.

Some ATG tools that, when combined with RV tools, create an EMC technique are the StateRover's white-box automatic test-generator [47] and NASA's Java Path Finder

(JPF) [48].

1) The specification/validation dimension of EMC

Although some early RV tools have used limited specification languages such as PLTL [24] and MTL [49], there is nothing inherent in the ATG, RV, and EMC techniques that limit the specification language. Indeed, the StateRover's specification language is Turing equivalent. In contrast, no specification language for MC or TP is Turing equivalent. In addition, the current state-of-practice considers UML diagrams as an easy-to-use modeling and specification language, rendering UML-based formal specification less costly to create and more powerful than specification languages used by MC and TP techniques. The availability of executable code for the formal assertions allows system designers to test specifications (via scenario simulation) independent of the prototype design, ensuring that the system designers truly understand the required system behavior without being tainted by any pre-conceived solutions [50]. Hence, we rank EMC as having high specification coverage and low specification cost.

2) The program/application dimension of EMC

The premise of RV is that it can be used for FV&V of any existing, almost unmodified Java, C, or C++ system, regardless of its size and complexity. This is true up to the insertion of instrumentation code. We therefore rank EMC as having high program coverage and low program cost.

3) The verification dimension of EMC

EMC is an execution-based FV&V method, where the system under test and the specification execute in tandem. Consequently, there is always a possibility that the ATG did not generate a test sequence that violates a requirement. Hence EMC's verification coverage cannot be 100% and we therefore rank EMC as having lower verification coverage than MC or TP. Depending on the level of automation of the test-generator, EMC is fully or partially automatic. EMC has a low verification cost when using an automatic ATG tool.

V. Sample Application of the tradeoff space

In 1993, the National Aeronautics and Space Administration (NASA) established an IV&V facility in the wake of the Space Shuttle Challenger accident as part of a plan "to provide the highest achievable levels of safety and cost-effectiveness for mission critical software" [51]. The facility has continuously developed their IV&V program, incorporating new technologies and better verification and validation techniques in an effort to improve the V&V process. Earlier versions of the V&V process relied on manual examination and independent testing of target code. These techniques are ineffective for use in validation because there is no provision in the process to validate the developer's correct understanding of the requirements as manifested by the system's features, capabilities, properties and functions. Moreover, the processes were unable to locate the subtle errors in increasingly complex software-intensive systems. Hence, the IV&V Facility is transitioning from using manual V&V processes to utilizing highly automated processes involving the application of advanced computer-aided V&V techniques. In 2007, the facility adopted a System Reference Model (SRM) framework that allows the IV&V teams to capture their own understanding of the problem and the expected behavior of *any* proposed system for solving the problem via executable formal assertions of mission- and safety-critical behaviors [52]. In particular, the facility was looking for formal techniques that can capture and validate sequencing behaviors with timing and time-series constraints.

After comparing the capabilities and costs among the three major FV&V approaches (TP, MC and EMC), the NASA IV&V Facility selected the EMC approach and chose to specify the mission- and safety-critical behaviors in terms of Statechart assertions for the following reasons:

1) As shown in the Specification dimension of the Coverage Space (Figure 1), it is very difficult, if not impossible, to specify sequencing behaviors with timing and time-series constraints using PLTL.
2) The IV&V teams found that Statechart assertions are easier to create and understand than the text-based PLTL assertions, as shown in the Specification dimension of the Cost Space (Figure 2). In addition, the IV&V teams can execute the formal assertions to validate their understanding of the expected behavior of *any* eventual system implementation without being tainted by the developer's agenda.
3) Since the IV&V Facility is often limited in its ability to perform detailed analyses of developer's code, they will need to rely on black-box testing to verify the correctness of the developer's systems. As shown in the Implementation dimension of the Coverage Space (Figure 1) and the Implementation and the Verification dimensions of the Cost Space (Figure 2), EMC provides superior coverage while being the least costly amongst the three FV&V techniques.
4) The limited test coverage of the EMC approach, as shown in the Verification dimension of the Coverage Space (Figure 1), does increase the risk of not being able to uncover errors in the delivered system. However, the impact of such risk to the IV&V effort is less significant than it is to the developer's V&V effort, since the IV&V teams only act as a second line of defense in the overall NASA safety and mission assurance program.

