SANDIA REPORT

SAND2014-20533
Unlimited Release
Printed December 2014

Survey of Existing Tools for Formal Verification

Robert C. Armstrong, Ratish J. Punnoose, Matthew H. Wong, Jackson R. Mayo

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.

Sandia National Laboratories

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 ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2014-20533
Unlimited Release
Printed December 2014

Survey of Existing Tools for Formal Verification

Robert C. Armstrong, Ratish J. Punnoose, Matthew H. Wong, Jackson R. Mayo

Abstract

Formal methods have come into wide use because of their effectiveness in verifying "safety and security" requirements of digital systems; a set of requirements for which testing is mostly ineffective. Formal methods are routinely used in the design and verification of high-consequence digital systems in industry. This report outlines our work in assessing the capabilities of commercial and open source formal tools and the ways in which they can be leveraged in digital design workflows.

Acknowledgment

The authors would like to acknowledge funding from the ASC program to compile this report. The work done under this program has guided the direction of research projects and LDRD proposals.

Contents

List of Figures

Nomenclature

LTL	Linear Temporal Logic
CSP	Communicating Sequential Processes
CTL	Computational Tree Logic
BDD	Binary Decision Diagram
ROBDD	Reduced Order Binary Decision Diagram
SMT	Satisfiability Modulo Theory
SAT	Propositional SATisfiability
FDR	Failure Divergences Refinement
CSP	Communicating Sequential Processes
EDA	Electronic Design Automation
HDL	Hardware Description Language
ASIC	Application Specific Integrated Circuit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
FPGA	Field Programmable Gate Array
RTL	Register Transfer Logic
SVA	SystemVerilog Assertions
PSL	Property Specification Language
ACSL	ANSI C Specification Language
CEGAR	Counterexample-Guided Abstraction Refinement
VDM	Vienna Development Method
UML	Unified Modeling Languagae
JVM	Java Virtual Machine
COTS	Commercial Off The Shelf
ACM	Association for Computing Machinery

Executive Summary

A. Limitations of Traditional Testing and Verification for Digital Systems

The software and hardware engineering techniques used today allow us to create digital systems of great complexity. However, the techniques used for verifying these designs trail quite far behind. Even relatively simple digital systems can have subtle problems that cannot easily be uncovered through simulation or testing. Consider the floating point bug in Intel's Pentium Processor. It was rare enough that it was not uncovered through extensive testing, but not so rare as to prevent its discovery by users one year after release. To test a simple double-precision floating point division exhaustively requires 2^{128} tests. This is not feasible by any measure. Even small digital systems with just hundreds of bytes of memory can have a state space that is impossible to cover with simulation or testing.

B. Formal Methods for Digital Systems

Formal methods are a collection of techniques to analyze a description of a digital system (either native code or a model) as a mathematical object. Formal methods are particularly useful in sifting through the vast combinatorial spaces endemic to digital systems to make quantitative statements about their safety and security properties – usually beyond the reach of testing. Formal tools fall into two broad categories: 1) automated *model checkers*, which apply algorithmic shortcuts to verify desired properties exhaustively over a model's state space; 2) *theorem provers*, which often require human expertise to guide a proof of correctness (these are more powerful and span a larger variety of digital systems). Both off-the-shelf tools and customized tools are used in the design and verification of high-consequence industrial systems.

Commercial off-the-shelf (COTS) tools are designed for ease of application to common design problems. As such, these are exclusively model checkers. Most of them also operate directly on design information provided as source code. While COTS tools are convenient, they are usually targeted to a specific problem domain and verification of properties that are the most common in industry. They do a good job of tackling their problem domain but are necessarily limited in the breadth of problems that they can handle. They are not customizable and cannot be easily adapted to solve problems outside their primary focus area. Most customers of these tools tend to use them to debug rather than formally prove correctness. Thus the tool support is weighted towards this function. Outside their focus area, they are limited. Most industrial practitioners who need absolute verification invest in customizing a tool that is more suited to their product. Open source tools are also available and are leveraged by industry to customize tools to their needs.

This page intentionally left blank.

Chapter 1. Introduction to Formal Methods and Tools

Formal methods, broadly, just means applying mathematics to a problem: in this case, hardware and software computer programs. Because digital systems are themselves just machinery for computing logic, one might think it redundant. The vast combinatorial state space in common digital systems makes their predictability a real problem. A manifestation of a mathematical principle called the Turing Halting Problem, general digital systems are at once deterministic and unpredictable and this is the root cause of the cybersecurity problem so much in the news today. Formal methods, combined with the digital design process, seeks less general digital systems that can be proven to meet some security or safety concern.

1.1. Limitations of Testing and Simulation

Testing and simulation consist of providing a system with a variety of input conditions and ensuring that the output is as predicted. The inputs to be provided may be crafted by a designer or randomly generated. These methods sample the response of the systems to chosen inputs. As such, they can sample functional properties but cannot verify safety and security properties.

Functional properties - Given a nominal input it produces the specified output. It is often assumed that the variety of designed-in behaviors are usually small enough to adequately test for compliance. However consider even a small sequential digital systems with a few digital inputs (A,B,C), and a messaging interface. To be thorough with testing, all possible combinations of A,B,C have to be used. For a sequential system, that is not enough. All possible time-interleavings of A,B,C also have to be tested. For clocked systems, time-interleavings with different delays between the inputs also have to be tested. Also, this would have to be done for every possible message and every possible sequence of messages that could arrive over the interface. Enumerating all of these explodes the input space into a size that is not feasible for testing. In practice, systems are tested with some nominal input and some corner-cases determined by the test designer. This is only a small fraction of possible valid inputs.

Safety and security properties - Colloquially, safety and security properties are about what the digital system "is not supposed to do" rather than functional properties "what it is supposed to do". Given *any* input it must *not* produce a forbidden output. Because these properties are predicated on any path that could bring the system into an unsafe or insecure state, the input state space is vast and almost never accessible by testing. To be certain that safety and security requirements are met we must resort to mathematical analysis of the program, or what is broadly referred to as formal methods.

Given the extensive state-space of even simple digital systems, it is a rare digital system for which "comprehensive" testing is even possible. What is required is a suitably automated mathematical approach that can exhaustively check the entire state space without actually running the code, or can analyze properties of the system analytically. Unlike physical systems, digital systems are just engines for computing logic and are entirely deterministic (under nominal operation). Formal methods are a collection of automated mathematical techniques that exploit this principle to decide propositions about properties of a digital system.

1.2. Introduction to Formal Methods

Broadly, formal methods tools fall into two categories:

Model checkers check the design with respect to the specified properties encoded in a modeling language. A

model checker will attempt to do so automatically with limited human intervention and return one of three results:

1. Properties are satisfied by the design.
2. Properties are not satisfied, for which a counterexample will be given.
3. Indeterminate. The state space is such that the tool cannot compute a result in a reasonable amount of time.

Theorem provers , also called "proof assistants", combine automated techniques with manual guidance to prove correctness. They are generally more powerful than model checkers; developers can use built-in tactics or develop new ones that aid in proving safety and security propositions.

The difference between these two types of tools can be exaggerated. In general, model checkers embody what can be done entirely automatically and theorem provers provide a means for human intervention and creativity. Developers can also devise new tactics for specialized designs or design domains that can subsequently be automated to create a model checker. As we shall see in the next chapter, model checkers are beneficial for developers with little expertise in formal methods who just want a turn-key tool, but may be limited in what they can prove. Theorem provers are more capable but require more expertise and can become tedious for repetitive tasks. The Rodin tool uniquely combines both theorem provers and model checkers in the same package (see Section 5.2).

1.3. Levels of Abstraction in Formal Methods Tools

Just as different computer languages are targeted at different levels of abstraction (e.g., assembler at a low level, Java at a high level), so too are formal tools designed for use at various levels of abstraction. High level formal tools usually are used for proving a specification or system level model and concerned with requirements such as deadlock free behavior or race conditions. Low-level formal tools are used for proving RTL code and concerned with reachability correctness.

