SANDIA REPORT

SAND2012-7926
Unlimited Release
September 2012

Exploring Formal Verification Methodology for FPGA-based Digital Systems

Yalin Hu

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-7926
Unlimited Release
September 2012

Exploring Formal Verification Methodology for FPGA-based Digital Systems

Yalin Hu
Cyber Physical Systems
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94550

Abstract

With the Life Extension Programs (LEPs), many nuclear weapon (NW) analog electronic components are being replaced by modern digital devices, increasing system complexities dramatically. Ensuring the reliability, security, and robustness of these upgraded systems is critically important. Many custom hardware systems throughout the NW operations space rely on Field-Programmable Gate Arrays (FPGAs) to implement sophisticated logic. Effective verification, while increasing confidence, can reduce the overall effort in system debugging and testing.

This work explored Formal Verification (FV) of trusted FPGA-based hardware designs through the use of novel algorithms. The algorithms developed support the analysis of critical digital components, such as memory, with mathematical reasoning from automated theorem proving and model checking. Such verification will detect race conditions and corner cases at an early stage, eliminating system failure and instability during operation.

This work was funded by the Laboratory Directed Research and Development (LDRD) office at Sandia National Laboratories.

ACKNOWLEDGMENTS

The author would like to thank the LDRD office at Sandia National Laboratories for funding this work, and for helpful feedback during the course of the work.

I would like to thank Mary Gonzales, my project manager, for all the effort she has made to support this work.

I would also like to thank Robert Mariano and Michael Forman for their managerial support, Robert Armstrong, William Ballard, and Lyndon Pierson for their mentorship, Kevin Hulin for his contribution on implementing integrated formal verification approach, Douglas Demming and Gregory Wickstrom for their consulting on Orchestra, Vincent LoPresti for his effort promoting this work at the CTO's website and the 2012 NNSA LDRD Trilab Symposium.

CONTENTS

FIGURES

TABLES

NOMENCLATURE

ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Arrays
FSM	Finite State Machine
FTA	Fault Tree Analysis
FV	Formal Verification
GUI	Graphic User Interface
IC	Integrated Circuit
NW	Nuclear Weapon
RAM	Random Access Memory

This page intentionally left blank.

1. INTRODUCTION

Formal verification (FV) emerged as an alternative approach to traditional validation techniques, such as random simulation and directed testing, for ensuring correctness of hardware designs. While FV has been successful in many applications, such as aircraft navigation systems, cryptography, and medical devices, there has been little work at Sandia in this field for NW-related hardware systems.

Formal Verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification (property), using formal mathematical methods. For digital hardware designs, formal verification is the process of checking that the intent of the design is preserved in its implementation.

1.1. Formal Verification

Formal verification uses mathematical techniques to ensure that a given hardware design conforms to a set of precisely expressed notions of functional correctness. The basic goals are to verify that the design (1) does everything it is supposed to do, and (2) does not do anything that it is not supposed to do. Below is a list of properties that can be specified:

- Functional properties
- Timing properties
- Structure properties
- Fault tolerance properties
- Equivalence at various design stages

The two main aspects to formal verification in any design process are the formal framework and the verification techniques. The formal framework is used to specify desired and expected properties of a design. The verification techniques are used to reason about the relationship between a specification and a corresponding implementation. As a widely adopted technology to ensure correct functionality of digital systems in the Electronic Design Automation (EDA) industry, formal verification operates on (1) a design model, (2) a specification of the operational environment, and (3) a specification of the properties that the given design is intended to fulfill.

Commonly used FV techniques include:

- Equivalence checking (whether two representations of a design is equivalent)
- Model checking (whether a modeled design meets specification)
- Symbolic model checking (build propositional logic instead of graph for FSM)
- Automated theorem proving (proving of mathematical theorems with computer programs)

These techniques are targeting domain-specific problems relevant to systems in Sandia's mission areas, including NW and Energy, Climate, and Infrastructure Security (ECIS). However, the currently published and employed domain-specific verification algorithms for such applications ([14]) do not have the capability to verify critical components, such as memory blocks, of existing and future hardware designs in the NW space. The verification algorithms developed by this work support the analysis of such critical digital components with mathematical reasoning from automated theorem proving and model checking.

1.2. Digital Design with FPGAs – Simulation vs. Formal Verification

A typical flow for designing with FPGAs is shown in Figure 1.

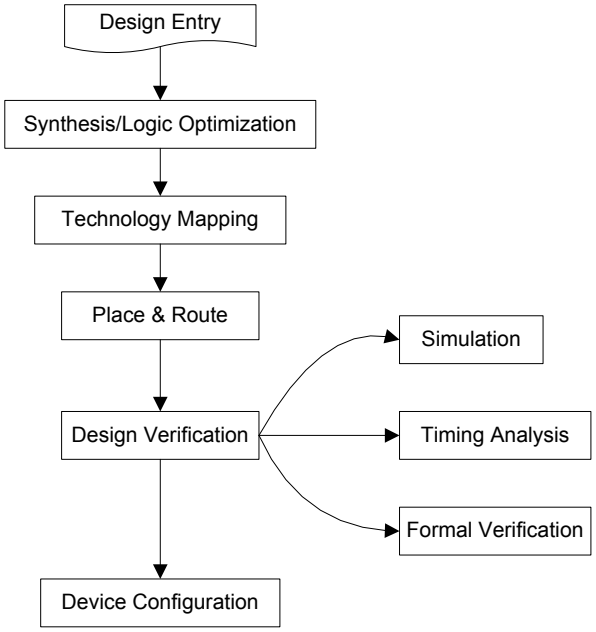

Figure 1. A typical FPGA-based design flow.

While simulation dynamically demonstrate that the design generates correct output given certain stimulus by proving correctness, formal verification statically proves that the implementation satisfies the requirements by catching fault. A simple comparison of the two techniques can be given as:

- Simulation
  - Fact: *potentially* identify the presence of a bug
  - Challenge: does not ensure the absence of a bug
- Formal verification
  - Fact: exhaustive explore all state space to uncover all incorrect behaviors
  - Challenge: identify *enough* properties to check

A simple adder circuit, shown in Figure 2, is used to demonstrate the feasibility of "complete simulation", that would test every single possible input case. Table 2 lists the time it takes to achieve a complete simulation for different sized adders, assuming the simulator can

execute one event per microsecond (capability of current state of the art simulator). It is clearly not a feasible solution to completely simulate a 32-bit adder, which is a very simple circuit. This example demonstrates the need for an alternative way to verify the design's correctness.

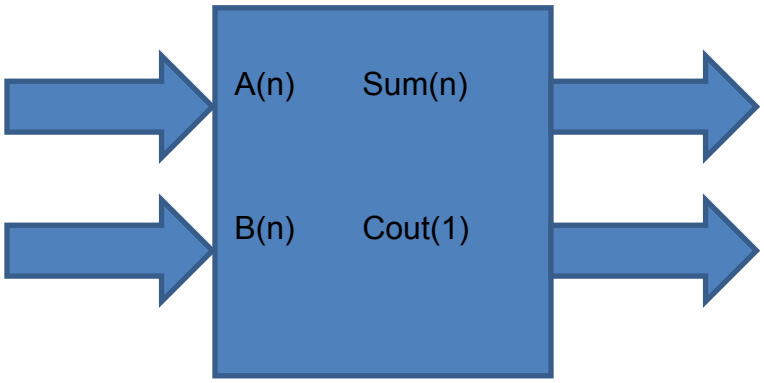

Figure 2. A simple n-bit adder circuit.

Table 1. Complete simulation runtime as adder size increases

n	Simulation run time
1	4 us
2	16 us
4	256 us
8	65 ms
16	1.2 hr
32	584,942 yr

This page intentionally left blank.

2. FORMAL VERIFICATION IN HIGH-CONSEQUENCE APPLICATIONS

Design and verification of highly complex, trustworthy hardware and software systems have always been a challenge. The use of formal verification techniques in such systems has been increasing in recent years. This paper presents a survey of the methodologies and applications that are targeted.

Mission-critical designs are those that have to work, otherwise a catastrophe could occur. Examples of such systems include: nuclear reactor control systems, automotive safety and control systems, aerospace control systems, spacecraft controllers, military communication systems, etc. [23]. Any fault in a mission-critical system leads to high consequence, and should be avoided even at significant costs. Other safety-critical systems, such as railroad/subway control systems and medical devices, also have very high requirements for reliability and stability. A fault in safety-critical systems could lead to the loss of human life or dramatic damage to the environment. Thus there are some similarities when applying formal verification techniques to mission-critical and safety-critical systems. Most of the time, these systems are required to go through stringent certification and assurance process, which is not required for pedestrian consumer products. Table 2 lists some of the widely followed safety standards for hardware and software systems. For high-consequence systems, high reliability leads to functional correctness, and high availability leads to low downtime of the system. Table 3 lists the system availability in terms of "9"s and the actual downtime to be expected.

Table 2. Safety-related hardware/software design standards

Standard	Application		Industry	Created By
	HW	SW		
DO-178B		√	Aerospace & Defense	Radio Technical Commission for Aeronautics (RTCA)
DO-254	√		Aerospace & Defense	RTCA
EN 50128		√	Railway Transportation	European Committee for Electrotechnical Standardization (CENELEC)
FSDA	√		Cryptographic Equipment	National Security Agency (NSA)
IEC 60601	√		Medical Equipment	International Electrotechnical Commission (IEC)
IEC 60880		√	Nuclear Power	IEC
IEC 61508	√	√	Heavy Equipment and Energy	IEC
ISO 26262	√	√	Automotive Electronics	International Organization for Standardization (ISO)

Table 3. System availability and downtime

Availability (%)	Downtime		
	Weekly	Monthly	Annually
90% “one 9”	16.8 hours	72 hours	36.5 days
99% “two 9s”	1.68 hours	7.2 hours	3.65 days
99.9% “three 9s”	10.1 minutes	43.2 minutes	8.76 hours
99.99% “four 9s”	1.01 minutes	4.32 minutes	52.56 minutes
99.999% “five 9s”	6.05 seconds	25.9 seconds	5.256 minutes
99.9999% “six 9s”	0.605 seconds	2.59 seconds	31.5 seconds

The complexity of mission-critical systems is continually increasing. In order to meet new challenges the systems need to be very robust and reliable. With the emergent technology in Integrated Circuits (IC), Field Programmable Gate Arrays (FPGAs) are becoming more and more popular, both in traditional digital systems designs, and in mission-critical system components [11][24]. Currently FPGAs can be delivered in 28nm node, with programmable logic blocks, configurable memory blocks, complex peripherals, and even embedded hardware Intellectual Property (IP) blocks. FPGAs are attractive because they are flexible, reconfigurable, and easily to designed with vendor-provided tool software.

To ensure security of mission-critical systems, sensitive Intellectual Properties (IPs) can be protected better with FPGAs compared to custom hardware. It is harder for attackers to target a specific IP or design, if the IP or design is not loaded onto the device until after it is manufactured. One challenge for ensuring system security with FPGA designs is the introduction of vulnerabilities. Often there are design “hooks” which are intended for future enhancement and possible optimization. But they can be used to introduce unintended functionalities, sometimes could be malicious. Other possibilities include design-tool subversion, trustworthiness of foundries, and at the final physical netlist protection.

Mission-critical systems often need to operate in harsh environment involving extreme temperature and radiation. Such hostile environment makes it infeasible to do a dynamic test of the design. At the same time when silicon becomes denser with smaller transistors, they are more sensitive to lower level of radiation. This trend has led to the need of more robust radiation-hardand radiation-tolerant designs. Technologies such as Triple Modular Redundancy (TMR) are introduced to mitigate radiation-induced errors. Being able to formally verify designs facing such environment is still a challenge.

In addition to rad-tolerant characteristics, mission-critical systems also need to be fault-tolerant under various circumstances. Frequently, faults are non-deterministic, making exhaustive testing infeasible and the verification task harder.

Traditionally, hardware designs are validated through simulation and emulation, while software systems are validated through code reviews and dynamic testing. As a mature technology, a good simulation test bench could demonstrate the presence of a design bug (i.e.

assure the design does what it is supposed to do), but can never ensure the absence of a design bug (i.e. assure the design does not do what it is not supposed to do).

Formal verification for both hardware and software systems provides high level of confidence, automation, and efficiency. As an example, NASA [20] highly recommends applying formal methods for safety-critical software development and verification.

2.1 Design and Verification of High-consequence Systems

A typical design flow that involves formal verification is shown in Figure 3**Error! Reference source not found.**. Specifications (system, functional, property) are normally described in plain text along with block diagrams. Implementation is done in two general ways: hardware description language such as Verilog and VHDL, software programming language such as C/C++. Property modeling can be done with formal semantics. The verification framework then generates the result, which can be used to modify the implementation or specification. Mission-critical and safety-critical systems have much rigorous requirement to be satisfiable [1][10][19].

