Application Specific Higher Order Logic Theorem Proving*

Daniel Kroening
Computer Science Department
Carnegie Mellon University
`kroening@cs.cmu.edu`

Abstract

Theorem proving allows the formal verification of the correctness of very large systems. In order to increase the acceptance of theorem proving systems during the design process, we implemented higher order logic proof systems for ANSI-C and Verilog within a framework for application specific proof systems. Furthermore, we implement the language of the PVS theorem prover as well-established higher order specification language. The tool allows the verification of the design languages using a PVS specification and the verification of hardware designs using a C program as specification. We implement powerful decision procedures using Model Checkers and satisfiability checkers. We provide experimental results that compare the performance of our tool with PVS on large industrial scale hardware examples.

1 Introduction

1.1 Challenge

Formal verification of complex systems such as hardware and software designs is the major thrust within the theorem proving community in the past years. Formal verification is applied in a wide range of domains. First of all, in the domain of safety critical systems a full proof of correctness is most desirable. Examples of live-critical embedded systems include medical devices and controllers in the avionics or automotive industry. Because of the high cost of design faults, theorem proving is already common in this area. Examples include the formal verification of a fault tolerant communication bus protocol [PSvH99, Pfe00] using PVS or the verification of tools for train borne control software systems [BT00] using ACL2 [KM96].

In case of the chip industry, design faults are expensive due to shortening time-to-market. A well known example is the bug in Intel's Pentium floating point unit [V. 95]. Despite of the progress of symbolic Model Checking, state of the art microprocessors are still too complex for completely automated formal verification methods. Theorem proving is currently the only technique known that is able to handle designs of this complexity. Examples of theorem proving within the microprocessor industry include the work of David Russinoff [Rus00b] on the correctness of the floating point units of the AMD Athlon processor series using ACL2 and of John Harrison on Intel's Merced [Har99] using HOL [CGM86].

However, theorem proving has not yet become an integral part of the design process. Main obstacles are the high amount of manual work required by theorem proving systems. We discuss two issues that are part of this problem. The first issue is the language that is used for design, specification and verification, and the second is automation of the proof itself.

Design Language The formal verification of hardware and software designs usually includes that the original design, given in a programming or hardware description language, first has to be translated to the native language of the theorem prover. If done manually, this translation process is error-prone, since while formalizing the design, the person doing the translation is likely to translate the "desired behavior" instead of a bug.

*This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, the U.S. government or any other entity.

The design is often harder to understand after translation into the language of the theorem proving system. This holds particularly for sequential programming languages, such as ANSI-C or Java. In case bugs are found while trying to prove the correctness of the design, one is required to understand the nature of the bug given an output in the native language of the theorem prover. The bug has to be fixed in the design language and then the translation has to be repeated.

An alternative approach is to write the design in the high level, native language of the theorem proving system. The design is later on obtained by translating or compiling the high level languages into the design language. However, this approach is not accepted in many areas because of existing code, that is to be re-used, and because of low efficiency of the translated code. This is a particular problem for embedded systems designers, where code size and speed are crucial for the total cost of the product.

We therefore think that the language a theorem prover uses is of high importance for the acceptance of theorem provers as a design tool. The theorem prover should accept both hardware and software designs in the original, low level design language. This allows efficient designs and formal verification without translation into a specific theorem proving language. Feedback of any kind, e.g., subgoals in case of an interactive theorem prover, or about bugs in the design, should be given in the original language as well.

Specification Language While the design language is intended for efficient compilation of the hardware or software product, the specification, in contrast, should be given in a concise high level language. In [Gor85], Gordon motivates the use of higher-order logic for the specification of hardware. We believe that the specification language should not be application specific. Higher-order logic ensures that the specification language applies to the widest range of applications. In [Rus97], Rushby describes the advantages of sub-typing, as supported by PVS, for writing easily understandable and concise specifications.

Proof Automation Most large practical verification problems in the hardware and software domain are solved compositional, i.e., the design is manually divided into smaller sub-problems that are then verified independently. This allows reducing the large problem into a number of problems that are small enough to be solved by efficient automated decision procedures. These decision problems are often very application specific. However, general purpose theorem proving systems often lack efficient decision procedures for these problems. They then have to be solved manually, which is an unnecessary burden.