Not surprisingly, model checking tools that purely deal with Boolean logic, asynchronous interference, and integer mathematics (this covers all of them) have drawbacks. They fail to take advantage of higher level abstractions that are commonly used by developers and designers. For example an often used idea of a counter will not *a priori* be recognized by a low-level model checker. As an example, Listing 1.1 shows an easily recognizable counter written in NuSMV's modeling language (used to describe hardware designs). The final line in the program asserts a requirement

```
1 MODULE main
2 VAR
3 i0 : 0..1000000;
4 ASSIGN
5 init(i0) := 0;
6 next(i0) := (i0 + 1) mod 1000001;
7 LTLSPEC F (i0 = 999999);
```

Listing 1.1: A simple counter in the NuSMV modeling language

that `i0` takes on the value 999999 at some point in the computation. The human eye can easily discern that the counter will count up to 1000000 and the variable `i0` will at some point take on the value 999999. Not so for the model checker: it has to run the state space all the way out to 999999 before it discovers that the assertion in the last line is true. The inability to infer a higher level property about a system is common among model checkers. The human eye in this case is taking advantage of *induction*, which the quick solution to this problem requires, and model checkers lack this facility. Human guided theorem provers, however, commonly rely on induction, proving systems that would otherwise be intractable.

1.4. Classification of Formal Tools

For the purposes of this report, we classify the available tools by their capability and usage scenario. Since there has been steady academic research in this area, many formal tools have resulted. But only a few of them have been maintained over time and seen popular use. Our survey of tools in this area is not exhaustive but captures the major ones and gives a flavor for their variety. We divide these tools into two classes: a) tools to verify correctness of a constructed model and b) tools to create a model or design that is correct by construction.

1.4.1. Tools to Verify Correctness of a Model

The tools in this category are useful for verifying the correctness of a model/design after it is created. They do not aid in the creation of the model/design directly, but can be used to test out ideas before implementation.

1.4.1.1. Tools That Verify Abstract Models

This class of tools requires the creation of an abstract model of a system in a modeling language. In all of the existing such tools, the modeling language is tool-specific, is targeted at a specific type of problem, and captures the semantics appropriate for the problem. In practice these tools might be applied to a design specification to ensure that it is self-consistent. Abstract formal tools are also useful to verify system-level behavior that includes digital systems but also physical systems together as an integrated whole. The modeling languages used by these tools have the following properties:

- They have well-defined semantics. This is essential for formal verification. This requirement also disqualifies many practical design languages from being used for formal verification.
- These languages are concise and often lack features to elaborate design detail.
- They may have features unique to verification (e.g., mechanism to specify formal properties, model in-determinism)

Correspondence of the abstract model to the implementation is always a concern with these tools. In many of these cases, the goal of these tools is to capture just portions of a system that are hard to reason about. It is not usually the goal of these tools to capture an entire system design in depth. Some of these tools are listed below.

1. Spin : Used to model concurrent software or asynchronous processes.
2. Uppaal : Used to model real-time systems.
3. SMV, NuSMV : Used to model synchronous digital logic.
4. FDR : Used to model asynchronous systems.
5. Alloy : Used to analyze consistency of software data structures.
6. Simulink Design Verifier : Used to verify models created in Simulink, a data-flow and state-machine simulation tool.

These tools are described in Chapter 2.

1.4.1.2. Tools That Verify Actual Design Descriptions

Tools that operate on existing designs take as input either the RTL description of a hardware circuit or source code for software. Generic programming languages not intended for mathematical analysis are usually not conducive to formal techniques and limited in what behaviors can be decided.

A. Software Verification Tools

There are large amounts of deployed software-based systems. And even the best scrutinized of these systems is not bug-free. There is great academic and commercial interest in being able to formally check software. However, almost all programming languages in use are Turing complete and it is very difficult to check software. It is very much the case that the tools in this category perform rather shallow checks, i.e., can only check for classes of simple problems. These tools therefore are limited in that they can only handle small designs and can only check rather simple properties.

1. Frama-C
2. BLAST
3. Java Pathfinder
4. Spark ADA
5. Malpas

These tools are described in Chapter 4.

B. Hardware Description Language (HDL) Verification Tools

There is a rich history of formal methods in hardware design. This is not accidental. Several factors have contributed to this:

The hardware design process is very expensive. Portions of the hardware design workflow require specialized skills. Hardware designers initially create a model of the hardware in a description language like VHDL or Verilog. The design is then synthesized. The synthesized design then goes through a layout process. Critical portions of the design are laid out manually. Thus if there is a flaw in the original design, the rework is very expensive. There is a commercial incentive to catch flaws as early as possible in the design process.

Changes are difficult once deployed. Unlike software, fixing hardware designs after release is usually not possible. Often this leads to recalls at great cost.

Analysis is feasible due to simplicity of the underlying representation. Hardware description languages can be used to create fairly complex systems. For example, a processor in hardware that runs instructions from memory is effectively a software system that defies serious analysis. However in practice, within a single ASIC or FPGA design, the hardware design can be represented as clocked sequential logic. Hardware designers eschew asynchronous designs as they are difficult to analyze. Effectively, most hardware designs resemble state machines and memory. These are the easiest types of digital systems to formally analyze.

The practicality and demand for checking hardware designs has given rise to focused academic research in this area. This has yielded techniques and tools such as SAT solvers, BDDs, ROBDDs, and SMT solvers that are now broadly used in other domains.

The safety and security properties that can be verified from the RTL design are limited to those that can be expressed as boolean expressions. Safety and security requirements that are expressed in higher level semantics are beyond the reach of these tools. These tools are most valuable in checking existing designs since no more input is necessary than the code itself.

The commercial demand for these types of tools keeps their price quite high. Some of these tools currently on the market are:

1. Questa Formal by Mentor Graphics
2. Solidify by Averant
3. JasperGold
4. Incisive by Cadence

These tools are described in Chapter 3.

1.4.2. Tools to Create a Provably Correct Design

The tools in this class provide a formal framework and methodology to mathematically model and prove properties about a system. These are all based on set theory. These tools can tackle a large set of problems, but to be useful, they have to be adapted for the domain of the problem. This step requires expertise. The adaptation is done by narrowing the semantics to deal specifically with the problems of interest.

1. VDM
2. Z, B, Event-B, and Rodin

These tools are described in Chapter 5.

1.4.3. Tool Comparisons

Most of the tools described so far are for "after the fact" verification. A program is created and afterwards verified for requirements. Safety and security requirements, given the vast state space over which they must hold, are difficult to preserve in the traditional digital design process without some scaffolding to ensure it.

Model checkers are generally used for this "after the fact" verification. Model checkers are generally easier to use without much additional expertise, but when some requirement is "proven" the practitioner has to take the model checker's word for it. There is no independent way of verifying that the proof is valid.

The tools in Section 1.4.2 usually include a theorem prover that enables the practitioner to adhere to safety/security requirements at every step of the design process and, in this way, produce a design that is formally correct by construction. Theorem provers have the disadvantage that they require greater expertise to use.

Some theorem provers are capable of producing an independently checkable "proof term". Checking the proof term does not need to rely on the theorem prover software to be verified. Generating a proof term as part of the design process is directly analogous to the traditional testing of digital systems for function. In fact, testing can be thought of a degenerate case of generating a proof term. A test is comparing input/output pairs to the original specification, and if they match, the proposition that the test poses: "do we get the right output for this specific input" is established as true. A proof term might establish a more complex proposition like: "for *all* possible input this particular output never occurs", but it is pretty much the same idea.

This page intentionally left blank.

Chapter 2. Tools for Checking Abstract Models

To perform formal verification of a design, a formal description (model) of the design is required. Practical design languages are not geared towards formal verification. Particularly, they are abundant with undefined behavior, unspecified behavior, and ambiguous semantics. Most of these languages are defined with efficiency and usability as primary goals and lack formal semantic definitions.

Many formal tools work with abstract models created in custom languages that are solely created to be amenable to formal verification. They are targeted towards specific application domains and their modeling languages reflect these biases. These tools are used to reason about high-level properties of a system that abstract out implementation detail. They are also limited in the size of the model that can be reasoned about. In this chapter, we describe the better known and used tools that fall in this category.

2.1. Spin

Spin is a model checker [1, 2]. It was first developed in 1980 at Bell Labs, to verify call processing on telephone switches. Since 1991, it has been available publicly as open-source. Today it is actively developed and maintained by NASA/JPL. It was awarded the 2001 ACM Software System Award.