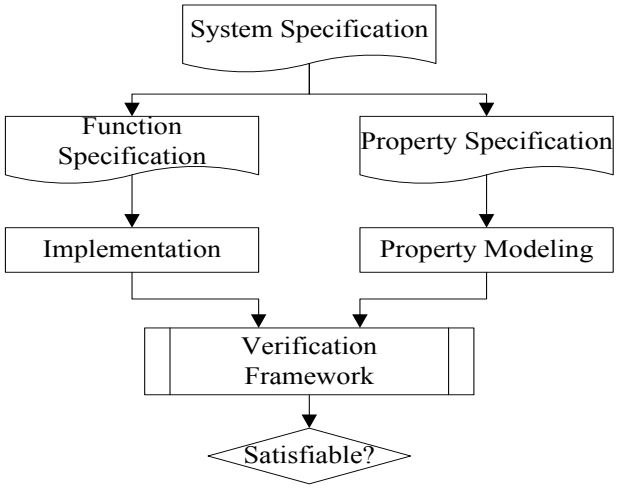

Figure 3. A typical design process with formal verification.

Table 4. Common Formal Specification Language

Formal Language	Description	Application
Cryptol [2]	Domain Specific Language (DSL)	Cryptography
Esterel [6]	Synchronous language with formal semantics	Aerospace
LOTOS [4]	Language of temporal ordering specification	Communication Protocols
Promela [4][6]	Process meta language	Aerospace, Medical Devices, Spacecraft
SIGNAL [5]	Block-diagram based synchronous language	Real-time System Design
SMV [4][17]	Synchronous language	Rail Transportation

Table 5. Common Formal Verification Framework

Formal Framework	Description	Supported Language
Cadence SMV	Deterministic	SMV, Verilog
CADP	Probabilistic	LOTOS
Cryptol Tool	Deterministic	Cryptol language
SCADE	Deterministic	Esterel
NuSMV	Deterministic	SMV
ROMEO	Deterministic	Time Petri Nets
SPIN	Deterministic	Promela

2.2 Case Studies

This section presents several case studies to demonstrate the application of formal methods and formal verification for mission-critical and safety-critical systems. There are both hardware and software applications and each one is summarized for their modeling language, formal framework, unique contribution, and the impact on the applications.

2.2.1 FPGA-based Aerospace hydraulic Monitoring System

Hammarberg and Nadjm-Tehrani [6] published an application of formal verification in an aerospace hydraulic monitoring system. The system detects hydraulic leakage inside a JAS 39

Gripen multi-role aircraft. This is a critical system, because an electrical fault could lead to the complete loss of control of the aircraft in worst case.

The co-designed system contains one software component and two FPGA-based hardware components. The purpose of using two separate FPGA devices is to increase redundancy in the system, making it more fault-tolerant.

Traditional Fault Tree Analysis (FTA) was not tractable for such a complex system, a formal verification based design model was implemented as shown in Figure 4. Esterel Studio provides two model checkers, one based on Binary Decision Diagrams (BDD) and another one based on propositional satisfiability (SAT). The SAT based solver was chosen for this particular design. The main goals of this verification are (1) verify single fault tolerance of the system, and (2) identify potential double fault combinations.

In order to achieve co-design and co-verification, all three components and nets that connect them are modeled in Esterel. The top level structure of a developed verification bench is shown in Figure 4. The highlighted verification bench is written as plug-in modules. These modules are solely for verification purposes, and are ignored during design code generation and system implementation. The output from the verification bench ("Alarm Signal") indicates whether there is a fault detected or not.

Possible hardware faults, such as bit flipping on silicon (FPGA or processor) can be caused by environmental factors, such as radiation, extreme temperature, and sudden power change. A fault switch is inserted to serve as a fault injector. The objective of such fault switch is to indicate whether a formally verified safety-related property would hold if an environment fault presents. An example of environmental fault modeled in this application is the arbitrary malfunction in either of the FPGA devices.

Esterel's built-in model checker does a good job in this application, especially with the support of user-provided constraints. The verification results are impressive by proving: (1) the components do not contain design faults causing violation of the safety property; (2) no combination of the potential faults can cause violation of the property; (3) no single random fault can cause violation of the property; and (4) the only double fault violating the property is when the software component and one of the FPGA component are faulty.

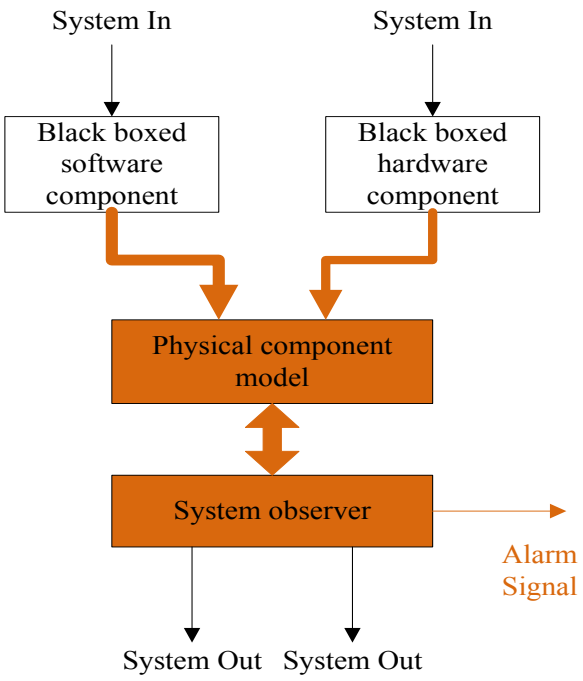

Figure 4. Hardware/software co-verification model.

Another advantage of this approach is the short run time. The model checking takes a few second to run, while a simulation test bench with descent coverage can easily run in hours, even days for such a complex system.

The authors also demonstrated a comparison between manually created and automatically generated VHDL design for another smaller safety-critical application. The example is the PID controller used in a brake control system for an aircraft arrester system. The same design is implemented in two ways: (1) manually created a VHDL design, and (2) automatically generated VHDL code from Esterel model. Both designs are then run through the FPGA design flow (synthesis, place & route, timing analysis). The manual design wins in both area (logic usage on device) and speed (Fmax of the design). However, Esterel generated VHDL design has smaller size (lines of code) in general.

This is a case study that demonstrates a practical design process for mission-critical system. The design is specified at a high abstraction level, which is implementation independent. With the built-in verification bench, it successfully detected random faults that are of high-consequence. The tradeoff is the implementation efficiency, which could lead to the need of a bigger and faster FPGA device. This tradeoff, however, can be easily justified for such applications.

2.2.2 Multi-thread Control Module for Space Craft

Havelund, Lowry and Penix [8] published a formal analysis case study of a space craft controller. The software to be verified is a component of NASA's Remote Agent (RA), an artificial intelligence (AI) based space craft control system architecture. The module is developed in LISP programming language and is multi-threaded. The Remote Agent itself is a mission-

critical application as it is the first AI based software that demonstrated the complete control of a space craft.

SPIN is chosen to be the model checker for this application, as it supports verification of finite state asynchronous process systems. A domain specific language (DSL) named Executive Support language (ESL) is used to specify the bottom layer of the module. The verification scheme is shown in

Figure 5. By abstraction, the original LISP program is reduced to a finite state system described in Promela, which is a C-like programming language used by SPIN. This abstraction is a critical step for efficient verification, as it makes feasible to create bounded state space. Two properties are fed into SPIN, described either as Promela assertion or Linear Temporal Logic (LTL) formulae. SPIN is then run to verify if both properties are satisfied.

Outputs from SPIN indicate both properties are not satisfied, with four software errors being identified immediately. With the error trace provided by SPIN for the four bugs, a design flaw (duplicated execution) is also identified. The result is the discovery of five hard-to-find errors, which would manifest themselves only under very particular circumstances involving precise timing. However, these errors are also of very high consequence. A real incident happened during an operation of RA in space, where the thrusting did not turn off as requested, resulting in an immediate action to put the space craft in stand-by mode. This happened when RA was onboard the DEEP_SPACE 1 space craft. It turned out the cause of the failure was an identical error identified by SPIN, but it existed in another module that was not formally analyzed.

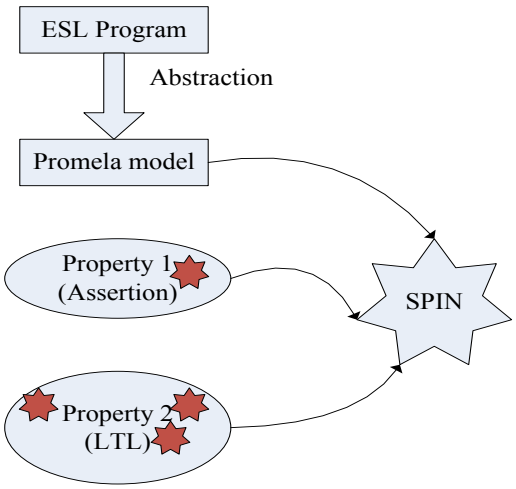

Figure 5. Verification scheme of control module (red stars indicate identified violations).

This work focused on the development of Promela model. The longest run time of SPIN is less than 1 minute. The result from this work had a major impact on the RA design team, with increased confidence of the delivered software.

This case study demonstrates a very successful application of SPIN's partial order reduction algorithm and state compression.

A related work is reported in [9] that formally analyzes the concurrent software system before and after flight.

2.2.3 Model Checking for Fault Tolerant Systems

Schneider, Easterbrook, Callahan, and Holzmann [22] published a model checking case study to verify a fault-tolerant embedded space craft controller, which is a real-time control system handling critical control sequences. The key contribution of their work is the effective verification based on partial specification. The higher abstraction level is achieved by ignoring unnecessary details, while keeping main properties. Due to the complexity of this application, reducing the state space is crucial to ensure the feasibility of model checking for critical system requirements.

The implementation of this verification scheme is shown in

Figure 6. A critical sequence is executed on a deterministic model, with non-deterministic faults injected. Three unrecoverable faults, each indicating a design problem, were identified by this verification scheme.

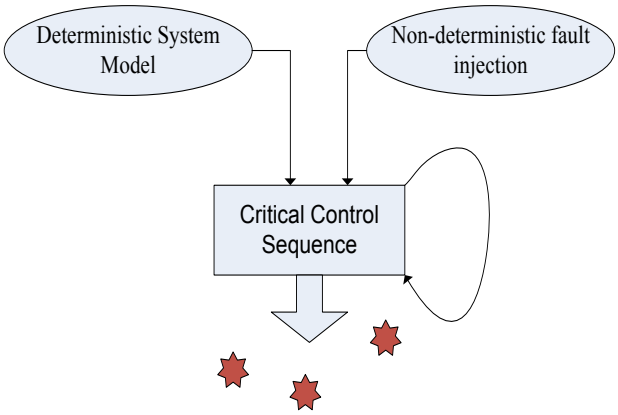

Figure 6. Fault injection model (red stars indicate identified faults).

With proper modeling, selection of reliable model checker (SPIN), and effective state space reduction, this case study delivered good results in very short run time. The exhaustive examination of selected partial specification runs for about 3 minutes, whereas the run time for full specification is estimated to be 10^{12} years. The three design problems identified could lead to potential fault control sequence. Another notable contribution of this case study is the parallel design-verification process, which allows prompt feedback and dynamic modification of both design and specification, as shown in Figure 7.

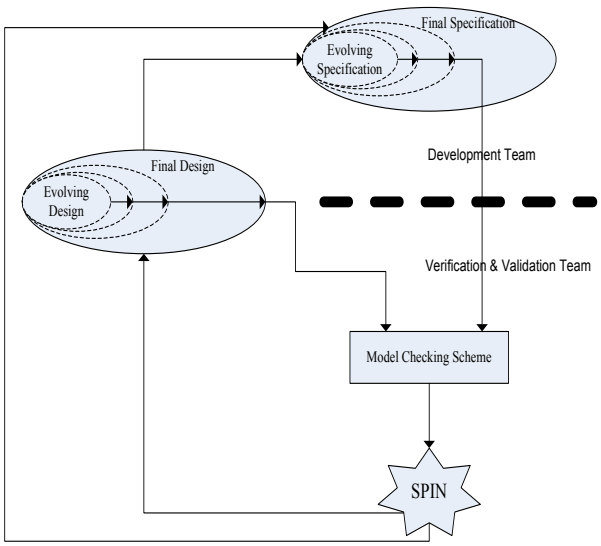

Figure 7. Co-design and verification process.