VI. Conclusions

Clearly, as visually depicted by Figures 1 and 2, we have identified a tradeoff space associated with FV&V. It follows from an analysis of this space that an organization may need to determine how to best allocate its limited V&V resources. For example, an organization that chooses TP or MC is effectively deciding in favor of good verification yet for a restricted set of behavioral (reactive) requirements, since many behavioral requirements of interest cannot be addressed by TP or MC. In addition, a choice of MC will limit the size or level of abstraction of the application being verified. EMC on the other hand, when compared with MC and TP, has superior specification coverage and cost as well as superior program coverage and cost, but inferior verification coverage.

Consequently, one can conclude from Figures 1 and 2 that the option boils down to a choice between:

1) Thoroughly verifying a limited application against a limited set of requirements with a high upfront cost of specification-development and program-adaptation;
2) Partially verifying an entire application as-is, against a wide set of real-life requirements.

This choice might also help explain the lackluster acceptance of FV&V techniques by the industry. In the past, MC and TP have been the prominent available FV&V techniques, forcing the marketplace to fund verification of limited or abstracted components against limited, often seen as over simplified, requirements: This is not in our opinion a good investment.

Studies of software failures typically point to the importance of correct requirements and the difficulties in getting the correct description of these requirements. One must start with the correct requirements specifications. Otherwise, it does not matter how effective and efficient a verification technique is, it becomes an exercise in futility to formally verify that a system conforms to invalid requirements (i.e., that we built the wrong system correctly). Hence, it is important to select the FV&V techniques that are both cost-effective and coverage-effective in the specification/validation dimension.