Spin is primarily targeted at the formal verification of software algorithms, specifically parallel, multi-threaded algorithms. As such, it is good for modeling asynchronous systems of interconnected components.

Spin models are written in Promela (Process or Protocol Meta Language). This is a simple language with "C" like syntax. It is small, unambiguous, and has features for verification. Until recently (Spin 6.0, 2010), Promela did not have a mechanism for modularizing a program into procedures or functions. That this was not severely limiting, is indicative of the size of most Promela models that were analyzed by Spin. Most Promela models are compact but express high-level properties that are difficult to reason about. Some examples are mutual exclusion, semaphores, lock-less data structures, cache coherency etc. Spin models are suited to analysis of two types of properties:

- Detailed properties of small models
- high-level properties of large systems.

Spin is an explicit-state model-checker. As such, Promela supports indeterminate behavior. Spin/Promela supports both simple assertions and complex linear temporal logic properties. Using these constructs, properties such as Reachability, Safety, Liveliness, and Deadlock Freedom can be specified and verified.

Spin is extremely useful for analyzing concurrent systems. Edsger Dikstra created one of the first attempted solutions to the mutual exclusion problem without using atomic sequences. In the decade following that, there were numerous presentations of simpler algorithms. Most were subsequently shown to be incorrect. It took fifteen years before Peterson's deceptively simple algorithm (shown in Listing 2.1) was discovered. The point is that devising and reasoning about a mutual exclusion algorithm or synchronization primitive is very difficult. However model-checkers like Spin make this trivial today. It is also easy to verify that making a small change on line 14 to

```
(flag[j]== false);
```

will break this algorithm, something that is not obvious by inspection.

```
bool turn, flag[2];
byte cnt;

active [2] proctype P1()
{
        pid i, j;
        i = _pid;
        j = 1 - _pid;

again:
        flag[i] = true;
        turn = i;

        (flag[j] == false || turn != i) -> /* wait until true */

        critical_section++;
        assert(critical_section == 1);  /* Only one process in critical section */
        critical_section--;

        flag[i] = false;
        goto again;
}
```

Listing 2.1: Peterson's Mutual Exclusion Algorithm in Spin/Promela

Spin is a command-line analysis tool but is accompanied by a graphical interface, ISpin (Figure 2.1).

2.2. Uppaal

Uppaal [3] is a tool for model-checking timed automated. It incorporates the notion of time. The functionality it provides is similar to that of Spin, with the exception of its treatment of time. It has a proprietary language with an interactive development environment in which models can be described and properties specified (Figure 2.2). It started out as an academic tool and has since then been commercialized.

2.3. SMV, NuSMV

NuSMV [4, 5] is to hardware design what Spin is to software. SMV [6], the predecessor to NuSMV, was developed in 1993 as a software tool for the formal verification of temporal properties of finite state systems. It uses a custom input language that allows description of synchronous and asynchronous systems. The language and its semantics were designed to model hardware logic design, but could be used for other domains as well. NuSMV and NuSMV2 extended the original SMV to add checks for both LTL and CTL properties. Underneath the hood, NuSMV uses Binary Decision Diagrams (BDDs), and propositional satisfiability (SAT). The current version of the tools do not support asynchronous systems.

2.4. FDR

Communicating Sequential Processes (CSP) is a process calculus for concurrent systems [7]. CSP has influenced many programming languages and even model-checking tools such as Spin.

Failure-Divergences Refinement (FDR and FDR2) is tool to check models expressed in the algebra of CSP [8]. FDR was developed at Oxford University and commercialized by FormalSystems (http://www.fsel.com/).

In the CSP/FDR world, a system is modeled as a set of processes connected by synchronization events. The composition of these processes using a set of standard operators provides a hierarchical description of the system. There is a hiding operator that provides an abstraction mechanism.

