EXECUTABLE FORMAL MODELS FOR VALIDATION AND SPECLESS VERIFICATION

David Greve, Rockwell Collins, Cedar Rapids, IA

Matthew Wilding, Rockwell Collins, Cedar Rapids, IA

Abstract

Verification and certification of flight critical software and application-specific integrated circuits (ASICs) is currently a labor-intensive, manual process involving extensive testing, inspections, and process documentation. The complexity of these systems and devices will increase both because increases in cockpit automation and application integration offer important safety benefits and because astonishing improvements in digital computing technology can potentially improve performance and decrease cost. The current approach to verification and certification will be challenged by this increased complexity. In order to reap fully the benefits of these technological advances we must develop new methods for verification and certification of flight critical devices that provide higher degrees of assurance for increasingly complex systems while simultaneously streamlining the verification process.

The development of executable formal models may offer higher degrees of assurance, address increased complexity, and streamline certain aspects of the verification process. Increased assurance can be obtained as a result of rigorous, mechanical, mathematically complete checks of consistency and completeness of system requirements as well as proofs of correctness of specific implementations. As vector-based testing becomes increasingly inadequate to assure correctness in the face of exponentially growing state space, formal proofs of correctness can encompass the entire design, demonstrating correctness once and for all. In addition, executing real world stimulus on a formal model helps allay concerns about inconsistencies between the model used to support reasoning and the actual implementation. Formal models representing specific implementations can also be used to support specless verification activities such as product family verification, symbolic simulation, and self-checking tool arrangements.

1. The Verification Challenge

Technological advances in the commercial realm are driving electronics prices lower while providing improvements in performance. Mass production and economies of scale have allowed incredible price/performance ratios and the accelerated pace of technology development and deployment has led to ever greater time-to-market pressures on technology producers. These trends in the everyday consumer electronics arena create the expectation that, even in a safety critical market such as avionics, cost and time to market should decline and that performance and functionality should improve. As a result of these and other forces, avionics systems are becoming increasingly complex.

These same trends, however, could have a negative impact on overall system safety. As complexity increases, the effort required to provide the same level of operational assurance increases as well. If left unchecked, simple state space arguments predict that a linear increase in complexity can result in an exponential increase in the effort required to provide a particular level of assurance. Even in the face of this additional complexity demand for improved safety continues to increase. The objective of NASA's current Aviation Safety Program, for example, is not simply to maintain current levels of safety, but to improve safety [6]. Obtaining such improvement in the face of increasing complexity and pressures to reduce overall costs will require advances on many fronts.

While classical verification has served us well in the past, it appears that new verification approaches will be needed to control verification costs and to provide improved safety in the face of increasing complexity. The challenge, therefore, is to find new verification techniques whose capabilities will scale to match the complexity of newer systems.

1.1 Formal Methods

One possible technique for addressing the verification challenge is the use of formal methods. The term “**formal methods**” describes a discipline in which mathematical specification and reasoning are applied to the development of digital systems. The adjective “formal” is often used in the sense of process, meaning that the process adheres to conventional or accepted methods or standards of operation. In contrast, the “formal” in “formal methods”, is used in the sense of mathematics, meaning that these methods employ logic, rules of inference, and languages with rigorously defined semantics. The use of formal methods does not exclude the use of a formal process. On the contrary, rigorous mathematical analysis of systems will prove most beneficial when performed as part of a well-defined verification process. The guidelines set forth in both DO-178B and DO-254, for example, describe how formal methods can be used in the context of a formal process to support certification of avionics systems.

The application of formal methods requires the development of formal models. A **formal model** is a model constructed in a language with a formal semantics, which is to say that the meaning of each language construct is precisely, and completely defined. Certain formal languages can be used in conjunction with a proof engine. A **proof engine** is an automated tool for applying logical rules of inference to the constructs of a language. Such engines enable mechanical, mathematical proofs of model properties. Theorem provers, model checkers, and equivalence checkers are all examples of different types of proof engines. The capstone of formal methods is formal verification. **Formal verification** is a procedure by which an implementation is mathematically proven to implement its specification. Note that formal verification, by definition, requires a formal model of both the implementation and the specification.

The primary purpose of formal verification is to demonstrate that, under all conditions of interest, a particular design performs as specified. The result of such a proof is confidence that, under any condition of interest, no matter what input stimuli is applied, a design will operate as expected. Note that this assurance can be provided without running test vectors against the design. It doesn’t matter how large the state space of the device is, because a formal proof of correctness can account for every one of the very large number of states the design might experience. Formal methods, therefore, have the potential to provide the crucial scaling property we are looking for in verifying increasingly complex designs.