REFERENCES

[1] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standard and Technology, Planning Report 02-3, May 2002. Available: http://www.nist.gov/director/prog-ofc/report02-3.pdf.
[2] P. Bourque and R. Dupuis, eds., Swebok: Guide to the Software Engineering Body of Knowledge (2004 Version), IEEE, 2004.
[3] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," in Proc. IEEE Int'l Symp. Requirements Engineering, IEEE, 1993, pp. 26-133.
[4] E. Clarke, J. Wing, et. al., "Formal Methods: State of the Art and Future Direction," ACM Computing Surveys, vol. 28, no. 4, pp. 626-643, Dec. 1996.
[5] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling," IEEE Trans. Software Eng., vol. 24, no. 1, pp. 4-14, Jan. 1998.
[6] D. Craigen, S. Gerhart and T. Ralston, "An International Survey of Industrial Applications of Formal Methods, vol. 1 and 2," Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, Tech. Report NIST GCR 93-626, March 1993.
[7] J. Bowen and M. Hinchey, "Ten Commandments of Formal Methods – Ten Years Later," IEEE Computer, pp. 40-48, Jan. 2006.
[8] W.W. Royce, "Managing the Development for Large Software Systems," in Proc. WESCON, 1970, pp. 1-9.
[9] B. Boehm, "A Spiral Model of Software Development and Enhancement," Computer, pp. 61-72, May 1988.
[10] I. Jacobson, G. Booch and J. Rumbaugh, The Unified Development Process, Addison-Wesley, Reading, MA, 1999.
[11] J. Cohen, D. Perrin and J.-E. Pin, "On the Expressive Power of Temporal Logic," J. Computer and System Sciences, vol. 46, no. 3, pp. 271-294, 1993.
[12] G. Holzmann, "The Model Checker SPIN," IEEE Trans. Software Engineering, vol. 23, no. 5, pp. 279-295, 1997.
[13] C. Heitmeyer, "Formal Methods for Specifying, Validating, and Verifying Requirements," J. Universal Computer Science, vol. 13, no. 5, pp. 607-618, 2007.
[14] G.J. Holzmann, Design and Validation of Computer Protocols, Englewood Cliffs, N.J.: Prentice Hall, 1991.
[15] M. Archer, "TAME: Using PVS strategies for special-purpose theorem proving," Annals of Mathematics and Artificial Intelligence, vol. 29, pp. 131–189, Feb. 2001.
[16] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert, "PVS Prover Guide, Version 2.4," Computer Science Lab, SRI International, Menlo Park, CA, Technical Report, November 2001.
[17] J. Spivey, The Z Notation: A Reference Manual (2nd Ed.), Prentice Hall, 1992. Available: http://spivey.oriel.ox.ac.uk/~mike/zrm/.
[18] D. Stringer-Calvert, S. Stepney, and I. Wand, "Using PVS to prove a Z refinement: A case study," in Formal Methods Europe (FME '97), 1997, J. Fitzgerald, C. Jones, and P. Lucas (eds.), LNCS, vol. 1313, Springer-Verlag, pp. 573-588.
[19] Kolyang, T. Santen and B. Wolf, "A structure preserving encoding of Z in Isabelle/HOL," in Theorem Proving in Higher Order Logics (TPHOLs '96), 1996, J. von Wright, J. Grundy, and J. Harrison (eds.), LNCS, vol. 1125, Springer-Verlag, pp. 283-298.
[20] M. Saaltink, "The Z-Eves system," in J. Bowen, M. Hinchey, and D. Till (ed.), Proc. Int'l Conf. of Z Users (ZUM '97), 1997, LNCS, vol. 1212, Springer-Verlag, pp. 72-85.
[21] G. Smith and L. Wildman, "Model checking Z specifications using SAL," in Proc. Int'l Conf. of B and Z Users (ZB 2005), 2005, H. Treharne, S. King, M. Henson and S. Schneider (eds.), LNCS, vol. 3455, Springer-Verlag, pp. 85-103.
[22] M. Kaufmann, P. Manolios, and J. S. Moore, Computer-Aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.
[23] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe, "STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems," in Proc. 8th Int'l Conf. Computer Aided Verification, 1996, LNCS 1102, Springer-Verlag, pp. 415-418.
[24] U. Nitsche, "Propositional Linear Temporal Logic and Language Homomorphisms," in Proc. 3rd Int'l Symp. Logical Foundations Computer Science, 1994, LNCS 813, Springer-Verlag, pp. 265-277.
[25] M. J. C. Gordon and T. F. Melham, eds., Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993.
[26] R.S. Boyer and J.S. Moore, A Computational Logic Handbook, Academic Press, 1988.
[27] L. C. Paulson, Isabelle: A Generic Theorem Prover, LNCS 828, Springer, 1994.
[28] D. King and R. Arthan, "Development of Practical Verification Tools," ICL Systems J., vol. 11, no. 1, pp. 106-122, 1996.
[29] R.W. S. Hale, "Programming in Temporal Logic," Ph.D. thesis, Computer Laboratory, Cambridge University, Cambridge, U.K., published as Technical Report 173, Oct. 1989.
[30] J. J. Joyce, Multi Level Verification of Microprocessor-Based Systems, Ph.D. thesis, Computer Laboratory, Cambridge University, Cambridge, U.K., published as technical report 195May 1990.
[31] J. von Wright, "Mechanizing the Temporal Logic of Actions in HOL," in Proc. Int'l Workshop HOL Theorem Proving System and Its Applications, 1991, IEEE, pp. 155-159.
[32] R. Cardell-Oliver, R. Hale and J. Herbert, "An Embedding of Timed Transition Systems in HOL," in Proc. Int'l Workshop Higher Order Logic Theorem Proving and its Applications, 1992, L. J. M. Claesen and M. J. C. Gordon, eds., North-Holland, pp. 263-278.
[33] D. Gries, The Science of Programming, Springer-Verlag, New York, 1981.
[34] K.R. Apt and E.-R. Olderrog, Verification and Validation of Sequential and Concurrent Programs (2nd Ed.), Springer-Verlag New York, 1997.
[35] J. Barnes, "The SPARK Way to Correctness is Via Abstraction," ACM SIGAda Ada Letters, vol. XX, no. 4, pp. 69-79, 2000.
[36] R.L. Constable, Implementing Mathematics with the Nuprl Proof Development System, Prentice Hall, New Jersey, 1986..
[37] W. Thomas, "Automata on infinite objects," in Handbook of Theoretical Computer Science (vol. B): Formal Models and Semantics, J. van Leeuwen (ed.), MIT Press, Cambridge, MA, pp. 133–191, 1990.
[38] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A Model-Checking Tool for Real-Time Systems," in Proc. 10th Int'l Conf. Computer-Aided Verification, 1998, A.J. Hu and M.Y. Vardi, eds., LNCS 1427, Springer-Verlag, pp. 546-550.
[39] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," Int'l J. Software Tools for Technology Transfer, vol. 1, nos. 1-2, pp. 134-152, 1997.