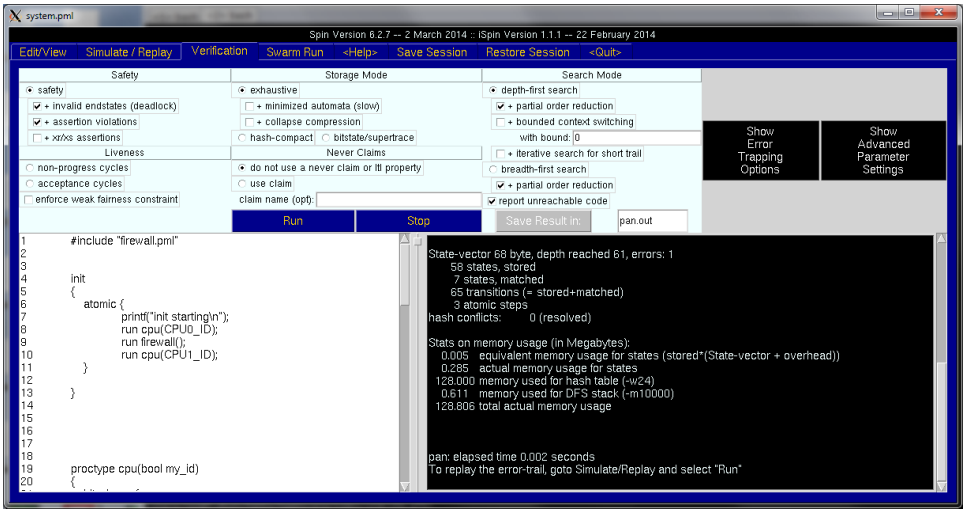

Figure 2.1. The ISpin graphical interface for Spin analysis

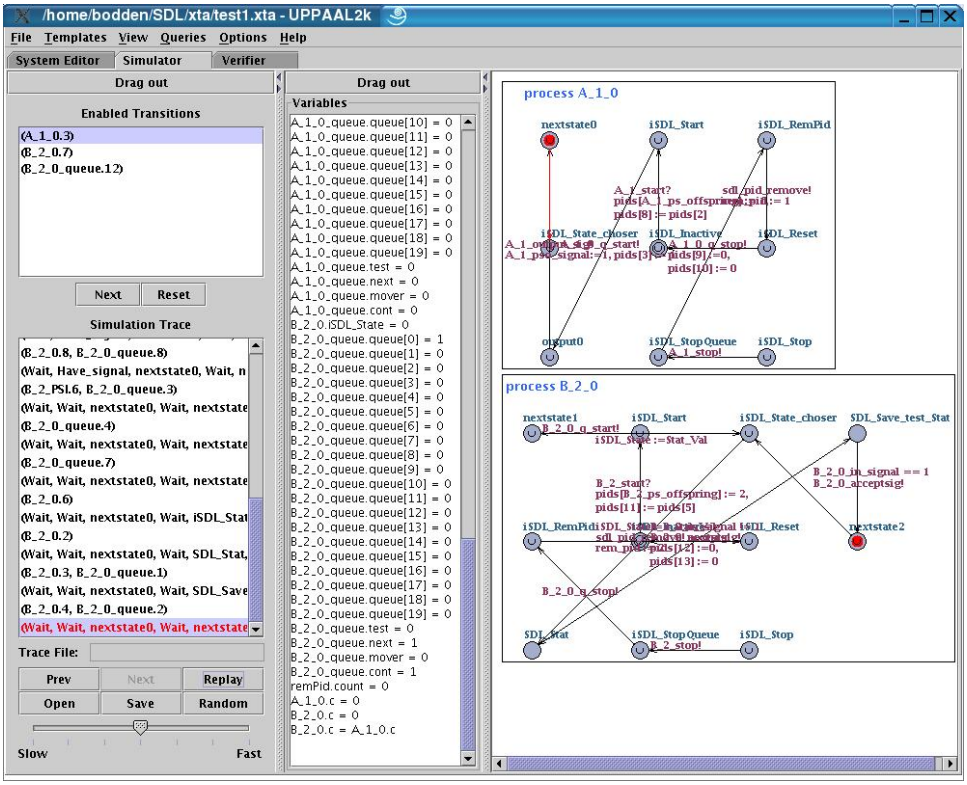

Figure 2.2. Uppaal environment for model-checking

```
MODULE big_state_machine(input1, input2)
VAR
  output : boolean;
    state : {idle, state1, state2, state3};
ASSIGN
  init(output) := FALSE;
  init(state) := idle;

  next(state) := case
                   state = idle   & input1 = TRUE :  state1;
                   state = state1 & input1 = FALSE & input2 = TRUE : {state2, state3};
                   state = state2 & input2 = FALSE : idle;
                 esac;
DEFINE
  output := case
              state = idle : FALSE;
              state = state1 : TRUE ;
              state = state2 : FALSE;
              state = state3 & input1 : FALSE;
              state = state3 & input2 : TRUE;
              TRUE : TRUE;
            esac;

MODULE small_machine(input1)
VAR
  output : boolean;
  state : {reset, start, stop}
ASSIGN:
  init(state) := reset;
  init(output) := FALSE;

  next(state) := case
                   state = reset & input1 = TRUE : start;
                   state = start : {start, idle};
                   state = stop & input1 = FALSE : idle;
                 esac;
DEFINE
  output := case
              state = start : TRUE;
              state = stop : FALSE;
              esac;

MODULE main
VAR
  sm1 : big_state_machine(sm2.output, sm3.output);
  sm2 : big_state_machine(sm1.output, sm4.output);
  sm3 : small_state_machine(sm1.output, sm4.output);
  sm4 : small_state_machine(sm2.output, sm3.output);
LTLSPEC
  -- Safety Property specification
  G ! (sm1.state = state2 & sm4.state = stop);
```

Listing 2.2: NuSMV model of a safety verification for interacting state machines

FDR takes as input a model of the system and the property to be checked, both expressed in CSP. It determines if the system model is a refinement of the property for a given semantics. A successful check shows that the system meets the property. FDR is an explicit-state model checker.

2.5. Alloy

Alloy is a an object and structure modeling language based on set-theory [9]. The Alloy tool takes a structure model in the Alloy language and specifications. It uses first-order logic to translate specifications into Boolean expressions and analyzes them by connecting to existing SAT solvers. Alloy uses “lightweight formal methods”. Alloy is used for analyzing the consistency of software data structures such as linked-lists, hash-tables, and for analyzing the set relationships between data types in databases, etc. It is thus primarily used for structural analysis of data. It cannot analyze temporal properties.

Alloy Analyzer (Figure 2.3) is an analysis tool that takes the model properties and finds solutions to an abstract

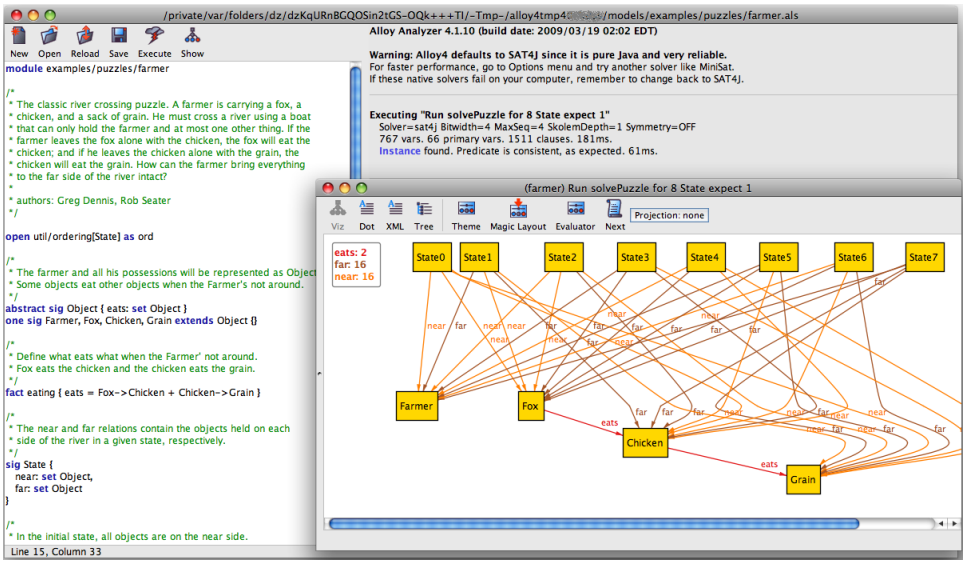

Figure 2.3. Alloy Analyzer

description of the model.

2.6. Simulink Design Verifier

The Simulink Design Verifier by Mathworks is a tool for the verification of formal properties of a Simulink design [10]. Simulink is a graphical language for the design and simulation of digital designs with control logic and signal processing. Simulink represents designs in two forms: data-flow diagrams and state machines. Both of these constructs restrict the designs to forms which in theory are amenable to formal analysis. Unlike the other tools mentioned in this chapter, Simulink is primarily a design and simulation tool widely used by engineers, not a formal verification tool. It is also possible to auto-generate an implementation from the Simulink design in terms of "C" language code or "VHDL/Verilog" for hardware instantiation. In this regard, Simulink is more than an abstract modeling tool but can be an implementation tool. However, at present, most users use it primarily for design and simulation.

Simulink Design Verifier is a toolbox for Simulink that allows engineers to incorporate formal verification into their normal design process. It internally uses technology from PROVER (http://www.prover.com/) to perform this analysis. The Simulink Design Verifier is however limited to assertions and a small set of time operators with fixed delay. That is, currently it only allows specification of a subset of LTL properties.

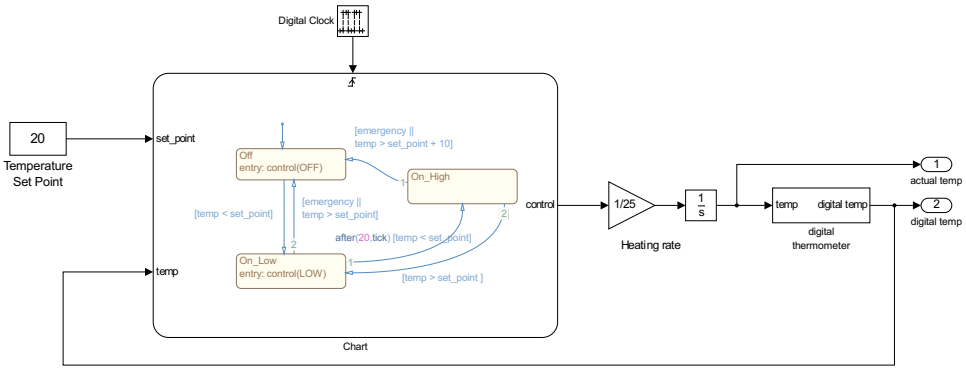

Figure 2.4. Simulink design tool

Chapter 3. Tools for Checking Hardware Description Languages

There are several tools today for checking hardware descriptions from the direct RTL description provided as Verilog or VHDL design files. This is a distinct advantage in that a separate model of the design does not have to be created. An existing design can be analyzed in this fashion. In a typical usage, a set of properties is written against a synthesizable design. "Synthesizable" is key since the full Verilog or VHDL standards have features that are only used for simulation. With these features, the languages become Turing complete and cannot easily be checked. The "synthesizable" subset of these languages used in practice can be fully described using Boolean logic and registers and thus is amenable to analysis. The hardware design is analyzed in conjunction with the specified properties.

3.1. Overview

3.1.1. Property Specification

Properties are specified either in the Property Specification Language (PSL) or as SystemVerilog Assertions (SVA) [11, 12]. PSL is language independent. PSL assertions are written in a stand-alone file. SVA, though it is part of the SystemVerilog standard, supports designs written in Verilog, SystemVerilog, and VHDL. SVA assertions can be written stand-alone or inline within a SystemVerilog based design.

Both languages can specify concurrent assertions and temporal logic assertions. The temporal assertions cover LTL (Linear Temporal Logic). PSL additionally supports properties written using branching logic (CTL). Properties can also contain modeling code. This allows the creation of quite complex properties.

Many of the tools also have add-ons to analyze common hardware design problems. These add-ons make it easier to generate properties for standard design components. For example, the ARM peripheral bus is often used in many hardware designs and the properties for such a bus are specified by the ARM bus standard. The add-ons provide a way to either use or generate standard properties with minimal user interaction.