2.2.4 Cryptographic Applications

Cryptographic applications require very high level of assurance, performance, reliability, and security. Historically programmable logic has not been widely used because of the challenge to support multiple levels of security and handle isolated redundancy. FPGAs are suitable for implementing cryptographic algorithms because there are a lot of bit-level operations, such as shifting and permutation. With the growing logic density and performance of FPGA devices and development tools [2][7][16], it is now feasible to implement a cryptographic system (even Type I) on a single FPGA chip. However, such designs have to be partitioned in a way that isolated subsystems do not leak information to each other. For example, strong isolation is expected to segregate plain text (red text) and cipher text (black text). The communication between these partitions has to be tightly controlled to meet the National Security Agency's (NSA) Fail Safe Design Assurance (FSDA) requirements.

The primary goal of verifying a cryptographic system is to ensure the risk of compromising its integrity caused by a hardware fault is minimized. Lewis, Hoffman, and Browning [14] published a design and verification flow for implementing a single FPGA-based cryptographic system. This flow leverages a Domain Specific Language (DSL) named Cryptol and tools to support it. Cryptol is a functional description language designed for the NSA as a public standard for cryptographic algorithm specification. It allows the user to create specifications at a much higher level of abstraction compared to structural or behavioral description of digital systems. Even the final implementation is physically on a FPGA, the design process is independent of hardware features and detailed configuration. Compared to any hardware design language (HDL) such as Verilog or VHDL, Cryptol enables the designers to focus on the functional level.

The formal verification feature provided by Cryptol tools focus on equivalence checking. Based on SAT and Satisfiability Modulo Theories (SMT), equivalence checking can be done at various design stages throughout the design process. Similar to Esterel used in an earlier cast study, Cryptol can also generate lower level VHDL designs, which can then be synthesized, placed and routed on a FPGA device. One attractive feature of Cryptol is that the generated VHDL code comes with a formal proof to ensure the functional equivalence. Results have shown

that the auto-generated implementations are comparable or better compared to manually written Verilog/VHDL implementation, in terms of area and speed. With the introduction of Signal-Processing Intermediate Representation (SPIR) model, Cryptol provides a nice mixture of easy development at higher level and easy access to lower detailed implementation information. An overview of the design and verification flow for Cryptol is shown in Figure 8.

This case study demonstrates an effective co-design/verification flow for systems with very high-assurance and high-reliability.

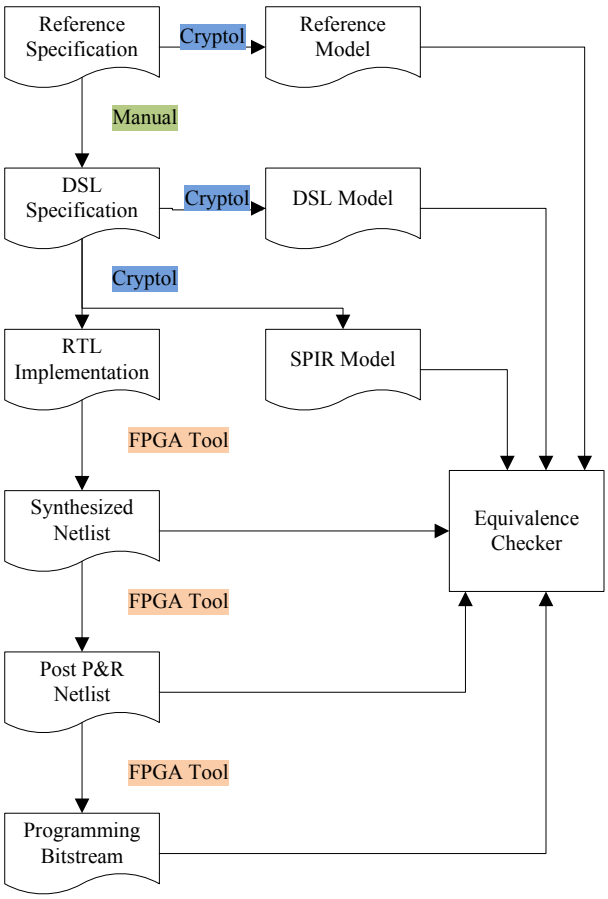

Figure 8. Design/verification flow provided by Cryptol.

2.2.5 Control Software for B-2Test Program

Chang et al. [3] published a case study in which formal method has made significant contribution to the verification of a mission-critical software system. The targeted application is the Tape Copy and Management System (TCAMS) built for United States Air Force. TCAMS is an important part of the B-2 bomber testing program, which handles enormous amount of flight data during testing. Due to the complexity and extreme requirements of B-2, TCAMS has to be exceptionally reliable.

The overall process of this application is shown in Figure 9. Continuous software verification was made possible through a matrix development model [Tomayko96]. At the requirement analysis stage, formal method was combined with object-oriented analysis to model

system specification. At the high level design stage, formal method was used to describe data flow, serial processing ordering, and process parallelization. At the integrated test stage, formal methods were used to create test procedures and validation criteria.

Without giving the details of applied the formal method and formal verification technique, the authors confirmed that verification of the system was enhanced. The final delivered system achieved exceptional quality and reliability, proven by continuous successful operation upon deployment.

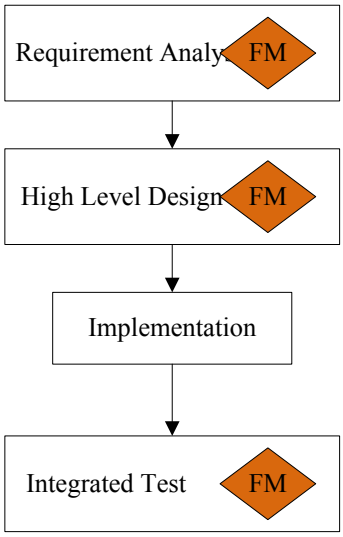

Figure 9. TCAMS design process coupled with formal methods.

2.2.6 Formal Modeling and Analysis Military Avionics Systems

A collaborative research project between the University of South Australia and Australia's Defense Science and Technology Organization aiming at modeling and analyzing avionics mission systems is another success story [21]. The application is an avionics mission system (AMS) for AP-3C Orion maritime surveillance aircraft. The complexity of such systems comes from the large number of hardware and software components, and their integration.

The key contribution of this work is to combine state space methods and Colored Petri Nets (CPN) to reason system properties. Due to the vast number of subsystems and components, complexity can only be managed by higher level of abstraction. CPN was chosen because (1) it provides primitives for modeling concurrency and synchronization; (2) it provides primitives for modeling data manipulation; (3) it is parameterized and can easily be shared for different systems; (4) it supports hierarchical design specification; and (5) it is executable thus can be simulated. In this case study, CPN was used to model different levels of abstraction, allowing formal specification of communications between various subsystems and the avionics bus.

The most challenging tasks for AMS is task scheduling and data transfer management. Task scheduling problem was handled by a state space search approach in this application. If a path from an initial state to a final state is found, then a schedule has been successfully identified. Compared to traditional scheduling algorithms, this approach creates a single model that can be used for both task scheduling and property specification.

All data transfer in this case study happen on a shared data bus, making it critical to ensure the safety and accuracy of data. In this system, data can be transferred between sensors, central control unit, display and storage. The CPN model allows a high level description of the entire data management network.

A remaining challenge for this cast study was the state space explosion problem. As the number of system tasks increase, the state space of the CPN model grows significantly. In the original publication, the author proposed to investigate more advanced methods for reducing the state space in similar models.

Overall, this case study represents an effective formal modeling and analysis approach for a real mission-critical application. The result of this work was the high confidence level of the AP-3C aircraft mission system, which contributes to the aircraft's major missions,, including anti-subsurface/surface warfare, surveillance, search/rescue, and maritime strike.

2.2.7 Aircraft Safety-critical Software

A recent publication by Yin, Liu, and Su [25] reported a formal verification technique for an aircraft safety-critical software (ASCS) – an aircraft inertia/satellite navigation system. Realizing the general effectiveness of extended finite state machine (EFSM) in formal verification of embedded software systems and its incapability to meet real-time requirements of the ASCS, this work introduced a real-time extension of EFSM, named RT-EFSM.

The developed RT-EFSM model is used to describe the following properties of ASCS: (1) behavior (static and dynamic); (2) real-time characteristics; (3) complex state transition. The same model is also used to solve the state explosion problem and ensure the consistency of ASCS models.

The validation of RT-EFSM involves checking of several critical properties of the model, as shown in Figure 10. Once validated, the model can be used to generate valuable test sequence. A time extended unique input/output (UIO) sequence was introduced to accommodate the real time system. During the test sequence generation, depth-first search tree is constructed for easy traversing and improving test coverage.

The developed formal approach was applied to an aircraft inertia/satellite navigation system. It is reported that the verification methodology is very effective for this application.

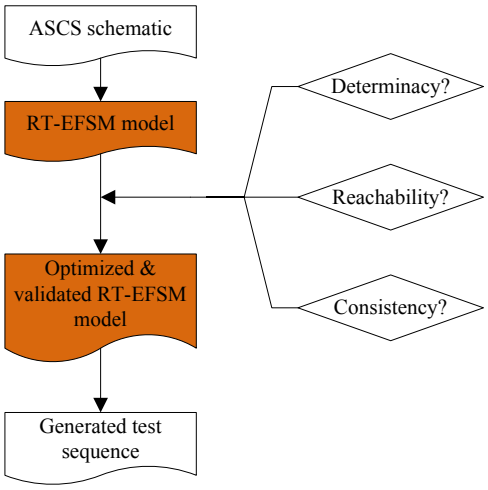

Figure 10. RT-EFSM based verification for ASCS.

2.3 Conclusion

Formal verification has been used in many applications as an alternative to traditional testing approaches – simulation for hardware designs and dynamic testing for software systems. With the extreme requirements of the reliability of mission-critical and safety-critical systems, the ability to effectively verify the design throughout the design cycle is highly desirable.

The cases studied include both hardware systems and software systems. For hardware systems, the survey focuses on designs implemented with FPGA, because of its flexibility, reconfigurability, and growing popularity in the targeted applications. The surveyed cases applied various formal methodologies to accommodate different applications, including both equivalence checking and formal model checking.

Additional application of formal verification in safety-critical systems include railway interlocking systems [15], hybrid emergency control components [13], medical device software systems [12] etc.

As formal methods research advances, more examples of success will be published. More advanced formal tools are also expected from the Electronic Design Automation (EDA) industry to further enhance verification. A recent success story is the joint effort between Northrop Grumman Italia and Mentor Graphics to achieve DO-254 compliance [18].

This page intentionally left blank.

3. COMPLETE FORMAL VERIFICATION OF STATEFUL DESIGNS FOR HIGH-CONSEQUENCE SYSTEMS

In certain high consequence systems, the requirement that safety and liveness properties are upheld is of paramount importance. The most common method for determining whether a system implementation is working to spec is simulation based validation. A large set of test inputs and expected outputs must be created in an attempt to cover all runtime paths that the system may exhibit. The system is then run on the given inputs and checked. Correct operation during validation is then used to claim that the system works correctly and is ready to be put into production. However, such tests cannot feasibly be exhaustive, and the reliance upon simulation based validation for producing high consequence systems is known to be a costly mistake [29][31].

Formal verification aims to eliminate these consequences by offering mathematically and logically sound techniques for determining whether a design implementation is specification adherent. Since its inception, two approaches have taken hold as viable techniques for formal verification: Model Checking (MC) and Automated Theorem Proving (ATP). MC is the exhaustive examination of a system's reachable states that ensures desired properties hold. ATP is the logical derivation of desired properties from a mathematical definition of the system implementation and a collection of known axioms. Each technique has its own strengths and weaknesses (as shown in Table I) and neither can really be considered a cure-all for the formal verification problem.

Table 6. Automated Theorem Proving vs. Model Checking

	Automated Theorem Proving	**Model Checking**
Strengths	• Ability to handle very complex systems • Expressive logic • Generation of machine checkable proof	• Easy generation of model from HDL source • Automatic verification • Generation of counter examples
Weaknesses	• Requires human input • No counter example • Not automated	• Design size limitation • Not feasible for complex data path

A key observation made when comparing ATP and MC is that one's weakness is the other's strength. Where ATP is unable to perform without human intervention, MC requires no human oversight; and where MC cannot handle complex systems, ATP is not limited by the system's complexity. Since these two techniques are complementary, an obvious solution would be to combine the best features of each and create a completely automatable verifier that is not limited to simple systems.

Much work has been done to this end, and while progress has been made to combine the techniques of ATP and MC into a hybrid verification tool, success has been limited to problems that are not easily generalized.