Both for low level programming languages, such as ANSI-C and Java, and, obviously, hardware design languages, decision support for bit vector arithmetic is highly desirable. This includes operations such as addition, multiplication, shifting and bit-wise operations on fixed-length bit vectors. However, in most existing theorem proving systems automated decision support for bit vector arithmetic is neglected.

The correctness proof of control logic of hardware designs is often very tedious using theorem proving systems. However, because of their small state space, it is a simple problem for Model Checkers. Thus, many theorem proving systems integrate Model Checkers as decision procedure, e.g., PVS [ORS97, ORSSC98, Rus00a]. The theorem prover is used to abstract the data paths, which makes Model Checking feasible.

Besides the lack of application specific decision procedures, the obstacle for the verification of large designs is the size of the decision problems. In general, the decision procedures of most theorem provers are not optimized for large decision problems. As an example, the correctness of many hardware designs, such as arithmetic units, is a pure propositional logic problem. Current decision procedures are able to handle even complex designs with thousands of gates completely automated. However, such a problem is just too big for the decision procedures of theorem provers such as ACL2 or PVS and requires a manual proof.

1.2 Related Work

Boulton et al. [BGG+92] formalize the semantics of several hardware description languages in HOL. They provide support of viewing problems in the original design language by translating back HOL to the original HDL. The specification can be done in higher order logics. They do not discuss application specific decision procedures for the verification problem. In [Bou97], the authors provide a framework to automate the integration of an application specific language into a formal reasoning tool. Support for generating parsers and pretty-printers and internal representations is provided.

As described above, PVS integrates a Model Checker in order to decide smaller sub problems automatically. It also provides means to abstract data types in order to make problems finite or small enough. It does not provide support to input or output application specific languages. ICS [FORS01], which will be integrated in future versions of PVS, provides automated support for bit vector decision problems. However, bit-vector multiplication and shifting by variable distance is not supported.

While the PVS language allows concise specifications, the type checking problem becomes undecidable. Our experiments with PVS show that the speed of the type check becomes critical for large projects; the verification of the actual proof is faster than the type check. The type check has to be performed again after any change to the theory, so it is part of the interaction.

The Stanford Pascal Verifier [Luc79] takes Pascal as input language and contains decision procedures specialized for the task. However, it does not contain further reasoning capabilities. Non-trivial verification conditions have to be verified by means of an external, generic theorem prover. The extended static checker [DLNS98] takes Java programs as input and also contains decision procedures that are specialized for this task; however, the tool focuses on specific, simple properties and does not aim at a full proof of correctness.

Other application specific verification tools include AX / SPIN and the SLAM project for ANSI-C. However, these tools lack a full higher-order logic theorem prover. The aim of the LOOP project is to formally verify Java programs by conversion to the language of a variety of theorem provers. However, the proofs are done using the language of the theorem prover. The specifications are done in a special language.

The combination of a functional programming language and theorem provers is much more natural than the combination of sequential programming languages and theorem provers. Thus, there are theorem provers that accept functional programming languages as input. ACL2 is both used as executable programming language and as specification language for the theorem prover. Another example is OCaml and Nuprl. A significant advantage is that this allows the theorem prover to argue about its own code.

When designing application specific theorem proving systems, many components can be re-used. The SyMP framework [Ber01, Ber02] allows the integration of application specific proof systems. The user interface and the proof manager are shared among all proof systems. Every proof system can provide an application specific input/output language. As examples, SyMP provides languages for security protocols and a language that is used for the verification of cache coherence protocols. In one of the proof systems, the Model Checker SMV is tightly integrated by extending the Gentzen sequent with an explicit model and temporal operators. The interaction with the user is done using the application specific language only.

However, the framework does not integrate the application specific languages by means of an internal higher-order logic representation. The disadvantage of this approach is that it prohibits the sharing of proof commands and decision procedures. No common specification language is provided.