Formal verification, however, is not a panacea for the verification challenge. One issue is model validation: what assurance is there that the final statement of correctness about a formal model applies directly to the physical system being reasoned about? While one might claim that formal verification establishes the correctness of a particular design, such a proof is of little value unless the model used during formal verification accurately represents the final device. Another fundamental issue is determining how to apply formal methods effectively to a complex design. While extensive informal documentation of system requirements is commonplace, it is unusual in practice to find precise formal system specifications against which a particular implementation can be proven correct. This paper discusses executable formal models and how they assist in the model validation process as well as techniques for applying formal methods to establish certain useful system properties without the need for extensive formal modeling or complete formal verification.

2. Model Validation

A model is a simplified replica or representation of a physical or logical object. Newton's laws of gravity are an example of a model of gravitational attraction. Because it is simplified, a model does not exhibit all of the possible behaviors of the modeled object. In order for a model to be useful it must, of course, reflect the characteristics of the object that influence the desirable properties of the object. More to the point, the model must exhibit influential characteristics of an object with sufficient fidelity to predict pertinent behavior of the actual object accurately. Newton’s laws of gravity predict with sufficient accuracy the behavior of pendulums and the fall of objects in a vacuum. They are, however, not capable of predicting the gravitational lens effect or the idiosyncrasy of the orbit of Mercury around the sun. We see from this example that if we want to be able to depend on Newton's laws we must make sure that they work well enough for our purposes under the conditions of interest.

Model validation involves comparing the desired or measured behavior of an object with the behavior predicted by a model of that object. Model validation is inherently informal (not mathematically rigorous), so when we say a model has been validated, we mean that we believe that it exhibits the expected behavior under some sufficient set of conditions. Newton's laws of gravity, for example, were validated through experiment and observation for slow moving, massive objects in relatively weak gravitational fields.

There are several ways in which a formal model can be validated. Rigorous walk-throughs are an example of one validation technique. Our focus, however, is on another validation technique: simulation. Simulation of a model involves computing the response of the model to a set of input stimuli. Simulation helps guarantee that a specification meets its expectations, it allows for the testing of a particular implementation against functional and regression test suites, and it enables designers to explore system level integration issues early in the design process. Simulation of formal models, however, is practical only if the formal models are executable. Executable formal models are the crucial link between formal proofs of correctness and the physical devices being constructed. **Executable formal models** are formal models that can be evaluated efficiently on concrete input data to produce observable output [4]. Such models must execute efficiently because they must be capable of running large tests in an acceptable time period. They must produce observable output so that the generated results can be compared with the expected results. Model validation using executable formal models can take many forms including rectifying specifications and expectations, contributing to system level co-design, assuring model maintenance, establishing a service history, and regression testing.

2.1 Specification versus Expectation

A **formal specification** is a rigorous description of how a system operates. An expectation is an informal notion of how the user and designer intend the system to operate. It is important to note that agreement on a specification does not imply agreement on design intention. Specifications are formalized expectations. Expectations that are not included in the specification are, by definition, not formalized. The difference between the formalized specification and design intentions is called the **expectation gap**. It is often the case that what people refer to as bugs in a system are actually conditions that fall in the expectation gap: conditions for which the user or designer has some expectation but for which there exist no formal specification. For example, it is a bug if a subroutine designed to increment a number returns the value of 3 when passed the value 1. What if, instead, the subroutine returned the expected value of 2 but required hours to do so? Is that a bug? Perhaps the specification merely states that the subroutine is supposed to increment a value, which it does. Implicitly, however, one expects this operation to be performed in reasonable time. This is an example of a case that lies in the expectation gap: an implementation that meets its specification but fails to meet an expectation.

Formal methods can be applied to formal specifications to prove that they are self-consistent and complete. **Self-consistent** means that there is no set of conditions under which the device is required to perform contradictory functions. A specification is **complete** if, for the list of identified conditions, the specification covers every possible combination of conditions. It is important to note that formally complete does not mean that the list of possible conditions has been exhausted. It can be the case that there are conditions that lie in the expectation gap that are not recorded in the specification. As a result, despite all of their potential, formal methods cannot prove the absence of an expectation gap. This is not to say that expectations cannot be formalized as requirements. What it does mean, however, is that there is no rigorous way to demonstrate that every expectation is covered by a specification.

What then can be done? A rigorous review cycle can help substantially reduce and even eliminate the size of the expectation gap. Such reviews, however, are sometimes most appropriate for the designer who is intimately familiar with the impact of various design decisions. It is sometimes more difficult to communicate such concerns with the users of the system, especially if the users are not familiar with the specification language.