3.1.2. Internal Operations

These tools primarily use model checking. Internally, after ingesting the source files, these tools synthesize the hardware description into simple Boolean logic and registers. They also synthesize the properties along with the design. The tools then use a variety of methods to analyze the properties. These methods can be applied in parallel. The tools have multiple analysis "engines" that can be applied in parallel on a given problem. All of these tools have a scripting interface that allows an external program to control the operation of the tool. Through this interface, the tools support changing the parameters of the engines, the strategy to apply to a given property, etc.

3.1.3. Tool Outputs

Property failure When a property fails, these tools provide a counterexample which clearly demonstrates the failure. This counterexample is expressed through visualizations of signal waveforms and linking to design HDL code.

Property pass However, when a property succeeds, they (like all model checkers) lack verbosity in reporting successfully verified assertions. The tool output is simply a "pass" message on the screen; proof details are not provided.

Indeterminate If the state space of the design or property to be checked is large, then the tools may not be able to complete an analysis.

3.1.4. Scalability

The complexity of a design is not just in the amount of gates but also in the number of memory elements in the design and the inter-connectivity between the components. Even a design with a small gate count but about 100 memory elements can result in an exceedingly large state space that is hard to analyze.

Design features such as memory and counters are hurdles to analysis. Even a simple long-running counter is difficult to analyze. We have seen that the existing tools simulate the entire operation of the counter rather than inferring its properties.

Some of the tools claim to abstract out counters. What is done here is that the output of an abstracted counter is allowed to hold any legal count value at any time. This can result in "maybe" violations, i.e., the tool flags something as a potential violation when it is not a problem.

3.1.5. Cost

The available tools today for this purpose are all commercial tools. They are also quite expensive. The business model of these tools is a yearly license and the vendors expect to consult with the designers to use these tools directly. Depending upon the licensing model of the vendor, scaling the analysis to multiple machines would be cost prohibitive.

The vendors are moving away from one-time perpetual licenses to a yearly license. A yearly license can cost as much as $400K.

3.2. Questa Formal by Mentor Graphics

Mentor Graphics is a large vendor of EDA software. Mentor has a large tool-suite and Questa Formal is one of its offerings. Questa Formal originated as a formal verification tool offered by 0-In Design Automation, which was acquired by Mentor Graphics. Mentor Graphics also acquired Axiom Design Automation, which used to produce an easy to use formal verification tool. Questa Formal offers a basic but established commercial tool, lacking some of the advanced formal methods research features that are contained in competing tools, such as abstraction of counters and advanced liveness features. The user interface for Questa Formal is easy to learn and understand. Questa Formal has an advantage in that it integrates well into the other tools provided by Mentor Graphics. This is important since EDA software is a significant investment. For example, when a counterexample is generated, Questa Formal can provide it in the form of a testbench that can directly be used by Questa Sim, which is the simulation tool provided by Mentor Graphics.

The tool works reasonably well for the most part. It appears to have on roughly seven analysis strategies or "engines".

3.3. Solidify by Averant

Solidify has been on the market longer than other tools. However, at present, Solidify's market share seems to have dipped substantially. Averant, located in Berkeley, is a very small company and Solidify is its only product. Solidify is more economical than the other tools, but lags them in features.

Solidify does not support abstraction of counters. It also does not support liveliness properties. The tool appears to have two analysis "engines".

During our evaluation of this tool, we ran across several critical bugs with this tool. The user interface is lacking, and there were times when the verification process would continue seemingly forever, even though we had set effort and time limitations, which were ignored.

Solidify also has an incomplete support for the VHDL language, and for some parts of the SystemVerilog Assertion language.

3.4. JasperGold

Jasper Design Automation is a commercial software company that only produces formal technology products. It sells JasperGold and several applications based on JasperGold. Out of the tools that we evaluated, this seems to be more capable than the others. Particularly, their user-interface is amenable to iterative analysis (performing a verification, making property changes and iterating on the verification).

They appear to have more analysis "engines" than the others. Each "engine" is good at analyzing certain types of properties. Jasper also has an add-on to verify properties about data-dependency. This can be useful for security verification, as these properties cannot easily be expressed using SVA or PSL.

There is automatic extraction of counters for abstraction, in addition to support for manual counter extraction. This feature would speed up the running time of verification of properties that rely on counters. However, as described before, counter abstraction has to be used with care in the knowledge of how it is done to avoid being fooled by false positives.

As of 2014, JasperGold appears to be one of the leading tools in this area by features, performance, and usage in industry. In June 2014, Jasper Design Automation was acquired by Cadence.

3.5. Incisive by Cadence

Cadence Design Systems is another of the large EDA vendors. Over time, Cadence has acquired multiple formal-tool vendors (Bell Labs Design Automation and Verplex). Incisive, the current formal verification offering by Cadence, has capabilities similar to Questa Formal and integrates well with the rest of the tool suite from Cadence.

In June 2014, Cadence acquired Jasper Design Automation. It remains to be seen if Incisive and JasperGold will be sold independently or merged into a single product.

This page intentionally left blank.

Chapter 4. Tools for Checking the Correctness of Software

Formal verification of a system requires a formal model of the system. To ensure that there are no discrepancies between the model being verified and the implementation, it is desirable if possible to analyze the design directly. In the case of software implementations, this would imply a formal analysis of the source code. This is a difficult task because software languages in use are not designed for verification. Most of them even lack a specification. However, due to the abundance of software based systems and the propensity for finding bugs in them, there is a great need for using formal verification with software.

There exists a limited set of tools today to help in this regard. They are limited in that they only permit a subset of the language features to be used so that the formal verification is tractable. Also, the properties that can be specified are restricted.

4.1. Frama-C

Frama-C is a collection of tools for the static analysis of C source code. Unlike many commercial static analysis tools for bug-finding, Frama-C allows the user to specify complex functional specifications and to prove the correctness of the source code with respect to the specifications. Functional specifications are written in a dedicated language called ANSI/ISO C Specification Language (ACSL). ACSL is a formal language. It can specify simple facts (such as the type of a function parameter) or complex ones (e.g., the input to a function is a non-empty linked list of integers and return value is the maximum value of the contents).

4.2. BLAST

BLAST is an automatic verification tool for checking temporal properties of C programs [13]. It is tuned towards checking safety properties (existence of paths to undesired states). BLAST takes as input a C program. The properties that it can check for are:

- Reachability: Is a particular program location (specified by a C label) reachable?
- Standard C logical assertions.
- Temporal properties specified by sequences of allowable behavior described in a custom property file.

BLAST will produce counterexamples for failed properties.

BLAST is fairly advanced among model checkers for software. It is also somewhat unique in this category in that it uses a sophisticated mix of techniques to perform the property check. It uses abstractions and counterexample-guided abstraction refinement (CEGAR) to make the checking efficient. It also employs theorem provers for constructing abstract state transitions and for checking the feasibility of error paths [14].

4.3. Java Pathfinder

Java PathFinder started out as an explicit-state model checker for Java bytecode [15]. It can thus perform model checking on programs written in languages that can target the JVM. Its initial implementation approach was to convert Java bytecode to Promela to be analyzed with the Spin model checker [16]. But since then, it checks Java byte code directly.

In its current form, it also allows many plugins to customize the properties and checks. In its bare form, Java PathFinder searches for deadlocks, assertion errors, and null pointer exceptions.

Due to practical state storage restrictions, the size of programs that can be analyzed by the tool is limited (to approximately 10K lines of code). Due to these limitations, it is not typically used for analyzing whole programs. Java PathFinder is useful for the following types of analysis:

- Analyzing the portions of a program that deal with concurrency.
- Analyzing an abstracted model of a program.

The plugin architecture allows a user to extends the type of properties that can be checked. These are implemented as custom code with a listener model to check for custom properties as the state space is explored. However, the state space exploration algorithm is blind to the type of custom properties and cannot customize the search to be efficient for those properties.

4.4. Spark ADA

SPARK is a formally defined subset of the ADA programming language [17]. It was motivated by the need for developing software for high reliability and safety critical applications. The SPARK language consists of a well-defined subset of the ADA language and property specifications. The properties are specified as program annotations, inline with the source code and embedded as ADA comments. As a subset of ADA, SPARK programs can be compiled by any ADA compiler.