1.3 Contribution

We provide theorem proving support for application specific design languages by extending the SyMP framework by a common, higher-order logic internal representation. All design languages are converted into this representation. Design languages currently implemented are ANSI-C, Verilog, and the SMV language. In addition to the design languages, we import the language of the PVS theorem prover, with minor restrictions. This language is used as common specification language for all design languages. In comparison with PVS, we improve the speed of the type check of PVS input files significantly. The user is able to pick the language for the interaction. This can be any design language or the specification language. The language can be switched while proving a theorem.

Besides the usual set of higher-order logic proof rules, we implement decision procedures that allow verifying larger sub-problems. We implement a SAT based decision procedure for bit vector arithmetic including multiplication. We implement a equality logic decision procedure that automatically applies user provided lemmas in the style of rewriting rules. In contrast to prior approaches, the algorithm is SAT based and allows large decision problems and, on the propositional level, arbitrary rules.

1.4 Outline

In section 2, we describe the overall framework of our approach. In section 3, we describe the hardware description language support, the ANSI-C support, and details of the PVS language module. In section 4, we describe the decision procedures.

2 Framework

Figure 1 shows an overview of the tool and the framework. Our tool is integrated into the SyMP (Symbolic Model Prover) framework, which was developed by Sergey Berezin [Ber02].

SyMP provides the proof manager, which manages the proof trees and controls the interactive proof construction. The proof manager interfaces to the user interface and the proof systems. SyMP comes with an Emacs user interface, which is much like

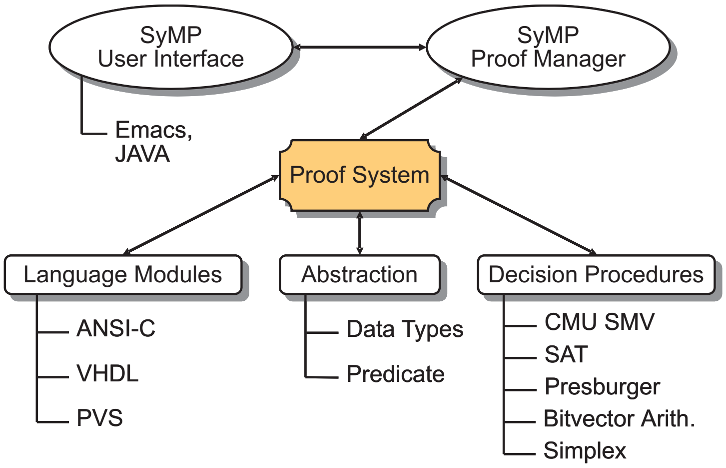

Figure 1: Tool Overview

the user interface of PVS. Flavio Lerda has implemented a new, graphical user interface in Java, which allows ”push-button” proof construction.

While the user interface and the proof manager are language and application independent, the proof systems of SyMP are application specific. They provide language support by implementing the parser and pretty-printer. We extend the SyMP framework by implementing one proof system that supports multiple, application specific languages by means of language modules.

A language module translates a given input language into a common, higher-order logic internal representation. It also translates the internal representation back into the application specific language for displaying proof obligations in the original language. This way, the engineer uses the design language during the whole design and verification process. The theorem prover integrates well in the design process.

The proof system also contains a common set of proof rules and decision procedures, which are shared. Thus, the proof rules and decision procedures are implemented only once and are used for all languages.

This concept also allows comparing designs and specifications that are equivalent but made in different languages. For example, one can compare ANSI-C designs to hardware designs given in Verilog or PVS language.

Internal Representation The internal representation is a tree-like data structure allowing arbitrary higher-order logic. After type check, every node in the tree is annotated with a type. Of all languages we implement, PVS has the most expressive type system. We therefore use the same type system as implemented by PVS: A type consists of a base type and a predicate. The base type is a basic type, such as the number type or an enumerative type, or combinations of basic types (tuples, records, ...). The predicate allows generating sub-types from basic types.