Executable formal models can be used to explore the ramifications of different design decisions. Because the models are formal, they can be shown to be self-consistent and complete. In addition, because the models are executable, they can be used to animate the specification in a way that makes sense to both the designer and the user of the system. As a result, differences between the system implementation and the expectation of the designers and users can be resolved early in the design process, thus helping to diminish the expectation gap. In a paper discussing methods for reducing the potential for mode confusion in avionics systems it was reported that, "Our experiences to date have shown that, if anything, we underestimated the power of [animating the specification]. In every demonstration, the visualization has generated vigorous, positive debate between [the developers and the users]."[2]

2.2 Co-Design

Abstraction is the process of refining a complex system into a simpler system that retains the properties of interest. An abstract description of a system is one that concentrates the essential characteristics of that system while eliminating irrelevant details. In order for the top-level specifications of complex systems to be tractable, they must be stated abstractly in terms of their composite subsystems. It is impossible in complex systems to enumerate all of the subsystem states and all of the internal conditions that may arise. A good specification, therefore, encapsulates the complexity of the subsystems and then uses properties of these abstract subsystems to express the expected behavior of the overall system. Unfortunately, the process of abstracting these subsystems gives rise to the potential for incompleteness (in the sense of expectations) in the top-level specification. As a result, careless abstraction can result in expectation gaps in the specification of the final, integrated system.

One way of addressing the problem of top-level system integration is co-design. **Co-design** is a technique that involves writing descriptions of each of the components of a complex design and combining them in a unified environment that allows testing and analysis of the system as a whole. This methodology allows analysis of a wide variety of test cases and helps to validate the behavior of the top-level system against system expectations. This kind of early testing and analysis allows the designer to identify system-level problems quickly and to deal with them early in the design cycle.

Executable formal models can be powerful components in the co-design process. The fact that the models are expressed formally allows proofs of correctness to be performed to justify each step in the abstraction process. This provides confidence that the abstraction does not result in (formal) incompleteness in the top-level specification. Because such models are executable it is also possible to integrate them at the system level and perform testing to help identify unexpected interactions between modules early in the design cycle.

2.3 Maintenance

A model can reflect the behavior of a system accurately only if the model evolves with changes in the system. One challenge experienced in the modeling of complex systems is the maintenance of the models themselves. As design specifications and implementations evolve over time, the models of those systems must be updated to reflect the changes. If a model is not an integral part of the design process, it is easy for it to fall into disrepair and eventually become useless in predicting the behavior of the system. It is crucial, therefore, to inject formal modeling into the design process. It is possible to dictate such actions through a rigorous design process, but it is more natural to make such modeling part of the process if the model contributes directly to the design. This situation is most easily attained if the formal model can act in the role of a simulator. It is exactly this situation that is enabled by executable formal models.

2.4 Service History

The perceived validity of a model is related to the service history of that model. As a result, service history is an important factor in establishing confidence in the correctness of a particular design. It is quite likely that a brand new piece of software will contain bugs. However, a piece of software that has been actively and successfully used in the field for many years is less likely to have bugs. When one is in the position of establishing confidence in a model, one should consider its service history. Being able to compare the performance of a model against millions of different test cases provides a substantial degree of confidence in the model. In addition, a model that is used as a simulator as an integral part of the design process will have fewer bugs than one developed as an aside in a vacuum. By enabling the use of formal models as simulators, executable formal models provide a means of achieving a substantial service history for formal models of a system.

2.5 Testing

Testing is another means for establishing the validity of a model. Current verification techniques rely extensively on testing. While simple state space arguments make clear that exhaustive testing is impossible, even for moderately complex designs, testing has been sufficient to provide, in part, the impressive level of safety we currently enjoy. Formal verification, on the other hand, promises to provide a comprehensive correctness guarantee by virtue of a single proof. Such claims, however, are only as valid as the model upon which they are based. How does one use formal verification to extend our current capabilities and provide greater degrees of assurance in the verification process?

Executable formal models may provide part of the answer since they allow us to leverage both testing and formal verification. By executing regression tests on the formal model itself, one has the assurance that the model actually represents the system under development. This work can then be extended though formal verification to support the proof that the system model is correct under all circumstances.

While the above technique focuses on the implementation, it is also possible to use executable formal models to test the system specification. While this may seem redundant, given that tests are frequently derived from the system specification, it makes sense when one looks at the tests as implicitly embodying the specification. Given that the tests are a reformulation of the specification, it makes sense to check for consistency between the tests and the specification.