Very early work in combing ATP and MC attempted to partition a system into properties that are control intensive (to be used with MC) and data intensive (to be used with ATP) [35]. The limitation of such an approach is that most systems, especially high consequence systems, have very complex interactions between these two categories, making partitioning infeasible or very hard. Some alternative approaches emerged to supplement model checking with proof assistants that aim to decompose a complete verification into several model-checkable subtasks [33]. Examples of decomposition rules include temporal splitting, data abstraction, and compositional verification. Among the listed, abstraction is a commonly used technique that can reduce the verification of a complete system to the verification of an abstract system.

Another verification approach aims to loosely integrate MC and ATP under into deductive environment [36]. This environment provides capabilities such as modular debugging and verification through abstraction and MC. The major obstacle for a tight integration of MC and ATP is the successful abstraction across domains and discovery of good abstract representations. The approach takes advantage of the automation of MC in combinatorial logic from NuSMV and avoids the state explosion problem by decomposing the model into small function preserving partitions. In order to maintain soundness, a theorem prover (ACL2) is employed and consequently, the verification results are able to be scaled up to arbitrarily large models. This approach is general enough that it can be applied to other digital systems.

RAM is chosen as a case study because of its wide application, especially in high consequence systems. With state-of-the-art semiconductor process technology, memory design and verification has drawn a lot of attention in both analog and digital aspects [32]. Verification of memory has been an important and challenging problem. Memory is unique because (1) there are normally a very large number of cells; (2) each of these cells has identical functionality and controlled by the same control signals; and 3) there are generally many structural symmetries in RAM architectures.

Verification of memory started with switch-level simulation [28], which works very well for small sized memories. Later on different techniques have been published, such as symbolic trajectory evaluation (STE) of memory arrays [34] and bounded model checking of embedded memories [30]. The STE based verification is essentially a form of symbolic simulation and is able to overcome the infeasible simulation coverage issue by reducing the system model – taking advantage of the structural symmetry of RAM. Bounded model checking (BMC) made the handling of large embedded memory designs feasible through an effective abstract model [30]. In this approach, each memory bit is abstracted and constraints are added at every analysis step.

However, because BMC is employed, soundness is not guaranteed for general systems. This work create a novel framework for verification of stateful hardware systems and employ its utility in verifying a RAM design, emphasizing the importance of automation and soundness.

In section 3.1, a RAM model is formally define as a Kripke structure and demonstrate the pitfalls of straight forward MC. In section 3.2, a decomposition approach is presented and its soundness is proven. Verification results are presented in section 3.3 and section 3.4 concludes with suggested future research in.

3.1 Random Access Memory (RAM)

Before discussing the formal verification of a RAM system, the system has to be formally defined in a way that is easy to understand.

3.1.1 Definitions

A Kripke structure that reflects the semantics of a generic RAM implementationis used.

Figure 11 formally describes the finite state ω-automaton that is used in model checking. A state is defined as a 5-tuple (I,A,T,O,Y) where I represents the input value, A the input address, T the Read/Write control bit, O the output value, and Y an ordered N-length list of M-bit values representing the values being stored within the RAM.

$$S = \{(I,A,T,O,Y) | I \in \mathbb{Z}_{2^{64}}, A \in \mathbb{Z}, T \in \{READ, WRITE\}, O \in \mathbb{Z}_{2^{64}}, Y \in \mathbb{Z}_M^N\}$$

$$S_0 = (0,0,\perp,0,0^N)$$

$$R \subseteq S \times S = \{(S,S') \mid T_S = READ \wedge read(S,S') \vee T_S = WRITE \wedge write(S,S')\}$$

$$L: S \to 2^{AP} = L\big((I,A,T,O,Y)\big) = \{input = I, address = A, control = T, output = O\} \cup \{memory_i = Y_i \mid i \in \mathbb{I}_N\}$$

$$read\big((I,A,T,O,Y),(I',A',T',O',Y')\big) = \left(\bigwedge_{i \in \mathbb{I}_N} (Y_i' = Y_i)\right) \wedge (A \in \mathbb{Z}_N \to O' = Y_A) \wedge (A \notin \mathbb{Z}_N \to O' = 0)$$

$$write\big((I,A,T,O,Y),(I',A',T',O',Y')\big) = \left(\bigwedge_{i \in \mathbb{I}_N} (i \neq A \to Y_i' = Y_i) \wedge \big(i = A \to Y_i' = mask_N(I)\big)\right) \wedge (O' = O)$$

$mask_N(X) = X \,\&\, (2^N - 1)$, Where "&" is the bitwise "AND" operation.

Figure 11. Formal definition for RAM Kripke structure.

The transition relation between states is defined by the Boolean relations $read$ and $write$.

1) Read. For the $read$ relation, two states S and S' are related if the following three statements hold:

1. $\forall_{i \in \mathbb{I}_N} Y_i' = Y_i$
2. $A \in \mathbb{Z}_N \rightarrow O' = Y_A$
3. $A \notin \mathbb{Z}_n \rightarrow O' = 0$

Semantically, the $read$ relation ensures that when a state transition is initiated by a read operation, the next state must (1) maintain the integrity of values being stored and should update the output value to either (2) reflect the value being stored in memory if the address is valid or (3) to 0 if the address line is not valid.

2) Write. For the $write$ relation to hold, two states S and S' must satisfy the following expressions:

1. $\forall_{i \in \mathbb{I}_N} (i \neq A \rightarrow Y_i' = Y_i)$
2. $\forall_{i \in \mathbb{I}_N} (i = A \rightarrow Y_i' = mask_N(I))$
3. $O' = O$

These rules ensure that when a state transition is initiated by a write operation, the next state should (1) maintain integrity of values Y_i where $A \neq i$, (2) update the value Y_i where $A = i$ to $mask_N(I)$, and (3) ensure that the output value does not change. Notice that these rules were crafted in order to preserve the safety of the system. That is, the $mask$ function ensures that only values of the proper bit-width are stored in memory, and the update step implicitly ensures that writes to illegal addresses do not corrupt the memory content.

3.1.2 Specifications

Next the formal specifications that the model checker verifies is described. Here, the naive specifications expressed in Computation Tree Logic (CTL) is presented. Optimizations for reducing complexity will be presented in the next section. Two liveness and one safety properties are enforced:

1) *Liveness*. The first liveness property checked is whether the implementation correctly implements the read operation. The property Read Liveness (RL) is defined to be: *If the status bit = $READ$ and the address = A, then in the next state, the output should be $Y[A]$.*

$$\mathbf{AG}(addr = a \wedge status = R) \rightarrow \mathbf{AX}\ output = mem[a]; \quad a \in \mathbb{Z}_N$$

The second liveness property ensures that the write operation is correct. *Write Liveness* (WL) is defined as: *If the status bit = $WRITE$ and the address = A and the masked_input = I, then if A is a valid address, in the next state $Y[A] = I$ will be true. Furthermore, if address $\neq A'$ then in the next state, $Y[A']$ will equal the current value of $Y[A']$.*

$$\boldsymbol{AG}\ (addr = a \wedge status\ = WRITE \wedge masked\_input = i) \rightarrow \boldsymbol{AX}\ mem[a] = i; a \in \mathbb{Z}_N, i \in \mathbb{Z}_{2^{64}}$$

$$\boldsymbol{AG}\ (addr \neq a \wedge status\ = WRITE \wedge mem[a] = k) \rightarrow \boldsymbol{AX}\ mem[a] = k; a \in \mathbb{Z}_N, k \in \mathbb{Z}_{2^{64}}$$

2) Safety. The only safety property enforced is that at all states, the values stored in memory should be members of a specified range of integers defined by the value M in the definition. In the specifications, *Safety* is stated to be: *For each memory address A, the value stored at Y[A] should be in the range of values 0 to* $2^M - 1$.

$$\boldsymbol{AG}\ (0 \leq mem[a] \wedge mem[a] \leq\ 2^M - 1);\ a \in \mathbb{Z}_N$$

3.1.3 Optimized specifications

In order to obtain viable runtime results for model checking, *Write Liveness* property needs to be rewritten to avoid $O(N \cdot 2^M)$ specifications. This is accomplished by performing a bit-level comparison across the M bits of data values. The improved specifications are shown in Figure 12. Using this optimization, the same properties in $O(N \cdot M)$ specifications can be covered.

$$\boldsymbol{AG}\ \Big(\big((mem[a]\ \&\ (1 \ll b)) > 0\big) \wedge (status \neq WRITE \mid addr \neq a)\Big) \rightarrow \boldsymbol{AX}(mem[a]\&\ (1 \ll b) > 0)$$

$$\boldsymbol{AG}\ \Big(\big((mem[a]\ \&\ (1 \ll b)) = 0\big) \wedge (status \neq WRITE \mid addr \neq a)\Big) \rightarrow \boldsymbol{AX}(mem[a]\&\ (1 \ll b) = 0)$$

$$\boldsymbol{AG}\ (input\ \&\ (1 \ll b) > 0 \wedge status = WRITE\ \&\ addr = a) \rightarrow \boldsymbol{AX}\ (mem[a]\ \&\ (1 \ll b) > 0)$$

$$\boldsymbol{AG}\ (input\ \&\ (1 \ll b) = 0 \wedge status = WRITE\ \&\ addr = a) \rightarrow \boldsymbol{AX}\ (mem[a]\ \&\ (1 \ll b) = 0)$$

$$a \in \mathbb{Z}_N, b \in \mathbb{Z}_M$$

Figure 12. Optimized liveness specification for memory writes .

3.1.4 Limitations

Despite the efforts to express the RAM model in a way that would make the model checking problem tractable, the fact of the matter remains that RAM is a stateful system and subject to the state explosion problem. Unable to model check RAM of size larger than 100 bytes, other approaches are investigated to solve the problem. Using a theorem prover, it would be trivial to verify the properties, however, as a completely automatable system is required, and thus direct theorem proving would not suffice.

Based on the thinking of a hybrid approach, an idea was developed or decomposing RAM into smaller pieces and model checking the pieces individually. While this seems trivial, the implications of being able to reduce an intractable problem into smaller tractable parts were very appealing. The first step would be to formally prove that such an approach would be work.

3.2 Decomposition of RAM for Formal Verification

The decomposition used to abstract the RAM model is fairly straightforward. Given a RAM, it is divided into arbitrarily small pieces and model check each piece individually. It is conjectured that the conjunction of results from these smaller pieces is equivalent to the overall result that would be obtained from model checking the original RAM.

In order to maintain soundness in the RAM verifier while taking advantage of a decomposition property, it was necessary to first ensure that the decomposition step was sound and did not affect the system's validity. The ACL2[26] theorem prover is used to prove the conjecture. Furthermore, it is proved that two smaller RAMs that satisfy the properties could be concatenated together and the resulting RAM would also satisfy the properties. Finally, the mapping for `READ` and `WRITE` operations from the large RAM onto the decomposed pieces is defined and their semantic equivalence is proven.

3.2.1 Decomposition Proof

$$(m, k, s) \qquad \text{RAM}$$
$$k \in \{0,1\}^* \qquad \text{mask}$$
$$s = \|m\| \in \mathbb{Z}^+ \qquad \text{size}$$
$$([0,0,\ldots,0], \{1\}^N, N) \qquad \text{Initial RAM}$$

$$\frac{(adr \geq 0) \wedge (adr < s)}{read\big((m,k,s), adr\big) \rightarrow m_{adr}} \; (Memory\ Read)$$

$$\frac{(adr \geq 0) \wedge (adr < s)}{write\big((m,k,s), adr, val\big) \rightarrow \big(m_{(0..adr-1)} :: (val \,\&\, k) :: m_{(adr+1)..(s-1)}, k, s\big)} \; (Memory\ Write)$$

$$\frac{(r > 0) \wedge (r < n)}{decompose\big((m,k,s), r\big) \rightarrow \Big(\big(m_{0..(r-1)}, k, r\big), \big(m_{r..(s-1)}, k, (n-r)\big)\Big)} \; (Decomposition)$$

$$\frac{}{compose\big((m_1,k,s_1),(m_2,k,s_2)\big) \rightarrow (m_1 :: m_2, k, s_1 + s_2)} \; (Composition)$$

$$\frac{(adr \geq 0) \wedge (adr < s_1 + s_2)}{read_{decomp}\big((m_1,k,s_1),(m_2,k,s_2), adr\big) \rightarrow \big(adr < s_1 ?\ read\big((m_1,k,s_1), adr\big) :\ read\big((m_2,k,s_2), adr - s_1\big)\big)} \; (Decomposed\ Read)$$

$$\frac{(adr \geq 0) \wedge (adr < s_1 + s_2)}{write_{decomp}\big((m_1,k,s_1),(m_2,k,s_2), adr, val\big) \rightarrow \big(adr < s_1 ?\ \big(write\big((m_1,k,s_1), adr, val\big), (m_2,k,s_2)\big) :\ \big((m_1,k,s_1), write\big((m_2,k,s_2), adr - s_1, val\big)\big)\big)} \; (Decomposed\ Write)$$

Figure 13. Syntax and operational semantics for RAM as modeled in ACL2.