The properties can be specified in the following ways:

- Pre and post-conditions to sub-programs.
- Loop invariants.
- Dataflow relationships (dependence of a variable on another).

The SPARK toolset to verify programs consists of the following:

- The Examiner performs static analysis to ensure that a program is well-formed and it generates verification conditions (or proof obligations) for the property specifications.
- The SPADE simplifier is an automated theorem prover that can discharge the verification conditions. This may however not be able to discharge all the proof obligations. The remaining proofs will have to be discharged manually.
- The SPADE proof checker can be used to check proofs that are discharged manually.

4.5. Malpas

The MALvern Program Analysis Suite (Malpas) is a toolset for the analysis of software programs [18]. Though this is primarily a static analysis tool, it can also provide a rigorous check that a program meets a specification. The program to be analyzed has to be in the Malpas Intermediate Language. In practice, this is easier than it sounds as there are automated translators from common programming languages and assembly languages to the Malpas Intermediate Language.

The static analysis portion of the toolset provides code metrics and analysis for dead code paths, uninitialized data, unexpected dependencies, etc. The formal analysis checks for mathematical conformance of the program to a specification. The specification is provided as:

- Pre and post-conditions.

- Invariants.
- Assertions.

This page intentionally left blank.

Chapter 5. Tools to Create a Provably Correct Design

The tools described previously are for "after the fact" verification, i.e., a design or a model is created and then verified for correctness. Design for correctness, by contrast, involves the use of formal techniques in the creation of the design itself, maintaining the required properties at every step of the design. When completed, the design is known to be correct by construction. This process starts with an abstract model or design that satisfies the required properties and then is iterated (refined) into a form that can be implemented. A notion of *refinement* is built-in to these tools to permit this traversal from specification to implementation while guaranteeing that safety, security and other properties are preserved. Overall a methodology for creating provably correct designs is provided by these tools.

5.1. VDM

The Vienna Development Method (VDM) is a collection of tools and techniques rooted in a common formal specification language. This toolset is used for modeling computer programs. Modeling is performed at a very abstract level but there exist techniques to transform an abstract model into a detailed design using a refinement process.

VDM originated at the IBM Laboratory in Vienna towards the goal of developing a compiler from a language definition [19]. As a loose collection of tools, the tools have been modified in divergent ways.

5.1.1. Data Types

The VDM Specification Language (VDM-SL) provides basic types such as Booleans, natural numbers, integers, rational numbers, real numbers, and characters. The basic types can be used to create unions, Cartesian products, and composite types. VDM-SL also supports higher level collections of types such as sets, sequences, and maps. In all, it provides a very rich set of data types and operators on these data types.

5.1.2. Model Creation

VDM supports functional modeling and state-based modeling. Functional modeling can be implicit or explicit.

Implicit modeling is the specification of the properties of a function, rather than its execution. This is done in the form of stating pre and post-conditions that form a contract that have to be satisfied by any implementation of the program.

Explicit modeling is the specification of the computation of the function.

State-based modeling is the description of the changes to global state variables by an operation.

5.1.3. Model Analysis

There are different tools for the analysis of VDM, both commercial and open source. VDMTools, a leading commercial tool, can analyze a VDM model and generate proof obligations. These tools can automatically check the correctness of properties that can be decided statically based on a type checker [20]. Other properties require a check of the dynamic behavior of the program. These properties are usually beyond the reach of the standard tools and may require an external theorem prover. By the nature of the models that can be specified, there may also be some properties that are undecidable.

5.1.4. Refinement

It is possible to start with an abstract VDM model and then iteratively add more detail to the model to arrive at an implementation. This refinement is a two-step process at each iteration.

Data reification This step expands an abstract data type into more concrete data types. To maintain the correctness of the abstract program it is necessary to show adequacy of the new data type.

Operation decomposition This step makes the changes to the model to operate on the newly expanded data types. To prove that the new operations and functions maintain the original properties, it is necessary to discharge proof obligations.

5.2. Z, B, Event-B, and Rodin

Event-B is a formal language based on set theory to describe abstract state machines[21]. Event-B is an evolution of the earlier B method, which in turn is derived from the Z notation, all originated by Jean-Raymond Abrial, a French computer scientist [22, 23]. Z, B, and Event-B are all based on set theory. Z and B are used for the specification of computer programs. Event-B is used for the specification and analysis of systems by modeling them as state machines.

A powerful feature of B and Event-B is the notion of refinement. this allows a user to start the modeling process with a very simple abstract model. Once the simple abstract model has been shown to be sound, additional detail can be incrementally added, i.e., the abstract model can be refined. The power of this technique is that to prove the soundness of a refined model, it is sufficient to take the proven abstract model and show that certain refinement properties hold. In this manner, a simple abstract system can gradually be refined into something that is close to the implementation of the system.

Rodin is a suite of tools to aid in the design and analysis of Event-B models. Rodin [24] is unique among other tools of its class for two reasons:

1. It spans abstraction levels with a refinement methodology.
2. It contains both a theorem prover and a model checker in the same tool operating on the same model.

For reasons mentioned in Section 1.3 it is important to have the ability to represent many levels of abstraction systematically. It is best and most effective to be able to do this in the same environment using the same modeling language. This way both model checkers (best for straightforward but tedious verifications) and theorem provers (best for verifications where some higher level insight must employed) can be simultaneously applied to the design.

Almost all digital designs begin as a function that cannot be implemented in hardware directly but must be broken down and translated into a representation that can be implemented in silicon, similar to the job of a compiler. Likewise, a requirement for the high-level design must systematically track the process of successive refinement from original concept to its reification in silicon. Event-B, and the Rodin framework that facilitates concrete designs in Event-B, seek to do exactly this. Starting with a high-level model that is, in effect, an executable specification, the model, along with its formal requirements, is successively refined until a design implementable in silicon is achieved.

The Event-B computational model is that of a state machine (see Listing 5.2). Built-in to the Event-B [21] modeling language used by Rodin is a methodology for traversing various levels of abstraction:

1. Begin with a model specification for the system at a high level of abstraction that is closest to the designer's original conception, but is too high-level to be implemented in a device.
2. "Refine" the model to a new model that has a one-to-one correspondence to the original but contains more detail so that it more closely resembles the semantics of an implementation.
3. From an analysis of these two different models (or in Rodin vernacular: "machines") the Rodin tool automatically creates "proof obligations" that must be discharged (i.e., proven) to ensure consistency between the two

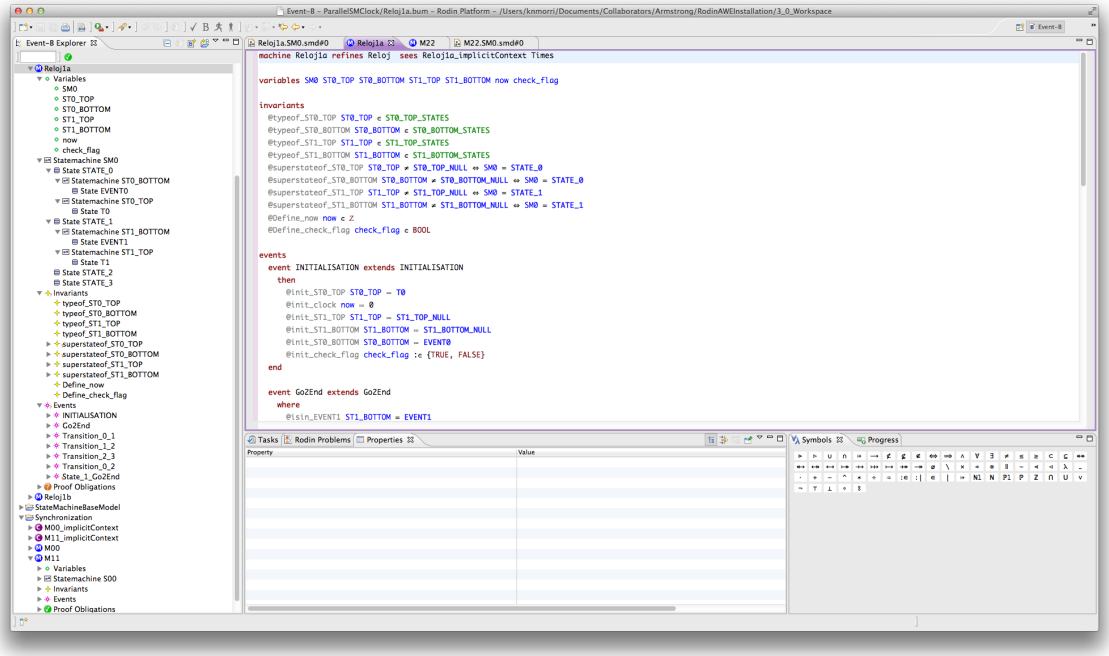

Figure 5.1. Rodin Tool

models.