3 Language Modules

3.1 Hardware Description Languages

We implement conversion into and from the internal representation for Verilog [TM91] and the language of SMV, an automated Model Checker. This allows reading hardware designs directly into the tool. The formalization of the languages in higher-order logics is simple. In case of Verilog, we assume that the design is given as clocked circuit with only one clock. Besides the registers, the circuit has to be completely combinatorial. We therefore import a record type, which represents the registers, and a transition function. We do not support any non-synthesizable language features. We do not generate theorems from Verilog files.

In case of the SMV language, we import a record type, which represents the defined variables and their types, the transition relation, and an initial state predicate. We also import any given specification, which may contain temporal operators, as a theorem.

3.2 ANSI-C

3.2.1 Formalization

The semantics of ANSI-C are defined in the 1999 ISO/IEC C Standard (”C99”) [Int99], henceforth referred to as ”the standard”. They provide many options to the implementation of the compiler. Examples include the bit codings of the number types, padding of structures, signedness of several data types, and the sub-expression evaluation order. This has both advantages and disadvantages. The advantage is that ANSI-C can be used in many environments, and easily and efficiently adapts to many different architectures.

The disadvantage is that the behavioral semantics of ANSI-C programs are quite often not well defined or depend on the actual architecture. This proposes a challenge for formal verification.

One of the main parts of the project therefore is the formalization of the semantics of ANSI-C. For this task, we extend Hoare's logic with concepts required for ANSI-C such as functions with side effects, pointers and so on. We offer two ways to handle the options allowed by the standard:

1. **Fix it.** One option is to allow the user to pick a particular option. For example, we require the user to define the number of bits of the ANSI-C arithmetic data types such `int` and `char`.

2. **Verify them all.** Committing to one option is not useful in several cases. One example is the evaluation order of sub-expressions. For example, in the expression

   ```
   a = f(x) + g(x);
   ```

   the order in which the functions `f` and `g` are called is not defined by the standard. The compiler may call them in any order. In this situation, we require that the behavior of the circuit shall match the behavior of the program regardless of the order in which `f` and `g` are executed, i.e., we allow the program to pick an ordering nondeterministically.

Another example is the semantics of several operators, such as bit-wise shifting. These operands only return well-defined results on signed integers in case the operands are not negative. We return an arbitrary result, which is chosen nondeterministically, in case an operand is negative.

We formalize the semantics of the ANSI-C standard using an extended but traditional Hoare logic, which is easily embedded within higher-order logic. Hoare's logic allows tools that are intuitive to use for software engineers since technical details such as program counters (PC) are hidden by the logic.

Using the traditional Hoare axioms, the proof of correctness of the program is constructed by beginning at the end of the program. Using the traditional axioms, we conclude rules that allow to construct the proof in program execution order, which is more intuitive for engineers. This is the ”forward-version” of the assignment axiom:

$$\frac{\{\exists a' : p[a/a'] \wedge a = t[a/a']\} S \{q\}}{\{p\} \quad a = t; \quad S \quad \{q\}}$$

After skolemization of the existential quantifier, this rule technically is just variable renaming, which can be performed automatically and efficiently.

There is a similar rule for arrays, which includes constraints for the array bounds in order to assert that the semantics of the array access are well-defined. Note that most security flaws in software systems written in ANSI-C are caused by out-of-bounds pointers.

$$\frac{\{\exists a' : p[a/a'] \wedge a[j] = \left\{ \begin{array}{lcl} t[a/a'] & : & j = i \\ a'[j] & : & \text{otherw.} \end{array} \right. \} S\{q\}, i \geq 0 \wedge i < size(a)}{\{p\} \quad a[i] = t; \quad S \quad \{q\}}$$

There are similar rules for all other ANSI-C constructs, such as `for`, `switch`, function calls and so on.

3.2.2 Pointers and Dynamic Memory

ANSI-C programs make heavy use of pointers. They are used to implement arrays and pass-by-reference for function call arguments.

Let $\mathcal{V}$ be the set of variables (before renaming), and let $\mathcal{F}$ be the set of functions. We assume that both are disjunct. Furthermore, the special symbol `NULL` must not be an element of either set.