In ACL2, memory is modeled as a 3-tuple (m, k, s) where m is an ordered list of size s and k is the value mask that is applied upon memory writes.

The syntax and operational semantics for the model are defined in Figure 13. In the remainder of this section, 1) property adherence for the model is proven, 2) the equivalence of the decomposed operations with their corresponding simple operations on the original memory is proven, and 3) a soundness proof for decomposed property verification is concluded. In the following theorems, let $R = (m, k, s) \in RAM$, $p \in \mathbb{Z}_s$.

Theorem 1 (Liveness for Read and Write). If the read operation is invoked with a valid memory address, then the resulting output should be the corresponding value located in memory. Similarly, if the write operation is invoked with a valid memory address, then the resulting memory should be identical to the original with the exception that the value at the de signated

memory address has been updated to reflect the input value.

Proof. The proof of this theorem is a straight-forward application of the definitions for Read and Write operations.

Theorem 2 (Safety for Read and Write). When a write operation is performed on a RAM with a valid memory address, given that the RAM is initially safe, the resulting RAM will also be safe. Here, safety is defined as in section III, namely that after every read or write operation, every value being stored should be within a specified range.

Proof. For this theorem, an exhaustive proof across operations (namely read and write) is performed. For read operations, the proof is trivial since reads have no effects on the values stored in memory, as shown in the operational semantics. Writes performed, however, do affect the memory store. Let $R = (\mu, \kappa, \sigma)$ be a RAM that satisfies safety with respect to the system parameter $k = 2^M - 1$. The goal is to prove the safety of $R' = write(R, adr, val)$. Consider that R' is safe iff $(val \,\&\, \kappa)$ is in the range $(0..k)$. By definition, $(val \,\&\, \kappa)$ is in the range $(0..\kappa)$. Since R is given to be safe, it follows that $(0..\kappa) \subseteq (0..k)$ and that safety is preserved.

Theorem 3 (Decomposition and Inverse). When memory is decomposed into two partitions, these partitions can each be classified as a RAM by definition. Furthermore, the composition of two RAMs sharing the same mask value into a single memory yields a RAM. Finally, the ordered composition of partitions resulting from decomposition of a RAM results in a RAM that is semantically equivalent to the original.

$$compose(R_0', R_1') = R \leftrightarrow decompose(R, p) = (R_0', R_1')$$

Theorem 4 (Decomposed Read Liveness). If a decomposed read operation is performed on two partitions of RAM, the resulting output is the same as that of the read operation performed on the parent RAM.

$$Let\ R' = decompose(R, p)$$
$$read_{decomp}(R_0', R_1', addr) \equiv read(compose(R_0', R_1'), addr) \equiv read(R, addr)$$

Theorem 5 (Decomposed Write Liveness). When a decomposed write operation is performed on two partitions of a parent RAM, the resulting partitions are equivalent to those resulting from the decomposition of the updated RAM.

$$Let\ R' = decompose(R, p)$$
$$write_{decomp}(R_0', R_1', adr, val) \equiv decompose(write(R, adr, val), p)$$

Proof. Proofs of theorems 3 – 5 follow from a straightforward application of definitions from Figure 13.

Theorem 6 (Decomposition Soundness). If a RAM is decomposed into to partitions, and those partitions satisfy the safety and liveness properties for RAM stated in section III.B, then the original RAM also satisfies these properties.

Proof. For decomposition soundness, each property is proven separately as its own lemma, namely Read Liveness (RL), Write Liveness (WL), and Safety. In the following lemmas, let

$decompose(R, p) = (R'_0, R'_1), p \in \mathbb{Z}_s$.

Lemma 1. $RL(R'_0) \wedge RL(R'_1) \rightarrow RL(R)$.

Proof. Let *R_0^'=(μ_0,k,p).* By definition,

$$RL(R\_0^{'}) \equiv AG\ (addr=i \wedge status=R) \rightarrow AX\ output=\mu\_0\ [i]; i \in Z\_p.$$

Similarly, let

$$R\_1^{'}=(\mu\_1,k,s-p).$$

Again, by definition, this time taking the offset p into account

$$RL(R\_1^{'}) = AG\ (addr=(i-p) \wedge status=R) \rightarrow AX\ output=\mu\_2\ [i-p]; i \in Z\_s / Z\_p\ .$$

From here, it follows

$$RL(R\_0^{'}) \wedge RL(R\_1^{'}) \equiv AG\ (addr=i \wedge status=R) \rightarrow AX\ output=mem[i]; i \in Z\_(p+(s-p)\ ) = Z\_s = RL(R).$$

Lemma 2. $WL(R'_0) \wedge WL(R'_1) \rightarrow WL(R)$.

Lemma 3. $Safety(R'_0) \wedge Safety(R'_1) \rightarrow Safety(R)$.

Proof. For Lemmas 2 and 3, the proof takes a similar form to the proof given in Lemma 1. The underlying property that allows decomposition soundness to hold is the fact that all semantics of RAM can be described in a piecewise fashion and that each property is enforced over these individual pieces.
From these soundness lemmas, it is concluded that decomposition is sound with respect to the properties.

3.2.2 Implications

The utility of such a decompos ition property is an obvious advantage to model checking as it allows one to convert a problem of size $O(N \cdot 2^M)$ into N problems of size $O(2^M)$. Furthermore, because the transition space of such a graph is sparse, the construction of an efficient Binary Decision Diagram (BDD) is easy, further reducing the problem's complexity into something computable on commodity hardware.

In addition to reduced complexity, the division of one problem into N problems is an obvious candidate for parallel computing, thus yielding further computational benefits.

3.3 Formal Verification of Decomposed RAM with NuSMV

In this se ction, the run-time performance for the verification system is described. The runtime analysis begins with using a m odel checker only. NuSMV, an open source symbolic model checker, is chosen as the base line for measuring performance.

3.3.1 NuSMV Model Checking

NuSMV's performance is captured for increasingly large memory size. As defined in the model, M is the word width and N is the number of words being modeled in the RAM. Through experiments, the best performance can be achieved when running with *coi*, *df*, and *dynamic* flags enabled (cone-of-influence, do-not-compute-reachable-set, and dynam ic-variable-reordering respectively). All results here were obtained on a Windows 7 64 -bit PC with an Intel Core i5 -2500 3.3GHz CPU and 8GB of RAM.

A runtime comparison is first looked at by using the initial set of naïve specifications that included some Linear Temporal Logic (LTL) specifications not mentioned here (Figure 14). Sampling ten runs per data point, the importance of efficient specifications is demonstrated and a conclusion is drawn that beyond a memory size of about 12 words, this approach would not complete in a reasonable amount of time (execution was terminated at 3 days for M=16).

3.3.2 Hybrid Verifier

Next runtimes across two verification approaches are compared – first on NuSMV with optimized specifications, and then in the hybrid approach (Figure 15). The performance gain from efficient specifications is obvious, however, state explosionis seen again beyond about 600 bits of RAM. This success of model checking 2^{600} states can be attributed to NuSMV's efficient BDD representation for the model, but reiterate that most modern digital systems have more than 100 bytes of RAM.

The hybrid verifier performed the best over -all. In the graph, the total decomposition runtime is computed as the time required to verify the proof in ACL2 + the time required to run N instances of decomposed RAM in NuSMV. The linear growth is expected to continue well beyond the point where simple model checking fails. Furthermore, it is speculated that parallelization would yield even better runtimes, and it is noted that the re-verification of the machine proof is actually a one-time cost. It is included for completeness.

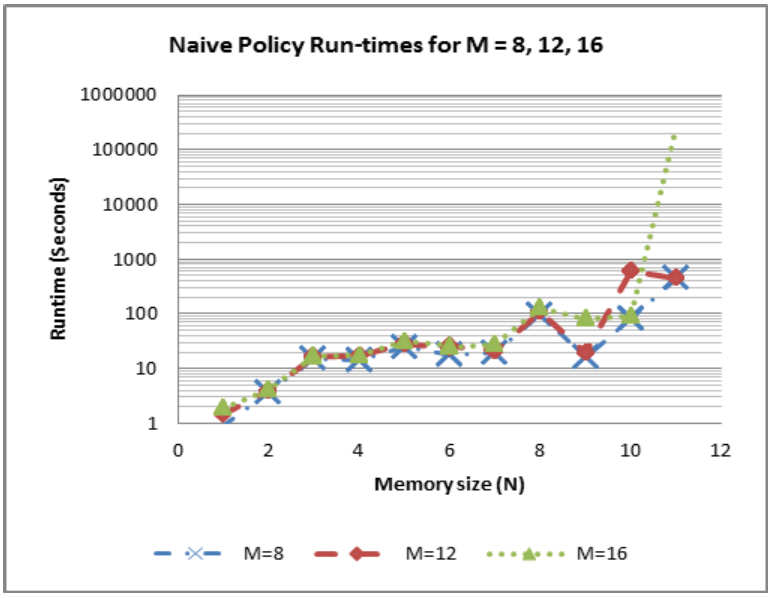

Figure 14. Run time explosion for naïve NuSMV verification.

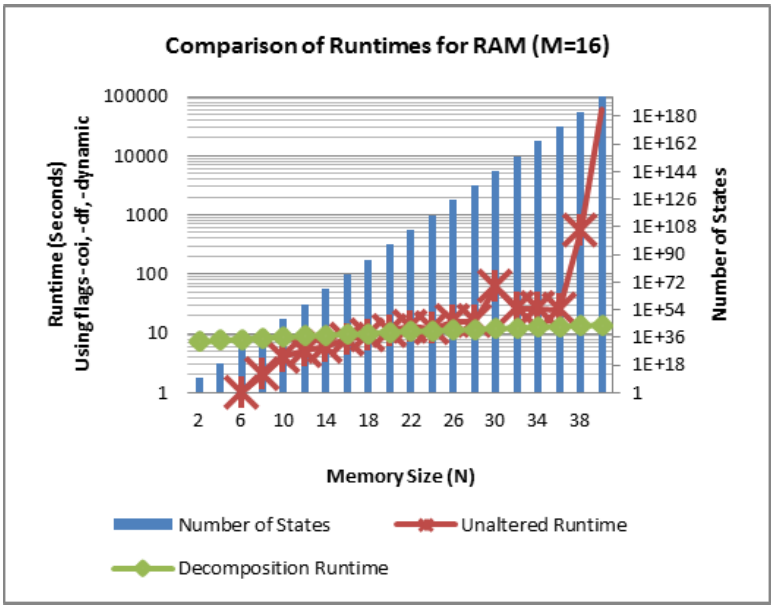

Figure 15. Run time comparison for decomposed and unaltered verification.

3.4 Conclusion

In this section a novel approach to formally verifying the subset of stateful digital systems that exhibits the deco mposition property as defined is presented. When utilized, this property allows a verifier to model check partitions of the system individually, eliminating the unnecessary overhead of checking specifications in states that are inconsequential to the specification's validity and avoiding the state explosion problem.

The emphasis in designing this framework is aimed to maximize automation – a key feature in promoting its use in design verification. Furthermore, because of the integrated use of a theorem prover to validate the decomposition property, the verifier is sound.

This page intentionally left blank.

4. PRACTICAL INTEGRATION OF SIMULATION AND EARLY FORMAL VERIFICATION

Another important component of this work is a practical integration of simulation and early formal verification of embedded systems through library module pairing. Embedded systems are designed for various applications in the NW space, thus the capability of formally verifying these systems are of great value.

An inhibiting factor in formal verification of embedded systems is the time, effort, and expertise required in correctly applying its use. Furthermore, due to the state explosion problem, verification of complex systems requires abstractions that are often not straight forward. Thus, the goal is to integrate formal verification into the current design and test cycle in a way that is most practical for use by system engineers, maximizing automation and ease of use and minimizing redundancy.

This work is built upon Orchestra, a Java based timing-based timing-accurate, event-driven simulator. The first goal is to produce hierarchical model skeleton representations of abstract designs. Then a GUI interface is provided for users to systematically flesh out high level designs into formal modules by "plugging-in" pre-constructed library modules. From here, embedded systems are verified as a whole or in parts using the NuSMV model checker.

The verification system is presented here and a detailed description of the library module macro language is given. The difficulties of bridging the multiple layers of abstraction from high level Java objects to NuSMV modules is discussed. The practicality of the system is demonstrated with an example.