2.6 Summary

Executable formal models provide a crucial link between current design and verification practice and the grand promise of formal verification. Such models extend the impact of formal verification by contributing to the entire design and verification lifecycle. Executable formal models contribute to the early portions of the process by providing a means of rectifying specifications and expectations as well as a means for performing system level co-design. The models are maintained and acquire a service history because they are tied to crucial design and verification activities such as simulation. Finally, executable formal models are tied to the final implementation through regression testing, thus providing confidence that top-level proofs of correctness apply to the actual physical device being produced.

3. Specless Verification

While extensive informal documentation of system requirements is commonplace, it is unusual in practice to find precise formal system specifications against which a particular implementation can be proven correct. Without a formal specification, it is of course impossible to prove the correctness of a given implementation. Is there a place for formal methods in such situations?

In this section we discuss three situations that allow for the application of what we call **specless verification**. Specless verification does not necessarily mean that there is no specification. Rather, it means that the specification is implicit or derived from existing artifacts. The three situations we consider are product family verification, symbolic simulation, and self-checking tool arrangements.

3.1 Product Family Verification

Product family verification is the process of establishing that the behavior a new member of a product family is a superset of some previous member of that family. It is sometimes the case that service history and sunk cost is a deciding factor in the design of new systems. An old design with substantial service history can become a de facto specification and making the new design work just like the old becomes the design specification.

Sometimes the reason is legacy: there is a large library of software or several systems that were designed and verified around the features of the original design. Sometimes the reason is certification costs: changing one aspect of the system design would require re-certification of the entire design. Consequently, even in the absence of a formal specification, given a validated formal model of an implementation of a previous member of the product family one can formally verify that a new implementation of that design implements the old design under the conditions of interest.

3.2 Symbolic Simulation

Another form of specless verification is symbolic simulation. **Symbolic simulation** is the partial evaluation of a formal model on an incompletely specified, or symbolic, state [5]. Normally, in order to simulate the behavior of a system, one must specify the value of each of the inputs required during simulation. The result of such simulation is a set of output expressed as numeric values. Symbolic simulation, however, allows one to simulate the behavior of a system for all possible input values. The result of a symbolic simulation is an algebraic expression that represents the behavior of that system for every possible value of the input.

The primary purpose of symbolic simulation is to extract specific behavior from a complex design and to allow the designer to decide whether this behavior is correct. Symbolic simulation is truly a form of specless verification, since no formal proof is performed. The types of errors most easily detected by symbolic simulation involve extraneous dependencies and unintended side effects. Given such manual inspections, this technique works best when the desired behavior can be expressed concisely and there exists a constructive technique for mapping (abstracting) the implementation to this concise behavioral description. Without a constructive mapping to a concise description, the symbolic expression for the system functionality can become overwhelmingly complex and human inspection is rendered impractical.

Even in cases where the expressions resulting from symbolic simulations are too complex to be evaluated manually, they can still be used to perform formal regression testing. After storing the symbolic execution results from a baseline design, one can modify the design and derive a new symbolic expression. Formal proofs can then be carried out to isolate those cases in which the modified design differs functionally from the baseline design. The proofs that succeed demonstrate that the functionality of the design under those conditions remains unchanged. The proofs that do not succeed provide information to the designer concerning those cases where the design has been modified. Such results can be useful in detecting unexpected system performance changes due to localized changes in the design.

While we currently consider effective use of symbolic simulation a research topic, we note that EDA companies Innologic Systems Inc. and Chrysalis both provide commercial tool support for performing symbolic simulation of hardware designs expressed in hardware description languages.

3.3 Self-Checking Tool Arrangements

Although in this paper we are primarily focusing on establishing the correctness of an implementation of a particular design, there are other applications of formal methods that can establish the correctness of other tools used in the design chain. Such techniques are particularly interesting when organized in a self-checking tool arrangement [3]. Two examples of such applications are hardware synthesis checkers and software compiler checkers.

3.3.1 VHDL Equivalence

The use of hardware design languages such as VHDL and Verilog poses a problem to those working on high criticality hardware designs: how does one guarantee that the synthesis process does not introduce errors into the design? In particular, how does one trace the resulting design artifact back to the original design requirements? Design reviews, for example, typically take place at the source code level since it is impractical to read the output of most VHDL compilers. Exacerbating the problem, HDL compilers are complex software systems that must deal with complex languages such as VHDL and Verilog. Given this situation, there is naturally concern about the possibility of inadvertently introducing design errors during the synthesis process. How does one then justify the use of hardware description languages and sophisticated hardware compilers on safety critical systems?