Pointers are modeled as tuple with two components (v, o). The first component v is an element of the set of variables (before renaming), the set of functions, or the special value `NULL`. Let $p.v$ denote the first component of pointer p, $p.o$ the second.

$$p = (v, o)$$

$$p.v \in \mathcal{V} \cup \mathcal{F} \cup \{\mathtt{NULL}\}$$

If $p.v$ is an element of the set of variables, $p.o$ is used to denote the offset within that variable. Otherwise, $p.o$ is not used. Pointers are dereferenced using the following rule: if $p.v$ is equal to variable $a \in \mathcal{V}$ and $p.o$ is within the bounds of the variable, $*p$ is equal to a. Variables that are not of an array type have size 1.

$$p.v = a \wedge p.o \geq 0 \wedge p.o < size(a) \quad \Longrightarrow \quad *p = a[p.o]$$

Example: Consider the following code fragment:

```
int a[4], b, *p;
if(x) p=&a[2]; else p=&b;
```

After the execution of the second line, the pointer p may have two values, depending on x: $(a, 2)$ or $(b, 0)$.

In many cases the constant propagation provides fixed values for both components. In this case, a read or write access to a dereferenced pointer is handled just as a variable read or write access. However, in practice there are multiple possibilities if assignments are made to pointers that depend on input variables. In this case, a case split that considers all possible values of the two components is performed. This can yield significant blowup.

We optionally add subgoals that assert that p points to a valid variable and that the offset is within the bounds of the variable (left hand side of implication above). This allows checking whether exceptions can occur.

Pointer Arithmetic Pointer arithmetic on p is performed by bit vector arithmetic on $p.o$:

$$p + i \quad = \quad (p.v, p.o + i)$$

The difference of two pointers is only defined if the pointers point to the same variable $a \in \mathcal{V}$. We add a subgoal $p.v = q.v$.

$$p.v = a \wedge q.v = a \quad \Longrightarrow \quad p - q = p.o - q.o$$

Dynamic Memory Allocation We verify programs that make use of dynamic memory allocation, e.g., for dynamically sized arrays or data structures such as lists or graphs. This is easily realized by replacing every call to `malloc` or `calloc` by the address of a new variable. For this, we assume that the type of the new variable is given by ether an explicit or implicit type cast to a pointer that points to a variable of type t. In case of `malloc`, let x be a new variable of an array type with elements of type t and size s/`sizeof(`t`)`. We assert that s is an integer multiple of `sizeof(`t`)`.

$$(t\ \texttt{*})\ \texttt{malloc(}s\texttt{)} \quad \longrightarrow \quad \texttt{\&}x$$

In case of `calloc`, let n denote the number of elements to be allocated and s denote the size of each element. We add a subgoal that asserts that s matches `sizeof(`t`)`. Let x be a new variable of an array type with elements of type t and size n.

$$(t\ \texttt{*})\ \texttt{calloc(}n\texttt{,}\ s\texttt{)} \quad \longrightarrow \quad \texttt{\&}x$$

```
{-1}  output == 0
[-2]  factor2 == in2
[-3]  factor1 == in1
--------------------------
{
  while(factor1 != 0)
  {
    if(factor1 & 1) output = output + factor2;

    factor1 = factor1 >> 1;
    factor2 = factor2 << 1;
    assert(output == in1 * in2 - factor1 * factor2);
  }
}
--------------------------
[1]  output == in1 * in2
```

Figure 2: Combination of Hoare Triple and Gentzen sequent

3.2.3 Proof Construction for ANSI-C Programs

Using Hoare's axioms, and given that loops have an upper run-time bound, the ANSI-C program can be transformed into a Boolean propositional formula completely automated by a proof strategy. Also given that the specification written in bit vector logic, the verification problem is within bit vector logic and therefore decideable. We also support arbitrary higher-order logic properties given as PVS specification. In this case, the proof has to be constructed manually.

The user interaction is done using a variant of the Gentzen sequent: In the Gentzen sequent, the verification problem is displayed as implication as follows:

$$\bigwedge_i a_i \implies \bigvee_j c_j$$

We combine the Gentzen sequent and the Hoare triple as follows: the precondition is displayed as conjunction, and the post-condition is displayed as disjunction:

$$\{\bigwedge_i a_i\} \quad \texttt{code} \quad \{\bigvee_j c_j\}$$

This allows using a consistent set of proof commands, such as `copy` and `delete` for the manipulation of sequents for both pure Gentzen sequents and Hoare triples. See figure 2 for an example.