[40] E. M. Clarke and E. A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic,” in Proc. Workshop on Logic of Programs, 1981, D. Kozen, ed., LNCS 131, Springer-Verlag, pp. 52-71.

[41] E. A. Emerson and J. Y. Halpern, “‘Sometimes’ and ‘Not Never’ Revisited: On Branching versus Linear Time Temporal Logic,” J. ACM, vol. 33, no. 1, pp. 151-178, 1986.

[42] E. Clarke, O. Grumberg, and D. Long, “Model Checking and Abstraction,” ACM Trans. Programming Languages and Systems, vol. 16, no. 5, pp. 1512-1542, 1994.

[43] D. Drusinsky, “The Temporal Rover and the ATG Rover,” in Proc. SPIN 2000 Workshop, 2000, LNCS 1885, Springer-Verlag, pp. 323-329.

[44] K. Havelund and G. Rosu, “An Overview of the Runtime Verification Tool Java PathExplorer,” Formal Methods in System Design, vol. 24, Springer Netherlands, pp. 189-215, 2004.

[45] U. Sammapun, I. Lee, and O. Sokolsky, “RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties,” in Proc. 11th IEEE Int’l Conf. Embedded and Real-Time Computing Systems and Applications, 2005, IEEE, pp. 147-153.

[46] D. Drusinsky, “Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions,” in Proc. 4th Workshop on Runtime Verification, Electronic Notes in Theoretical Computer Science [online], vol. 113, Springer, pp. 3-21, 2005.

[47] D. Drusinsky, Modeling and Verification Using UML Statecharts - A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking, Elsevier, 2006.

[48] K. Havelund and T. Pressburger, “Model Checking Java Programs using Java PathFinder,” Int’l J. Software Tools for Technology Transfer, vol. 2, no. 4, pp. 366-381, 2000.

[49] E. Chang, A. Pnueli and Z. Manna, “Compositional Verification of Real-Time Systems,” in Proc. 9th IEEE Symp. Logic in Computer Science, 1994, IEEE, pp. 458-465.

[50] D. Drusinsky, M. Shing, and K. Demir, “Creating and Validating Embedded Assertion Statecharts,” IEEE Distributed Systems Online [online], vol. 8, no. 5, 2007.

[51] About IV&V, NASA IV&V Facility, Accessed in Jun 2008, http://www.nasa.gov/centers/ivv/about/index.html

[52] D. Drusinsky, J.B. Michael, and M. Shing , “A framework for computer-aided validation,” Innovations in Systems and Software Engineering, vol. 4, no. 2, pp. 161-168, June 2008.

Doron Drusinsky is an associate professor of computer science at the Naval Postgraduate School, and president of Time-Rover. His research interests are formal methods, requirement elicitation and validation, and sound construction of safety-critical systems. He is one of the world’s leading experts on UML statecharts. He worked for Sony from 1988 until 1993 when he founded R-Active Concepts and authored BetterState, a UML statecharts design tool. BetterState was acquired by ISIWindRiver Systems in 1997. Doron established Time Rover, Inc. and authored the Temporal Rover, DBRover and StateRover formal verification tools. He has written many technical papers and a book on the subject. He received his PhD in computer science from the Weizmann Institute of Science.

James Bret Michael (S’87, M’92, SM’97) is a professor of computer science and electrical and computer engineering at the Naval Postgraduate School. His research interests include reliability, safety, and security engineering, in the context of distributed systems. He serves as an Associate Editor-in-Chief for IEEE Security & Privacy magazine, an Associate Editor for the IEEE Systems Journal, and member of the Advisory Board of IEEE Software magazine. He is the chair of the IEEE Technical Committee on Safety of Systems and a member of the IEEE Reliability Society’s Administrative Committee. He received his PhD in Information Technology from the George Mason University in 1993.

Man-Tak Shing (M’03, SM’07) is an associate professor of computer science at the Naval Postgraduate School. His research interests include software engineering, modeling and design of real-time and distributed systems, and the specification, validation, and runtime monitoring of temporal assertions. He is on the program committees of several conferences dedicated to software engineering and is a member of the Steering Committee of the IEEE International Rapid System Symposium. He was the program co-chair for the IEEE Rapid System Prototyping Workshop in 2004 prior to being the general co-chair for the symposium in 2008. He received his PhD in computer science from the University of California, San Diego.