One possible answer is simply to test the synthesized netlist against the original HDL. Unfortunately, exhaustive testing of the synthesized netlist of a large design might take days, weeks, months – or longer. A relatively new formal verification technique called **equivalence checking** is able to demonstrate whether two designs are functionally equivalent, and do so in time that is similar to that required by the compilation process. Such low level equivalence checkers provide confidence that the design has not in some way been corrupted during the synthesis process, and guarantees that no single fault, either in the synthesizer or in the equivalence checker, will result in an error.

It is interesting to note that, in this context, the executable formal model is the design as expressed in the HDL. The tools that have been developed to perform equivalence checking have formalized a subset of either VHDL or Verilog and used that formalism as part of their proof. Of course, VHDL and Verilog both have very mature simulation environments, so such hardware description languages might be considered among the most popular, commercially available executable formal models.

3.3.2 Compiler Checker

While equivalence checkers can play an important role in the verification of synthesized hardware, such tools are perhaps even more important when one considers software. While one can imagine exhaustive testing of appropriately designed hardware, even the simplest software is impossible to test exhaustively. Stringent testing criteria have been established to help ensure the absence of compiler bugs in compiled code destined for the most critical applications. However, equivalence checking of executable object code against its original software source code might provide a means for reducing the testing burden on software developer and improve the likelihood of detecting compiler errors [1].

A proof of the correctness of object code generated by a compiler is not as easy as performing equivalence checking on hardware. Nonetheless, because the generated object code will tend to have structure similar to that of the source code, this kind of proof may still be easier than other possible challenges, such as a proof of the correctness of the compiler itself. Such a checker would confirm that the behavior of the object code is equivalent to the behavior of the source code specification. Note that this does not guarantee that the original source code was correct, only that the compiler did not distort its meaning. The ability to perform such a proof would also provide assurance in the software development process that no single fault, either in the compiler or in the compiler checker, will result in an error.

Note that executable formal models could play an important role in the development of compiler checkers. If it were possible to synthesize executable code directly from a formal model, the model itself would be the source level specification against which the object code is compared. This would save the step of formalizing the semantics of an arbitrary programming language, such as Ada or C.

3.3 Summary

Formal verification holds great promise for addressing the complexity of modern systems, but it will require a change in the current design and verification paradigm. Specless verification, verification in which the specification is implicit or derived from existing artifacts, is a type of formal verification that does not require a radical change in the way things are currently done. Product family verification, symbolic simulation, and self-checking tool arrangements are all specless verification techniques that can provide an incremental improvement in over current practice without requiring radical changes to the design and verification process.

4. Conclusion

Avionics systems suppliers are being driven to track commercial computer trends and to provide increased functionality and performance at lower cost. In order to reap fully the benefits of these technological advances we must develop new methods for the verification of flight critical devices that provide higher degrees of assurance for increasingly complex systems while simultaneously streamlining the verification process. While classical verification has served us well in the past, it seems likely that new verification approaches will eventually be necessary to control verification costs and to provide improved safety in the face of increasing complexity.

Formal methods have the potential to provide the crucial scaling property we are looking for in verifying complex designs. Two of the issues facing the application of formal methods today are model validation and a means for incremental adoption of the techniques. We have outlined how executable formal models can be used to address the issues associated with model validation, issues such as minimization of the expectation gap, regression testing, assuring model maintenance, establishing a service history, and system level co-design. We have also illustrated several specless verification techniques: symbolic simulation, product family verification, and self-checking tools. These approaches can provide an incremental improvement to the current process while preparing the way for a more comprehensive application of formal methods in the future.

References

[1] Yuan Yu, 1992, “Automated Proofs of Object Code for a Widely Used Microprocessor”, PhD thesis, The University of Texas at Austin

[2] Steven Miller, James Potts, 1999, “Detecting Mode Confusion Through Formal Modeling and Analysis”, NASA/CR-1999-208971

[3] David Greve, Matthew Wilding, Mark Bickford, David Guaspari, 2000, “Orpheus: A Self-Checking Translation Tool Arrangement for Flight Critical Hardware Development”, Langley Formal Methods Workshop (LFM00).

[4] Matthew Wilding, David Greve, David Hardin, 2000, “Efficient Simulation of Formal Processor Models”, Formal Methods in System Design (FMSD), Kluwer Academic Publishers (to appear).

[5] David Greve, 1998, “Symbolic Simulation of the JEM1 Microprocessor”, Formal Methods in Computer Aided Design 1998, Springer-Verlag Lecture Notes in Computer Science volume 1522.

[6] http://avsp.larc.nasa.gov