The importance of the early discovery of system design errors is well known. As the complexity of hardware and software designs grow at an accelerating pace, so grows the need for efficient and effective verification techniques. Furthermore, because bugs are much more expensive to fix at later stages of system development, discovering design errors at an early stage is of great interest, both in academia and industry [40].

Traditionally, verification of embedded systems has been accomplished through simulation – high test coverage indicating high assurance of system correctness. However, because successful testing in general does not imply functional correctness, system bugs are still possible [29][32]. In more formal approaches, on the other hand, verification of embedded systems is performed through mathematically sound logic and reasoning techniques (such as theorem proving and model checking). In this case, because verification is performed on the system as a whole, success implies system-wide policy adherence [4].

In hardware designs especially, the use of formal verification has enjoyed increased use for more than a decade [37]. Unfortunately, in software the application of formal verification techniques is difficult, requiring a significant amount of time and expertise. Furthermore, the well-known state explosion problem prevents hands-free verification of moderately complex systems from being feasible. In such cases, abstractions that reduce the overall complexity of a design are invoked – the proper use of which is an open area of research [48].

Much effort has been put into creating embedded system development environments that promote the early detection of bugs through formal verification [41][45][46][47]. In these approaches, a system must be represented at the correct level of abstraction to reduce complexity for the model checker, a problem to which many solutions have been proposed [44].

In one solution, compositional development, functional building blocks are interconnected to achieve a desired behavior. These components can be represented at varying levels of abstraction, promoting rapid development and allowing simulation without unnecessary design complexity [38][39][45]. Additionally, research geared toward verification of models across multiple layers of abstraction has further demonstrated the utility of this approach [38][39][42][43].

The verification system focus on the detection of bugs at the conceptual design phase. The goal is to integrate into a Java based compositional development environment to extract hierarchical system information and allow developers to formally verify behavioral properties through an intuitive library module pairing procedure.

This work differs from prior attempts to formally verify abstract systems in that full unrestricted access to the Java API is allowed when developing and simulating a system design. Furthermore, the pairing technique promotes the separation of formal model creation and system development, allowing current hardware developers to take advantage of the verification tools without expertise in formal verification.

The remainder of this work is arranged as follows. Section 4.1 is an overview of the background components of the verification system – Orchestra and NuSMV. Section 4.2 describes the formal verification system, detailing the design choices and demonstrating the library module macro language. Section 4.3 gives a demonstration of the system's practicality while simultaneously describing the underlying verification process and graphical user interface. Section 4.4 concludes this work with a summary of the work and areas for improvement.

4.1 Background

An overview of the tools that are used is presented before discussing the verification system. The system is integrated into the Orchestra simulation environment and makes extensive use of NuSMV model checker.

4.1.1 Orchestra

In order to minimize the added effort required for designers wishing to integrate formal verification into their system's design cycle, a verification system is built into the Orchestra simulation environment already in use.

4.1.1.1 Description

Orchestra is a timing-accurate, Java-based simulator and design assistant that allows system engineers to simulate their implementations at all stages of the development process, across various levels of abstraction.

The three basic constructs within an Orchestra model are: modules, connections, and ports. Modules are used to represent a block with some specific behavior, connections represent communications conduit between modules, and ports provide an interface from module to a connection.

These basic simulation constructs are non-specific to any level of abstraction. For example, a module can faithfully represent the behavior of a digital logic 'and' gate (VHDL), a class structure (UML), or a high level system such as a stop light at an intersection. This flexibility also extends to connections and ports. As a result, Orchestra can be used to model and simulate interactions of 'components' at varying levels of abstraction all within a single environment.

Also, because Orchestra is based on the Java programming language, developers are encouraged to make use its API libraries where convenient. For example, one may model a memory module using a Java ArrayList as the storage structure rather than logic gates. This effectively allows the developer to abstract away any details that are unnecessary for simulation and saves time in the process.

Once a system has been conceptually mapped out, the specific parts that the developer is responsible for can be migrated down to high levels of fidelity, leaving other parts abstract. A clarifying example is given in the next section.

4.1.1.2 Example Use

Consider that a developer is responsible for delivering a controller module that controls all of the traffic lights at an intersection and accepts commands from a central controller that manages all intersections within the city.

The developer would first model the stop lights at an intersection, the controller, and the central control system as very abstract modules in Orchestra, capturing only the critical aspects of the system (state, data, communication, operation sequences, etc.).

Next, the developer would have the option of creating interactive GUI controls through the Java Swing API either to initiate commands sent from the central control system or to provide a visual representation for the state of a traffic light.

From here, the developer can simulate the model through all required operational scenarios and error conditions to ensure the overall architecture and concept satisfies all system requirements.

Satisfied with the concept, the developer then inserts a processor model in place of the stop light controller module. The original abstract testing infrastructure can now be used to test the actual application software implementation.

4.1.1.3 Advantages and Disadvantages

Because the conceptual infrastructure occurs at such a high level of abstraction, the conceptual phase of model development proceeds relatively fast. Additionally, developers are given the option of providing multiple visual interpretations of the system state by taking advantage of Java's Swing API (e.g. a traffic light graphic or a wave form viewer). Furthermore, since Orchestra is able to operate across levels of abstraction, developers can reuse their conceptual testing environments for later phases of development where certain modules would be exchanged for more detailed versions.

While Orchestra provides for rapid progression through the initial development phases, there is currently no way to translate an Orchestra model one-to-one into an RTL model. This

fact holds true in general for simulation environments and is partially what makes integration of formal verification into the early design phase especially difficult.

4.1.2 NuSMV Model Checker

The NuSMV model checker is employed for verifying the models in order to maximize system automation and provide useful feedback to users wishing to verify their implementations.

4.1.2.1 Description

NuSMV is an open source symbolic model checker based on SMV, a BDD-based model checker [27]. It operates on models as finite state automata and has syntax very similar to Verilog. When provided with specifications written in computation tree logic (CTL) or linear temporal logic (LTL), the NuSMV engine computes whether or not the specifications hold for the model. In the event that a specification does not hold, a counter-example is provided. This is especially useful for developers and allows them to pinpoint errors without additional much effort.

4.2 Orchestra-based Formal Verification System

The over-arching goal though-out this work has been to maximize the feasibility of using formal verification at an early stage of the hardware design cycle. To achieve this goal, the benefits of formal verification must outweigh the time and effort costs required to actually perform the verification. In this section, design choices, the difficulties encountered, and solutions are discussed.

4.2.1 Design

In the design, it is assumed that the user has constructed an Orchestra model and is ready to verify its functionality. No assumption is made about the abstraction level of the model or the system's state of completeness.

The system is implemented as a Java add-on to Orchestra, allowing it full access to Orchestra's module, port, and connection data structures as well as to the details of the model to be verified. When the verification procedure is invoked, a formal NuSMV model is constructed and the user is presented with a verification console. The model has two components: the state variable and module hierarchy, and the logic relations between states.

4.2.1.1 Hierarchy Extraction

The first and easier step is to extract hierarchical information from the model. This is done by performing a depth first search through the collection of modules that make up the Orchestra model and gathering information about how the modules are connected. From this step, information regarding state variables that may reside in each module is also extracted. Because the model can be abstract, the connections may not have a bus width specified, and thus

the state variables remain type-less. In these cases, the state variable to be revisited later is marked.

4.2.1.2 Logic Extraction

The next and harder step is to extract logic level information from the model. Because NuSMV operates at the RTL, equivalent logic level semantics for whatever abstract behavior is being represented must be developed. This problem is hard because the language that the model is written in (Java) exists at a much higher level of abstraction than the HDL-like language on which NuSMV operates. Furthermore, since NuSMV does not have control structures such as loops, it is not guaranteed that the Orchestra source can always be parsed into NuSMV code.

Thus, the first approach would be to simply have the developer fill in the blanks for logic level operations. This was not optimal, however, as it required the developer to waste time writing NuSMV code when they could alternatively just write the Verilog code and verify that. Next, it is attempted to provide a template for the user to specify program logic in the form of Java annotations within the source itself. This, however, would require developers to go back and modify source code to any existing modules that they would want to verify. Also, the annotations themselves required a fairly complex syntax in order to cover the full range of operations within NuSMV, again making the alternative of simply writing the Verilog source sound more appealing.

Finally, the idea of library modules is chosen. By compiling a list of the most commonly used modules and writing the corresponding NuSMV code for them, developers could be allowed to simply pair off these modules with their Orchestra modules and avoid the problem of parsing logic level operations from Java source.

Alternatively, developers could be forced to specify their models at a logic level, enabling the easy extract state transition information; however, the goal is to provide early verification in the embedded systems design cycle, so the ability to handle abstract conceptual designs is important.

4.2.2 Library modules

4.2.2.1 Benefits

The key advantage to maintaining a library of commonly used modules is the trade-off of paying a one-time programming cost to receive a repeated benefit. Developers pay a one-time cost of writing the library module and then are able to reuse that module later in future designs.

Furthermore, in order to increase the amount of reuse, developers are able to use special generic variables when they write their library modules that allow the module to be instantiated in various capacities. These generic variables can represent the bus width of a register or the number of input lines to a module. As an example, consider a multiplexor. In general, a mux is made up of N input lines each having the same bus width M, a control line with width at least $\log_2 N$, and an output also of width M. In a single library module (as will be shown later), all possible instantiations of this multiplexor can be covered, yielding obvious time benefits.

Generic library modules pose added benefits in that they provide a level of abstraction that can be extremely advantageous during verification. This is done by making it easy to abstract large state variables into smaller ones when they are irrelevant to the specification being tested. For example, to test whether or not a communication protocol only operates when certain "enabled" bits are on, it may not be necessary to model the actual communication channel as a 16-bit bus. This leads to improved performance during verification since the complexity of a given model's state space is exponential in the total number of state bits.

4.2.2.2 Syntax

The language for specifying library modules is similar to NuSMV with the addition of generic variables and macros, as well as supplemental input/output type information (Figure 16).

```
                                    LIBRARY MACRO MODULE GRAMMAR
MODULE := "MODULE" + name + params? + "(" + args + ")" + "=>" + outs + "{" + body + "}"
params := "<" + var + ("," var)* + ">"
name := [-a-z_A-Z][-a-z_A-Z0-9]*
args := arg + ("," + arg)*
arg := [-a-z_A-Z][-a-z_A-Z0-9]* + namefunc? + "[" + value + "]"
namefunc := "#(" + expression + ")" //Variable number of inputs/outputs
outs := out + ("," + out)*
out := [a-z][a-z0-9]* + namefunc
body := ("VAR" + text)? + ("ASSIGN" + text)? + ("DEFINE" + text)?
expression := expression "+" factor | expression "-" factor | factor
factor := factor * value | factor / value | lg(expression) | (expression) | value
value := var | num | "[::" + expression + "::]"
var := "%" + [a-z_A-Z][a-z_A-Z0-9]*
num := [0-9]+
text := ([-a-z_A-Z0-9 !@^&*+=()[];:{}<>/] | var | "[**" + textgen + "**]") | "[::" + expression +
"::]")*
textgen := for-func
for-func := "for(" + var + "," + expression + "," + expression + "," + text + ")"
comment := ";;" .* "\n"
```

Figure 16. Syntax for the Library Module Macro Language.

As mentioned above, generic variables allow a developer to cover multiple instantiations of the same module in a single generic module. The supplemental input/output type information will be useful later when it comes time to instantiate the module. Macros are expanded during instantiation and reduce the total number of lines within a module by allowing developers to identify repeated logic structures and replace them with a single macro function.

Macros are especially powerful when combined with generics. Consider again the multiplexor example. In order to correctly instantiate the logic behind an N inputs M bus width mux, a construct is needed that allows user to specify the correct logic for the output line. With macros, this is done easily in a single for-statement as demonstrated below:

```
MODULE sync-mux(in#(%N)[%M],
select[[::lg(%N)::]]) => out[%M]{
  VAR
    out : word[%M];
  ASSIGN
    init(out) := 0b%M_0;
    next(out) := case
      [**for(%i,0,%N, select =
0d[::lg(%N)::]_%i : in%i;
      )**]
    esac;
}
```

4.2.2.3 Instantiation

When a library module is instantiated, first, each type variable listed outside of the body is replaced with the provided instantiation value. Next, expressions inside the "[::" -brackets are evaluated. Finally, name-functions (e.g. variable numbered inputs) and text-gen functions (i.e. text generation macros such as for-func) are expanded. The result is a NuSMV module with the added input/output type information stored for later use. Here, an instantiation of the mux module with $N = 4$ and $M = 16$ is shown:

```
MODULE sync-mux_N4_M16(in0, in1,
in2, in3, select)
              VAR
    out : word[16];
  DEFINE
  ASSIGN
    init(out) := 0b16_0;
    next(out) := case
      select = 0d2_0 : in0;
      select = 0d2_1 : in1;
      select = 0d2_2 : in2;
      select = 0d2_3 : in3;
    esac;
```

4.2.3 GUI

Finally, a graphical user interface (GUI) is provided for making the connections between library modules and their Orchestra counterparts. The GUI was designed to provide developers with complete control over the library modules in a way that is intuitive and easy to use. In Figure 17, a screen capture of the GUI is given and will be referred to repeatedly in the next section as its use is demonstrated.