4. While the tool itself can discharge most of these proofs automatically, a few may need to be proven “by hand”.
5. Steps 2 through 4 are repeated until the an implementation can be produced.

What is particularly attractive about the Rodin tool is this ability to carry formal requirements from the specification level all the way through to its implementation.

5.2.1. Rodin Framework

The Rodin Framework is an Eclipse-based platform for developing and verifying Event-B models (Figure 5.1). As Event-B is a very general event description technique with refinement, it is possible to build additional capability on top of it. The Rodin platform supports many plugins that layer on the base Event-B analysis capability and could add to it.

One such set of plugins that could be very useful for the design of systems is a plugin for composition and decomposition.

Another domain that is close to hardware implementation is the representation of models as traditional hierarchical state machines. The state machine model mirrors the state machine subset of UML. The UML-B plugin allows the specification of hierarchical state machines (Figure 5.2). Currently, the analysis takes models such as these and flattens them into Event-B for analysis. This is not ideal, since the hierarchy and scope which could make formal verification easier is lost in this process.

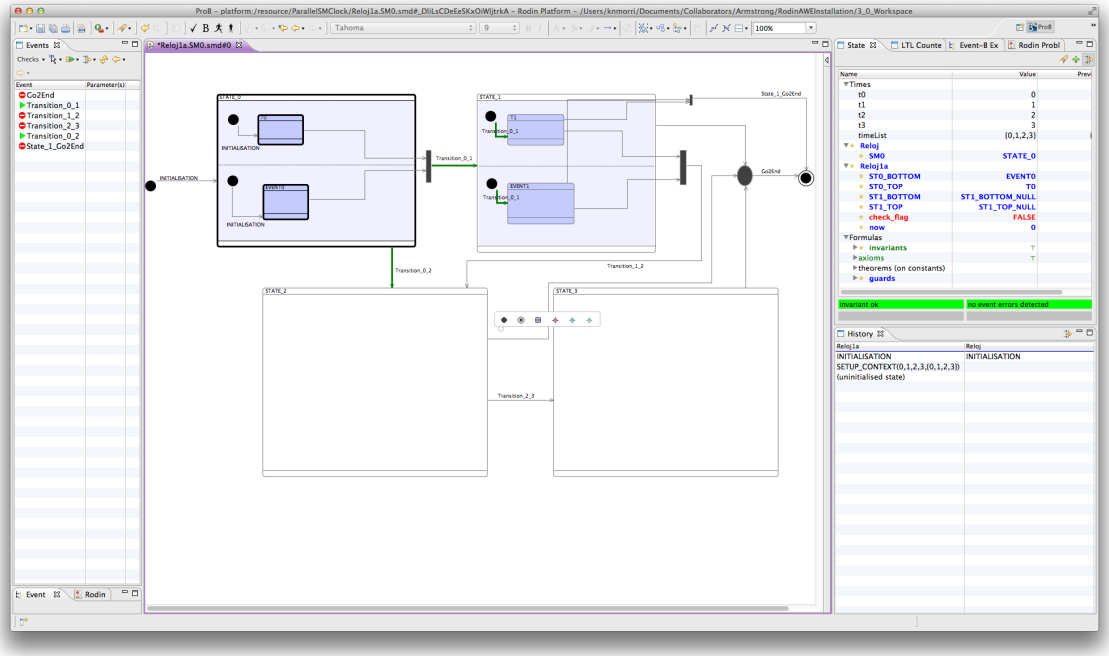

Figure 5.2. Rodin Tool showing a UML design

5.2.2. Applications to real systems and current limitations

Rodin/Event-B has been used in the design and verification of control algorithms for railway switches. Specifically, Rodin/Event-B is used for ensuring that safety properties hold.

Rodin/Event-B is the most advanced tool of its kind but it still has limitations that preclude its use in many use cases.

- Currently, Event-B refinements maintain "safety" properties, but "liveness" properties are not maintained. This means that a null refinement that does nothing is a valid refinement. For showing functional properties, this is insufficient. Refinement should also ensure that the refined system actually can perform the desired function.
- The Event-B language describes models without using name-spaces, scoping or hierarchy. The flat nature of Event-B models makes it difficult for designers to describe all but the smallest models.
- There exist mechanisms for composition and decomposition to modularize a model. However, this involves tedious manual intervention.
- Rodin currently uses closed-source theorem-provers and model-checkers from Atelier-B (http://www.atelierb.eu/en/). Extensibility to other theorem-provers and model-checkers would facilitate analysis.
- Event-B does not have a notion of time. Though there are ways to do this using explicit counting variables, these are very restrictive in practice.

```
values
-- This creates a representation of the junction
  p1 : Path = mk_token("A1North");
  p2 : Path = mk_token("A1South");
  p3 : Path = mk_token("A66East");
  p4 : Path = mk_token("A66West");

  lights : map Path to Light
         = {p1 |-> <Red>,  p2 |-> <Red>,  p3 |-> <Green>,    p4 |-> <Green>};

  conflicts : set of Conflict
            = {mk_Conflict(p1,p3), mk_Conflict(p1,p4), mk_Conflict(p2,p3), mk_Conflict(p2,p4),
               mk_Conflict(p3,p1), mk_Conflict(p4,p1), mk_Conflict(p3,p2), mk_Conflict(p4,p2)};

  kernel : Kernel = mk_Kernel(lights,conflicts)

types
  Light = <Red> | <Amber> | <Green>;
  Time = real
  inv t == t >= 0;
  Path = token;

  Conflict :: path1: Path
              path2: Path
  inv mk_Conflict(path1,path2) == path1 <> path2;

-- the kernel data structure has two components representing
-- 1) a mapping with the current status of the lights for each
-- direction and 2) an unordered collection of conflicts between
-- paths.

  Kernel :: lights    : map Path to Light
            conflicts : set of Conflict
  inv mk_Kernel(ls,cs) ==
        forall c in set cs &
              mk_Conflict(c.path2,c.path1) in set cs and
              c.path1 in set dom ls and
          c.path2 in set dom ls and
              (ls(c.path1) = <Red> or ls(c.path2) = <Red>)

functions
-- changing the light to green for a given path
  ToGreen: Path * Kernel -> Kernel
  ToGreen(p,mk_Kernel(lights,conflicts)) ==
    mk_Kernel(ChgLight(lights,p,<Green>),conflicts)
  pre p in set dom lights and
      lights(p) = <Red> and
      forall mk_Conflict(p1,p2) in set conflicts &
             (p = p1 => lights(p2) = <Red>);

-- changing the light to red for a given path
  ToRed: Path * Kernel -> Kernel
  ToRed(p,mk_Kernel(lights,conflicts)) ==
    mk_Kernel(ChgLight(lights,p,<Red>),conflicts)
  pre p in set dom lights and lights(p) = <Amber>;

-- changing the light to amber for a given path
  ToAmber: Path * Kernel -> Kernel
  ToAmber(p,mk_Kernel(lights,conflicts)) ==
    mk_Kernel(ChgLight(lights,p,<Amber>),conflicts)
  pre p in set dom lights and lights(p) = <Green>;

  ChgLight: (map Path to Light) * Path * Light -> (map Path to Light)
  ChgLight(lights,p,colour) ==
    lights ++ {p |-> colour}
```