3.3 PVS Language as Specification Language

The language of the theorem proving system PVS was chosen as common specification language. It is well-established in contrast to new approaches such as SAL [BGL+00]. As described above, the type check is time critical on large theories.

Restrictions We support the full set of language features of the PVS language with the following restrictions: For efficient parsing, we require a semicolon after each definition. This is optional in the original PVS language. We do not allow overloading the keyword `IF` (the parser is taken from [BBJ+02]). We do not resolve overloading ambiguities using the type of the expression, but rather only the type of arguments. Thus, the type of an expression only depends on the expression itself, never on its context.

We are using a state of the art microprocessor design as benchmark [BJ01, BBJ+02]. The microprocessor contains an out-of-order scheduler and a floating point unit. The design, specification and proof are done using PVS. Experiments show that the type check is time critical since it is part of the interaction. For large context, the type check using PVS takes up to thirty minutes. Our implementation takes less than four seconds for the same input. All times are obtained on a dual AMD Athlon with 1.5 GHZ and 3 GB of RAM. This result shows that our implementation is fast enough even for large designs that are too big for generic theorem provers.

4 Decision Procedures

4.1 Overview

Traditional theorem proving puts to much burden, i.e., manual work, on the verification engineer. By adding strong decision procedures as rule of inference, we can provide a very high degree of automation. Typically, the verification engineer splits the whole design into smaller parts or modules, which can then be verified automatically by automated decision procedures. The decision procedures that come with generic theorem provers are usually too slow for big decision problems. We therefore implement efficient decision procedures for subgoals that are large. Furthermore, most are optimized for specific applications.

The following decision procedures are implemented:

- For simplification, we have implemented a simple constant propagation and canonization algorithm. This is similar to the `simplify` rule in PVS.
- We implemented CMU SMV as decision procedure for smaller problems that include a model. Experiments showed that SMV is particularly useful for the verification of liveness properties of microprocessor control circuits.
- We implemented a strong SAT based decision procedure for bit vector decision problems. In the context of this project, the bit vector decision problems arise from the ANSI-C programs. We employ Chaff [MMZ$^+$01] as SAT checker. This decision procedure is described in more detail below.
- We integrated the Omega library [PW94] as decision procedure for Pressburger arithmetic.
- We implemented a variant of the Simplex algorithm for linear arithmetic on rational numbers.
- We implemented a rule based decision procedure using a SAT checker. This decision procedure is described in more detail below.

4.2 Proving Bit-vector Equations using SAT

Using Hoare style inference rules, we reduce the problem to Boolean predicates on the state of the program. It is therefore left to verify these equations. In case of ANSI-C programs and HDL specifications, these equations are bit-vector equations. We implemented a decision procedure for such equations that translates these equations into CNF. The translation is done the same way as done by BMC, i.e., by adding new variables. There is a set of alternative algorithms for bit vector decision problems [CMR97, MR98].

We implemented numerous operators, including shifting with variable distance and arithmetic operators including multiplication. We made experiments using the SAT checker *Chaff* [MMZ$^+$01].

The results are very promising unless nonlinear arithmetic is involved. Even large formulae are asserted within seconds, given that they do not contain nonlinear arithmetic. We experienced an exponential blowup on equations that include nonlinear arithmetic, as in the following example:

$$a * b \stackrel{!}{=} (a << 1) * (b >> 1) + \left\{ \begin{array}{lcl} a & : & b \& 1 \neq 0 \\ 0 & : & \text{otherwise} \end{array} \right.$$

This equation is the induction step of an ANSI-C program that does multiplication using left and right shifting (a has to be large enough to prevent an overflow during the left shift operation).

4.3 Proving Equations using Rules and SAT

Because of the limits of the approach described above, we also implemented a rule based decision procedure for bit vector equations. We convert the formula into a Boolean propositional formula by reducing all properties to equalities. These equalities are mapped to Boolean variables. The claim and the equality based rules are passed to the SAT checker Chaff. This allows even infinite data types such as natural numbers.