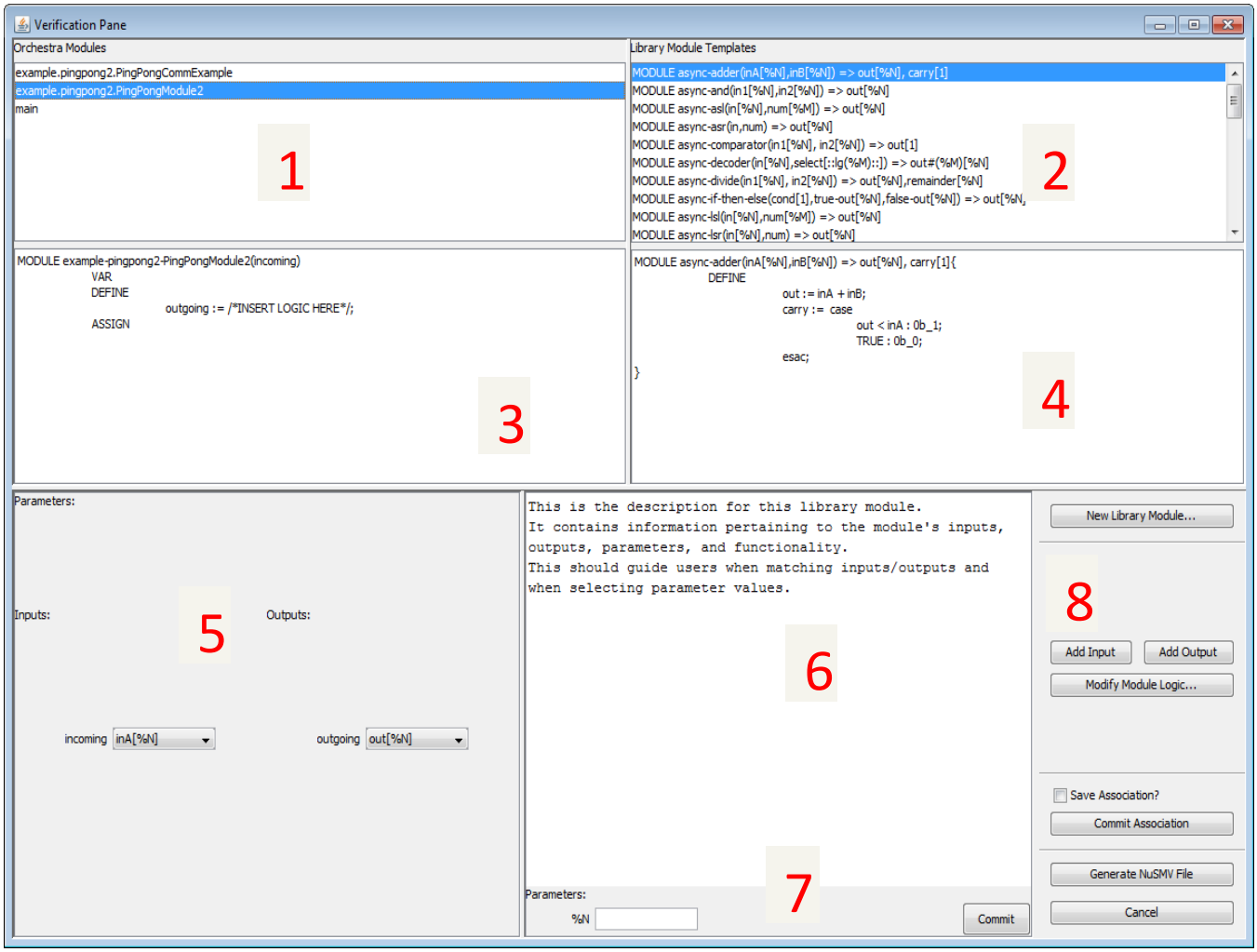

Figure 17. Verification GUI for pairing Orchestra and Library modules.

4.3 Example

To better illustrate the practicality of the system, an example is provided that demonstrates its use. A toy Ping Pong system is considered. In the Ping Pong system, it is

required to create two components that each take one input and one output, and, on each clock cycle, echo's the current input to its output line. The components are synchronized such that the initial values simply bounce back and forth between the components. The module is first presented with a synchronicity error and then illustrate how to correct it.

In the following subsections, Figure 17 and Figure 18 will be extensively used, referring to the labeled areas as they are discussed.

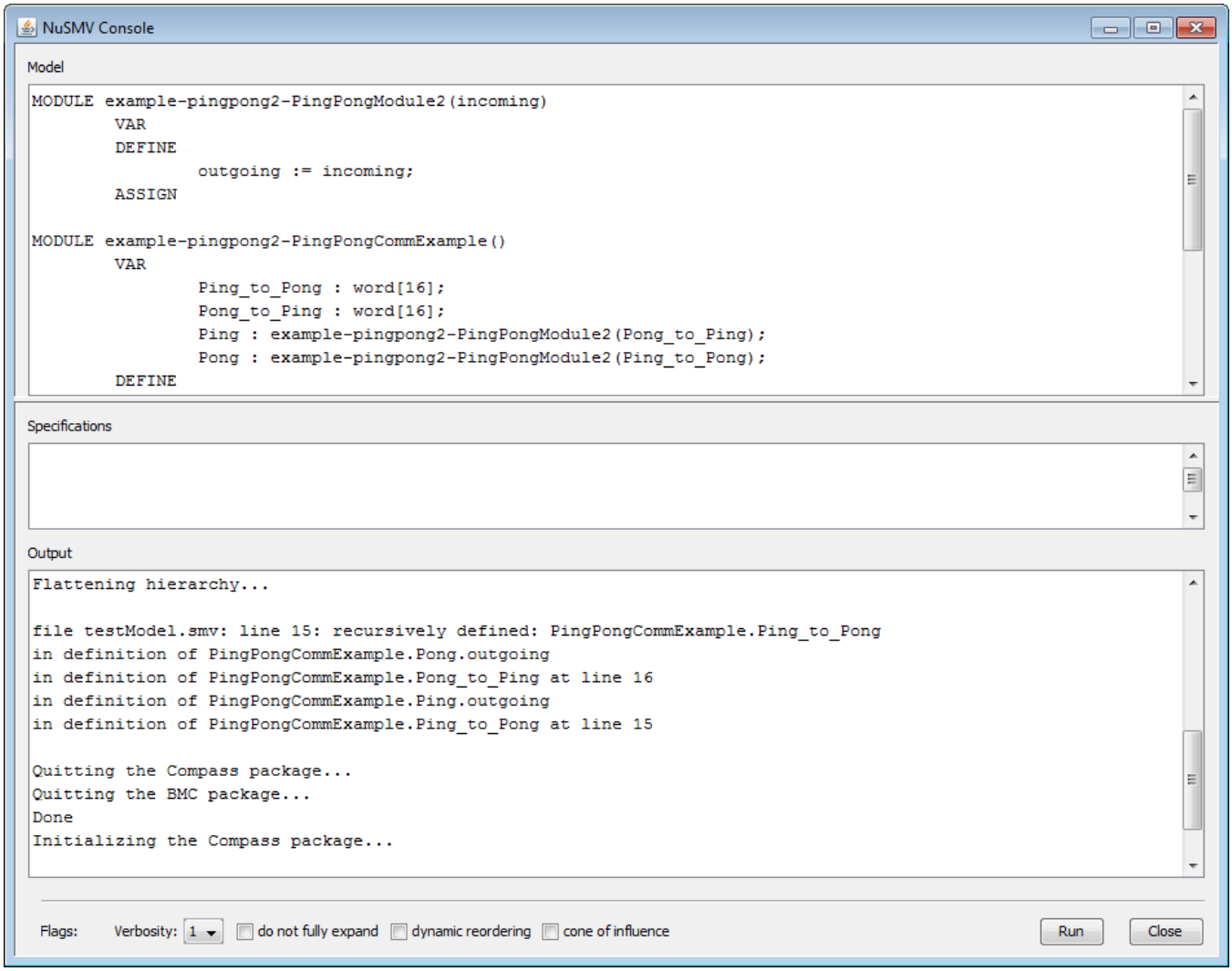

Figure 18. NuSMV Console for performing verification.

4.3.1 Instantiating the Modules

The first step is to pair up each Orchestra module with a corresponding library module. In the GUI, the list of Orchestra modules is presented in area 1. As a developer selects a module from this list, the information in areas 3 and 5 are updated to reflect the new module of interest. In the example, there is only one module of interest, so the PingPongModule is selected from the Orchestra module list.

Once a module has been selected for pairing, the developer selects a corresponding library module from area 2 that reflects the functionality of the Orchestra module. Similarly, when the developer selects a module from the library list, areas 4, 6, and 7 are updated to display information pertaining to the selected module.

If no library module is an exact match for the current Orchestra module, then the user has the option of creating a new library module or modifying an existing one. In either case, the developer will be presented with an editor window to build or make changes to the given module. Once modifications have been completed, the module will be checked for syntax errors and the developer will be given the option to save the new module to the library for future use.

In the Ping Pong example, there is no module that acts as a pass-through, so one is created easily:

```
MODULE
passthroughAsync(in1[%N])=>out1[%
N]{
      DEFINE
            out1 := in1;
}
```

Satisfied with the selected library module, the developer instantiates it. This is done by providing assignments to the each of the parameters in area 7 and hitting commit. Once committed, the library module is locked in focus and area 6 is updated to show the resulting NuSMV code. At this point the developer may make further modifications to the module if needed, this time using NuSMV to verify the syntactical correctness.

4.3.2 Pairing the Modules

Once the library module has been instantiated, the developer pairs the corresponding inputs and outputs using the dropdown menus in area 5. In the example, there is only one input and one output, so the pairings are made for us.

Finally, the developer selects “Commit Association”. This stores the association between the Orchestra module and the library module and replaces the description text of the Orchestra module with the instantiated form. Optionally, the developer can also save the association for future user. This way, at a later date, the developer uses the same Orchestra module, the system will be able to automatically associated it with the correct library module. Of course, at any time, the developer can also un-associate the module, at which point it is reverted to its original state.

The developer continues instantiating and pairing modules until all Orchestra modules requiring logic state transition information have been paired.

4.3.3 Verification

Once all required pairings have been made, the developer selects "Generate NuSMV File." Since in NuSMV, bus width information is contained in the parent node, separate from the child node that performs the logic, this information must be passed up. The system now fills these gaps by passing type information up the module hierarchy from child to parent until all bus widths have been well defined.

Next, the NuSMV console is displayed as shown in Figure 18. Here, the result of translating the full Orchestra model is written out as a NuSMV model and is ready for verification.

The first step of verification is to determine whether the model is valid. To do this, the developer simply selects "Run" while no specifications are shown. In this example, invoking run yields an error stating that variables have been recursively defined. This occurs because in the original extraction from Orchestra, it was not possible to extract information about the synchronicity of the PingPong module due to an error in the model. After correcting the error, the updated module is as follows:

```
MODULE example-pingpong2-
PingPongModule2(incoming)
  VAR
    outgoing : word[16];
  DEFINE
  ASSIGN
    next(outgoing) := incoming;
```

With the delay added between input and output logic, for the rerun, NuSMV responds with "Successful termination."

At this point, the developer is able to test specifications just as they would be able to when verifying a model with NuSMV. For example, it is possible to specify that if Ping_to_Pong is 1 at any given state, then in the next state, Pong_to_Ping should be 1 by the following specification (to which NuSMV will reply "true"):

```
CTLSPEC AG PingPongCommExample.Ping_to_Pong = 0b16_1 -> AX
PingPongCommExample.Pong_to_Ping = 0b16_1;
```

Alternatively, it is possible to test whether at all states, Ping_to_Pong and Pong_to_Ping are equal (to which NuSMV will reply false and give a counter example):

```
CTLSPEC AG PingPongCommExample.Ping_to_Pong =
PingPongCommExample.Pong_to_Ping;
```

For more complex models, NuSMV provides options to drastically increase its performance. Three options are included three (-df, -dynamic, and -coi) to be easily enabled or disabled by the developer.