Listing 5.1: A VDM model for a junction traffic light controller (example provided by VDM)

```
machine home_alarm sees home_alarm_context
variables time armed_indicator  alarm_enable sensor_err alarm_activated sensor_settled sensor_tripped

invariants
  @def_time                time IN N
  @def_armed_indicator     armed_indicator IN BOOL
  @def_alarm_enable        alarm_enable IN BOOL
  @def_alarm_activated     alarm_activated IN BOOL
  @def_sensor_settled      sensor_settled IN BOOL
  @def_sensor_err          sensor_err IN BOOL
  @def_sensor_tripped      sensor_tripped IN BOOL

  @prop1  (alarm_activated = TRUE ) IMPLIES (sensor_settled = TRUE AND
          alarm_enable = TRUE AND sensor_err = FALSE AND  sensor_tripped = TRUE )
  @prop2  (armed_indicator = TRUE ) IMPLIES (alarm_enable = TRUE)

events
  event INITIALISATION
    then
      @act4 armed_indicator = FALSE
      @act5 alarm_enable = FALSE
      @act6 alarm_activated = TRUE
      @act7 sensor_err = FALSE
      @act8 sensor_settled = FALSE
  end

  event  set_alarm
    then
      @act1  alarm_enable = TRUE
  end

  event reset_alarm
     then
      @act1  alarm_enable = FALSE
  end
  event sensor_fault
    then
      @act1 sensor_err = TRUE
  end

  event sensor_settle
    where
      @grd1 alarm_enable = TRUE
      @grd2 sensor_settled = FALSE
    then
      @act1 sensor_settled = TRUE
      @act1 time = 0
  end

  event sensor_settle_delay
    where
      @grd1 sensor_settled = TRUE
      @grd2 time = 2
    then
      @act1 armed_indicator = TRUE
  end

  event sense_event
    where
      @grd1 alarm_enable = TRUE
      @grd2 alarm_activated = FALSE
      @grd3 sensor_settled =  TRUE
      @grd4 sensor_fault =  FALSE

    then
      @act1  sensor_tripped = TRUE
      @act2  time = 0
  end

  event sense_event_delay
    where
      @sensor_tripped = TRUE
      @grd1 time = 5
    then
      @act1 alarm_activated = TRUE
  end

end
```

Listing 5.2: Event-B model of a flawed alarm system

Chapter 6. Summary

Even in systems with low complexity, ensuring that the behavior is correct under *all* input conditions is difficult to ensure through testing and simulation. Safety, security, and often reliability requirements are even more open-ended and cover a vast state space that can only be validated with a formal approach. The aim of this survey is to identify existing formal methods tools and techniques for designing and verifying the correctness of digital systems. The survey is not exhaustive but it captures the variety of the existing tools and describes the most common ones. Given below are recommendations for tools for use in specific types of digital designs.

6.1. Verifying the Correctness of a Design

6.1.1. FPGA and ASIC based development

Control-logic dominated designs Many FPGA and ASIC designs involve controllers for different applications. These tend to be dominated by control logic. These are eminently suitable for formal verification. At present, JasperGold by Jasper Design Automation (acquired by Cadence in 2014) is one of the leading tools in this area.

Designs with data security needs Some designs involve an implementation of information security. These designs tend to have security requirements on data leakage or data dependency. These types of properties are not easily expressed as SVA assertions. However, some of the existing COTS RTL verification tools have add-ons that provide the capability to express such properties and perform analysis. In this area, JasperGold has an add-on called Security Path Verification that may be suited to this. There are also some newly available tools targeted to these types of properties (OneSpin 360 DV from OneSpin).

Designs with signal processing algorithms Formal verification of algorithms is currently difficult with the existing tools. There are no COTS tools that can do this directly at present.

6.1.2. Software Based Systems

At present, only limited properties of software based systems can be verified. The analysis capability is very dependent on the type of software used for the design. If ADA is used as a development language, the SPARK tools are reasonably advanced in this regard. For C language software, Frama-C is a reasonable choice.

6.2. Looking Forward: Creating Designs that are Correct by Construction

To enable large scale designs that are designed to be correct, as opposed to being checked after design completion, the following properties are needed in a tool:

Abstract modeling Allow the creation of simple high-level models that can be verified before jumping into the details.

Refinement Iterative allow the addition of detail to a model to make it look closer to implementation.

Composition and decomposition Provide automated techniques to infer properties of a system using the properties of its sub-components. Provide techniques to modularize a system so that it can be divided into modular components, each of which can be verified.

Adaptability A tool should be adaptable to different problem domains.

Maximum automation with human guidance To prove complex properties, it is desirable to have as much automation available as possible. There will be a limit to this and it should be possible to provide manual input to capture human reasoning behind a design.

The tool that comes closest to these goals is the Rodin/Event-B tool. Its modeling representation is ideal for digital designs that are consciously designed to be state machines.

Enhancing Rodin/Event-B in the following ways would go a long way towards making it more accessible.

- Enhance the Event-B theory of refinement to support liveness. Currently, Event-B refinements maintain "safety" properties, but "liveness" properties are not maintained. This means that a null refinement that does nothing is a valid refinement. For showing functional properties, this is insufficient. Refinement should also ensure that the refined system actually can perform the desired function.
- Introduce scoping and name-spaces. The flat nature of Event-B models makes it difficult for designers to describe all but the smallest models.
- Simplify composition and decomposition. It would be best if this were automated.
- Rodin currently uses closed-source theorem-provers and model-checkers from Atelier-B (http://www.atelierb.eu/en/). Allow the use of external theorem-provers and model checkers that can be extended to support specialized domains.
- Incorporate a notion of time. Though there are ways to do this using explicit counting variables, these are very restrictive in practice.

Augmenting Rodin in this way would make it a viable tool for designing digital systems that are correct by construction.

References

[1] G. J. Holzmann, *The Spin Model Checker*. Addison Wesley, 2004.

[2] M. Ben-Ari, *Principles of the Spin Model Checker*. Springer, 2008.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal - a tool suite for automatic verification of real-time systems," (New Brunswick, New Jersey), Oct 1995.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M.Roveri, "NuSMV: a new Symbolic Model Verifier," in *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)* (N. Halbawchs and D. Peled, eds.), no. 1633 in Lecture Notes in Computer Science, (Trento, Italy), pp. 495–499, Springer, July 1999.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, LNCS, (Copenhagen, Denmark), Springer, July 2002.

[6] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[7] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[8] P. Broadfoot and B. Roscoe, "Tutorial on FDR and its applications," in *SPIN Model Checking and Software Verification* (K. Havelund, J. Penix, and W. Visser, eds.), vol. 1885 of *Lecture Notes in Computer Science*, pp. 322–322, Springer Berlin Heidelberg, 2000.

[9] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[10] "Simulink design verifier." http://www.mathworks.com/.

[11] *1850-2005 IEEE Standard for Property Specification Language (PSL)*. IEEE, 2005.

[12] *1800-2009 IEEE Standard for Property Specification Language (PSL)*. IEEE, 2009.

[13] *BLAST*. http://mtc.epfl.ch/software-tools/blast/index-epfl.php, 2008.

[14] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering," *Int. Journal on Software Tools for Technology Transfer*, vol. 9, pp. 505–525, 2007.

[15] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, April 2000.

[16] K. Havelund, "Java pathfinder, a translator from java to promela," in *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, (London, UK, UK), pp. 152–, Springer-Verlag, 1999.

[17] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[18] J. Webb and D. Mannering, "Malpas — verification of a safety critical system," in *Achieving Safety and Reliability with Computer Systems* (B. Daniels, ed.), pp. 44–58, Springer Netherlands, 1987.

[19] D. Bjorner and C. Jones, *The Vienna Development Method.. The Meta-Language*, vol. 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[20] S. D. Vermolen, *Automatically Discharging VDM Proof Obligations using HOL*. PhD thesis, Radbound University Nijmegan, 2007.

[21] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.

[22] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen, "The B-method," in *VDM Europe*, vol. 2, pp. 398–405, 1991.

[23] *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[24] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *STTT*, vol. 12, no. 6, pp. 447–466, 2010.

DISTRIBUTION:

1	MS	8953	Clay, Robert, 08953
1	MS	8961	Vanderveen, Keith, 08961
1	MS	8961	Armstrong, Robert, 08961
1	MS	9110	Punnoose, Ratish, 08229
1	MS	0899	Technical Library, 8944 (electronic copy)

Sandia National Laboratories