We have implemented rules for most common ANSI-C operators such as addition, multiplication, relational operators etc. For example, if the sequent contains an expression such as `a+b`, we add a commutativity constraint

$$a + b = b + a$$

In case of arbitrary functions f we do not have any information about, we just make them uninterpreted using congruence closure by adding

$$a = b \Longrightarrow f(a) = f(b)$$

as constraint. The algorithm queries all theorems and lemmas that have been verified so far and adds all appropriate theorems as constraint.

This certainly provides no completeness, but is able to decide many equations that arise from practical examples completely automated. The algorithm is very fast, and the generated CNF formulae typically only have a few thousand variables and are verified in less than a second by Chaff.

5 Conclusion and Future Work

We illustrated how to combine several completely different application specific design languages within one common higher-order logic theorem proving framework. We describe a tool implementing ANSI-C, Verilog, SMV, and PVS language.

Future work includes the addition of more design languages such as VHDL and object oriented languages like Java and C++, and more application specific decision procedures. In particular, for modular reasoning about languages with pointers or objects a decision procedure for a logic like separation logic is desirable.

References

[BBJ+02] Christoph Berg, Sven Beyer, Christian Jacobi, Daniel Kröning, and Dirk Leinenbach. Formal verification of the VAMP microprocessor (project status). In Witold Charatonik and Harald Ganzinger, editors, *Symposium on the Effectiveness of Logic in Computer Science (ELICS02),*, pages 31–36, 2002. Technical Report MPI-I-2002-2-007, Max-Planck-Institut für Informatik, Saarbruecken, Germany.

[Ber01] Sergey Berezin. The SyMP tool. `http://www.cs.cmu.edu/~modelcheck/symp.html`, 2001.

[Ber02] Sergey Berezin. *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, Carnegie Mellon University, January 2002.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[BJ01] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. In *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[Bou97] R. J. Boulton. A tool to support formal reasoning about computer languages. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 81–95, Enschede, The Netherlands, April 1997. Springer.

[BT00] P. Bertoli and P. Traverso. Design verification of a safety-critical embedded verifier. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[CGM86] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[CMR97] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1997.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC Research Report, 130 Lytton Ave., Palo Alto, 1998.

[FORS01] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. Ics: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Gor85] Michael J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In George J. Milne and P. A. Subrahmanyam, editors, *Proceedings of the 1985 Edinburgh Workshop on VLSI Design: Formal Aspects of VLSI Design*, pages 153–177, Edinburgh, Scotland, 1985. North Holland.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag.

[Int99] International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1999.

[KM96] Matt Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE Computer Society Press, 1996.

[Luc79] D. Luckham. Stanford Pascal verifier user manual. Technical Report STAN-CS-79-731, Stanford University Computer Science Department, March 1979.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[MR98] M. Oliver Möller and Harald Rueß. Solving bit-vector equations. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal methods in computer aided design: Second International Conference, FMCAD'98*, number 1522 in LNCS, pages 36–48. Springer-Verlag, 1998.

[ORS97] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.

[Pfe00] Holger Pfeifer. Formal verification of the TTP group membership algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Methods for Distributed System Development Proceedings of FORTE XIII / PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000. Kluwer Academic Publishers.

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In Charles B. Weinstock and John Rushby (eds.), editors, *Dependable Computing for Critical Applications 7*, volume 12 of *Dependable Computing and Fault-Tolerant Systems*, pages 207–226. IEEE Computer Society, January 1999.

[PW94] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[Rus97] John Rushby. Subtypes for specifications. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering–ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 4–19, Zurich, Switzerland, September 1997. Springer-Verlag.

[Rus00a] John Rushby. Theorem proving for verification. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modelling and Verification of Parallel Processes: MOVEP 2000*, number 2067 in Lecture Notes in Computer Science, pages 39–57, Nantes, France, June 2000. springer Verlag.

[Rus00b] David Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston;Dordrecht;London, 1991.

[V. 95] V. Pratt. Anatomy of the Pentium Bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, 1995.