At this point, the developer is able to correct any problems discovered early in the design phase. Once the model is able to successfully pass all requirement specifications, the developer can then move forward in development with a the strong sense that the system will be completed correctly to specification.

4.4 Conclusion

In this work, a verification system is presented that utilizes module libraries in order to promote the early detection of system design flaws in an unrestricted Java simulation environment. Through the system, the use of formal verification can be promoted while minimizing the time and expertise costs generally associated with it.

5. SUMMARY

During the course of this work, extensive research of Formal Verification (FV) in mission-critical, high-consequence applications was first conducted. Several case studies (e.g. NASA's aircraft controller) that demonstrate the effectiveness of FV have been analyzed.

Within the Sandia domain, this S&T was brought to the attention of several organizations that focus on digital system designs with FPGAs or ASICs. The finding confirmed that Sandia is lacking in an area where such techniques are commonplace in applications for which faults and vulnerabilities are arguably of less consequence. Currently Sandia's approach to system surety is limited to simulation-based verification.

Popular tools have been investigated for formally verifying FPGA-based designs, including open source tools and commercial tools. Each of them has advantages and disadvantages, with none that is tailored for NW specific requirements. By meeting with various organizations within Sandia, several candidate designs have been identified as case studies, from simple block level designs to complex designs such as an Intellectual Property (IP) core.

A decomposition approach was created and implemented to solve the challenging problem of formally verifying RAM, a commonly used digital component in the NW space. Traditional model checking has a significant limitation on the size of the memory due to the known state explosion problem. An novel approach was developed to combine the two major formal verification techniques, model checking and automatic theorem proving. The combined approach successfully solved the RAM verification challenge, by achieving almost constant (instead of exponentially increased) runtime.

In the future, it is possible to include other digital systems in this class of decomposable designs and possibly build a classifier that is able to automatically determine when a state space can be partitioned without compromising soundness. Such an automated system would prove invaluable in promoting the use of formal verification for creating provably secure systems.

Another major contribution of this work is the creation of a practical framework for integrating an existing event-driven simulator (Orchestra) with an advanced symbolic model checker (NuSMV) to meet the vision of Sandia's future digital design methodology. The developed verification system is built upon a library module macro language. The system and library approach was proven with a real design example.

In the future, the system's automation can be improved through guided specification generation and more robust model checking.

This page intentionally left blank.

6. REFERENCES

1. O. Åkerlund, S. Nadjm-Tehrani, and G. Stålmarck "Integration of formal methods into system safety and reliability analysis", Proc. 17th Internatinoal Conference on System Safety, September 1999.
2. S. Browning, M. Carlsson, L. Erkök, J. Matthews, B. Martin, and S. Weaver, "The next wave", in press.
3. T. Chang, A. Danylyzsn, S. Norimatsu, J. Rivera, D. Shepard, A. Lattanze, and J. Tomayko, "Continuous verification in mission critical software development", Proc. Thirtieth Hawaii International Conference on System Science, vol.5, pp 273-284, January 1997, doi: 10.1109/HICCS.1997.663184.
4. E.M. Clarke, O. Grumberg, and D.A. Peled, "Model checking", The MIT Press, 1999.
5. P. Guernic, T. Gautier, M. Borgne, and C. Maire, "Programming real-time applications with SIGNAL", Proc. IEEE, vol. 79, No. 9, pp. 1321-1336, September 1991.
6. J. Hammarberg and S. Nadjm-Tehrani, "Formal verification of falut tolerance in safety-critical reconfigurable modules", International Journal on Software Tools for Technology Transfer (STTT) – Special section on formal methods for industrial critical systems, vol. 7, Issue 3, June 2005.
7. D.S.Hardin, ed. Design and verification of microprocessor systems for high-assurance applications, Springer 2010.
8. K. Havelund, M. Lowry, and J. Penix, "Formal analysis of a space craft controller using SPIN", IEEE Transactions on Software Engineering, vol. 27, Issue 8, August 2001.
9. K. Havelund, M. Lowry,S. Park, C. Pecheur, J. Penix, W. Visser, and J. White, "Formal analysis of the remote agent before and after flight", Proc. 5th NASA Langley Formal Methods Workshop, Williamsburg, VA., June 2000.
10. K.L. Heninger, "Specifying software requiremetns for complex systems: new techniques and their application", IEEE Transactions on Software Engineering, vol. 6, pp. 2-13, 1980.
11. T. Huffmire, B. Brotherton, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Managing security in FPGA-based embedded systems", IEEE Design & Test of Computers, vol. 25, Issue 6, pp 590-598, November-December 2008, doi: 10.1109/MDT.2008.166.
12. P. Jones, R. Jetley, and J. Abraham, "A formal methods-based verification approach to medical device software analysis", Electronic Engineering (EE) Times, 2/9/ 2010.
13. C. Livadas and N. Lynch, "Formal verification of safety-critical hybrid systems", Proc. 1st International Workshop, Hybris Systems: Computation and Control (HSCC'98), vol. 1386, April 1998.
14. J.R. Lewis and B. Martin, "Cryptol: high assurance, retargetable crypto development aand validation", IEEE Military Communications Conference, vol. 2, pp. 820-825, October 2003.
15. W. Ma and X. Hei, "An approach for design and formal verification of safety-critical software", Proc. 2010 International Conference on Computer Application and System Modeling (ICCASM), vol. 4, pp264-268, October 2010, doi: 10.1109/ICCAM.2010.5620084.
16. M. McLean and J. Moore, "FPGA-based single chip cryptographic solution", Military Embedded Systems, Msrch 2007.

17. K.L. McMillan, “Symbolic model checking: an approach to the state explosion problem”, Kluwer Academics, 1993.
18. Mentor Graphics’ Website published success story, http://www.mentor.com/products/fpga/success/northrop-grumman.
19. L. Moser and P.M. Melliar-Smith, “Formal verification of safety-critical systems”, Software—Practice and Experience, vol. 20, issue 8, pp. 799-821, August 1990.
20. NASA Software Safety Guidebook[C], NASA-GB-8719.13, NASA 2004.
21. Z. Qureshi, “Formal modelling and analysis of mission-critical software in military avionics systems”, Proc. 11th Australian Workshop on Safety Related Programmable Systems (SCS’06), Conference in Research and Practice in Information Technology, vol. 69, pp. 67-77,
22. F. Schneider, S. Easterbrook, J. Callahan, and G. Holzmann, “Validating requirements for fault tolerant systems using model checking”, Proc. Third International Conference on Requirements Engineering, 1998, pp. 4-13, doi: 10.1109/ICRE.1998.667803.
23. A. Sutton, “No room for error: creating highly reliable, high-availability FPGA designs”, Synopsys Inc. White Paper, November 2010.
24. P. Taylor, “Using FPGA in mission-critical systems”, Electronic Engineering (EE) Times, 12/6/2010.
25. Y. Yin and B. Liu, “Research on formal verification techniques for aircraft safety-critical software”, Journal of Computers, vol. 5, No. 8, pp1152 -1159, August 2010, doi: 10.4304/jcp.5.8.
26. ACL2 Automated Theorem Proving Tool (http://www.cs.utexas.edu/~moore/acl2/).
27. NuSMV Model Checking Tool (http://nusmv.fbk.eu/)
28. R.E. Bryant, “Formal verification of memory circuits by switch-level simulation”, IEEE Transactions on Computer-Aided Design, vol 10, No. 1, January 1991.
29. Intel Corporation, Statistical Analysis of Floating Point Flaw, FDIV Replacement Program, November 1994.
30. M. Ganai, A. Gupta, and P. Ashar, “Efficient modeling of embedded memories in bounded model checking”, Proceedings of CAV’2004, pp 440-452, 2004.
31. J.L. Lions, Report by the Inquiry Board, Ariane 5 Flight 501 Failure, July 1996.
32. B. McGaughy, S. Wuensche, and KK Hung, “Advanced simulation technology and its application in memory design and verification”, 2005 IEEE International Workshop on Memory Technology, Design, and Testing.
33. K.L. McMillan, “Verification of infinite state systems by compositional model checking”, *Correct Hardware Design and Verification Method,* LNCS 1703, pp 219-233, Springer Verlag 1999.
34. M. Pandey and R.E. Bryant, “Formal verification of memory arrays using symbolic trajectory evaluation”, Proceedings of International Workshop on Memory Technology, Design and Testing, pp 42-49, 1997.
35. N. Shankar, “Combining Theorem Proving and Model Checking through Symbolic Analysis”, CONCUR 2000, LNCS 1877, pp 1-16, Springer Verlag 2000.
36. T. Uribe, “Combinations of model checking and theorem proving”, FroCos 2000, LNAI 1794, pp 151-170, Springer Verlag 2000.

37. Edmund M. Clarke and Jeannette M. Wing. 1996. Formal methods: state of the art and future directions. *ACM Comput. Surv.* 28, 4 (December 1996), 626-643. DOI=10.1145/242223.242257 http://doi.acm.org/10.1145/242223.242257

38. Shangzhu Wang, George S. Avrunin, and Lori A. Clarke. 2008. Plug-and-Play Architectural Design and Verification. In *Architecting Dependable Systems V*, Rogerio Lemos, Felicita Giandomenico, Cristina Gacek, Henry Muccini, and Marlon Vieira (Eds.). Lecture Notes In Computer Science, Vol. 5135. Springer-Verlag, Berlin, Heidelberg 273-297. DOI=10.1007/978-3-540-85571-2_12 http://dx.doi.org/10.1007/978-3-540-85571-2_12

39. Shangzhu Wang, George S. Avrunin, and Lori A. Clarke. 2006. Verification support for plug-and-play architectural design. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis* (ROSATEA '06). ACM, New York, NY, USA, 49-50. DOI=10.1145/1147249.1147255 http://doi.acm.org/10.1145/1147249.1147255

40. Mirko Loghi, Tiziana Margaria, Graziano Pravadelli, and Bernhard Steffen. 2005. Dynamic and formal verification of embedded systems: a comparative survey. *Int. J. Parallel Program.* 33, 6 (December 2005), 585-611. DOI=10.1007/s10766-005-8911-2 http://dx.doi.org/10.1007/s10766-005-8911-2

41. Iskander Kort, Sofiene Tahar, and Paul Curzon. 2001. Hierarchical Verification Using an MDG-HOL Hybrid Tool. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (CHARME '01), Tiziana Margaria and Thomas F. Melham (Eds.). Springer-Verlag, London, UK, 244-258.

42. Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. 2003. Case Studies of Model Checking for Embedded System Designs. In *Proceedings of the Third International Conference on Application of Concurrency to System Design* (ACSD '03). IEEE Computer Society, Washington, DC, USA, 20-.

43. Xi Chen, Fang Chen, H. Hsieh, F. Balarin, and Y. Watanabe. 2002. Formal verification of embedded system designs at multiple levels of abstraction. In *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop* (HLDVT '02). IEEE Computer Society, Washington, DC, USA, 125-.

44. Robert John Allen. 1997. *A Formal Approach to Software Architecture*. Ph.D. Dissertation. Carnegie Mellon Univ., Pittsburgh, PA, USA. AAI9813815.

45. Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. 2003. Metropolis: An Integrated Electronic System Design Environment. *Computer* 36, 4 (April 2003), 45-52. DOI=10.1109/MC.2003.1193228 http://dx.doi.org/10.1109/MC.2003.1193228

46. Marco A. Wehrmeister, Joao G. Packer, and Luis M. Ceron. 2011. Framework to Simulate the Behavior of Embedded Real-Time Systems Specified in UML Models. In *Proceedings of the 2011 Brazilian Symposium on Computing System Engineering* (SBESC '11). IEEE Computer Society, Washington, DC, USA, 1-7. DOI=10.1109/SBESC.2011.47 http://dx.doi.org/10.1109/SBESC.2011.47

47. Popovici, K.; Lalo, M. 2009. Formal model and code verification in Model-Based Design. Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA '09. Joint IEEE North-East Workshop on , vol., no., pp.1-4, June 28 2009-July 1 2009 DOI=10.1109/NEWCAS.2009.5290500

48. Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. ACM Comput. Surv. 41, 4, Article 21 (October 2009), 54 pages. DOI=10.1145/1592434.1592438 http://doi.acm.org/10.1145/1592434.1592438

DISTRIBUTION

1	MS1327	Hart, William	1464
1	MS9001	Mariano, Robert	8005
1	MS9042	Gonzales, Mary	8250
1	MS9102	Forman, Michael	8136
1	MS9102	Hu, Yalin	8136
1	MS9154	Ballard, William	8200
1	MS9158	Armstrong, Robert	8961
1	MS0899	Technical Library	9536 (electronic copy)
1	MS0359	D. Chavez, LDRD Office	1911


Sandia National Laboratories