Verifying Properties of Hardware and Software
by Predicate Abstraction and Model Checking

Randal E. Bryant
Carnegie Mellon University, Pittsburgh, PA, USA.
Randy.Bryant@cs.cmu.edu

Sriram K. Rajamani
Microsoft Research, Redmond, WA, USA.
sriram@microsoft.com

Abstract
This tutorial describes automatic techniques for formally
verifying hardware and software by creating Boolean ab-
stractions of the underlying unbounded system state vari-
bles.

Keywords
Formal verification, Predicate abstraction, Model Check-
ing, Symbolic execution, decision procedures

INTRODUCTION
The historical development of formal verification tech-
niques for hardware and for software followed different
paths. For hardware, most verification operates at the bit
level, where the system state is represented as a finite set of
Boolean state variables. Highly automated techniques,
such as symbolic model checkers, have been developed to
prove properties about the set of all possible reachable
states of these finite-state systems. For software, classic
verification operates at an infinite-word level, where pro-
gram variables take values over infinite domains such as
integers or memory addresses. For notations that support
possibly recursive procedures, program execution may also
create an unbounded number of variables, accessed by a
stack discipline. Verification of such models typically re-
quires proving theorems by induction, either manually or
with the aid of automatic theorem provers.

In recent years, the domains of hardware and software ver-
ification have begun to converge, with hardware being
modeled at more abstract levels, and with the development
of techniques to extract and model check Boolean abstrac-
tions of infinite-state systems.

This presentation describes two verification tools: UCLID, a
tool developed to model and verify hardware systems mod-
eled at a word level, and SLAM, a tool developed to auto-
atically prove properties of programs written in the C
programming language. Both support predicate abstrac-
tion [2], in which the state of the system is characterized by
a set of Boolean predicates, describing properties and rela-
tions among the state variables, yielding a Boolean abstrac-
tion of the underlying system. With UCLID, this abstraction
yields a finite-state model of the system, while SLAM sup-
ports a stack-based Boolean Program model. Techniques
similar to those used in symbolic model checking can then
be used to characterize the set of reachable states of a sys-
tem. In the case of Boolean Programs, the stack can be
handled by combining symbolic model checking with com-
piler-style dataflow analysis.

The Carnegie Mellon University UCLID Project
The UCLID verifier [7] is designed to verify infinite-state
models of hardware systems. The system state consists of a
set of state variables, each having a next-state function.
State variables can be Boolean, unbounded integers, or
functions mapping integer arguments to either integer or
Boolean values. Functional state variables are useful when
modeling different memory structures. For example, a ran-
dom-access memory can be viewed as a function mapping
an integer address to the value stored at that address. This
form of modeling abstracts away many system details, such
as the sizes and encodings of data words, and system pa-
rameters such as memory sizes.

System behavior is defined in terms of the next-state func-
tions for the state variables. These functions are expressed
in a notation combining Boolean operations and restricted
forms of integer operations and function definitions. Ab-
stract functional behavior is described in terms of uninter-
preted functions, where the verifier treats a functional unit
as a “black box” and must show that the system will obey
the specified properties regardless of the specific function
computed by the unit. Integer operations are restricted to
increment, decrement, and comparison operations. These
operations are used mainly when modeling the pointer be-
behavior of different stacks and queues in the system.

UCLID supports several different verification techniques,
including bounded model checking, correspondence check-
ing, and invariant generation by predicate abstraction [3].

UCLID has been used to verify models of a number of sys-
tems, including both in-order and out-of-order processors,
directory-based cache protocols, and mutual exclusion pro-
ocols.
Applying UCLID to actual hardware designs requires generating the abstract model from a register-transfer language (RTL) description. A preliminary tool has been developed for this purpose [1], but more work is required to generalize this capability.

The Microsoft SLAM Project

The SLAM verification toolkit [4][5] checks properties of programs written in C, using a combination of predicate abstraction, symbolic execution, model checking, and abstraction-refinement. The SLAM toolkit works by automatically constructing and refining abstractions called Boolean Programs. A Boolean Program is a C program with only Boolean variables. Each Boolean variable represents a predicate over the variables in the corresponding scope of the C program. Boolean Programs are constructed using $\text{c2bp}$, which implements predicate-abstraction adapted to handle software features such as procedure calls and pointers. We model check Boolean programs using the model checker $\text{b2lbp}$, which implements interprocedural data-flow analysis using Binary Decision Diagrams. Automatic counter-example driven refinement is done using $\text{NEwTON}$, which implements symbolic execution with predicate generation.

We have applied SLAM to check the use of the Windows I/O manager interface by device driver clients written both at Microsoft and by third-party software developers. The properties range from simple rules about ordering of function calls to data-dependent properties, which require the tool to keeping track of value-flow and aliasing.

While the Boolean Program model is a viable approach for checking properties that are strongly dependent on control flow of sequential programs, other approaches are needed for reasoning about concurrency and deeper properties of the heap. Recently, we have started building another tool $\text{ZINC}$ [6], which has a richer modeling language with dynamically created threads and objects. $\text{ZINC}$ was initially designed as an explicit-state model checker, but we have recently added support for symbolic execution by using a mixed state representation containing both explicit and symbolic components.

REFERENCES


