Combination of Isabelle/HOL with Automatic Tools

Sergey Tverdyshev *

Saarland University, Germany
deru@wjpserver.cs.uni-sb.de

Abstract. We describe results and status of a sub project of the Verisoft [1] project. While the Verisoft project aims at verification of a complete computer system starting with hardware and up to user applications, the goal of our sub project is an efficient *hardware* verification.

We use the Isabelle theorem prover [2] as the major tool for hardware design and verification. Since many hardware verification problems can be efficiently solved by automatic tools, we combine Isabelle with model checkers and SAT solvers. This combination of tools speeds up verification of hardware and simplifies sharing of the results with verification of the whole computer system. To increase the range of problems which can be solved by external tools we implemented in Isabelle several algorithms for handling uninterpreted functions and data abstraction.

The resulting combination was applied to verify many different hardware circuits, automata, and processors.

In our project we use open source tools that are free for academical and commercial purposes.

1 Introduction

In large verification projects such as verification of a complete computer system the linking of verification results from different parts plays a major role. Specifying and proving all theorems within one environment, e.g. a higher order logic (HOL) theorem prover, makes linking a lot easier. Such a combination is also much safer because verification gaps, due to a manual transfer of the results from one system into another, are excluded. This was one of the motivations for this work.

In a long-term project Verisoft we are currently working on verification of a computer system starting with hardware, going through compiler, operating system kernel, operating system and up to end user applications. The main verification tool for all parts of the project is the Isabelle theorem prover for higher order logic. Because many hardware verification problems can be efficiently solved by external automatic tools, we combined Isabelle with the NuSMV model checker [3] and SAT solvers. In this paper we describe the result of the combination and demonstrate applications of this combination for hardware verification.

* Supported by The Verisoft Project under grant 01 IS C38 of the German Ministry for Education and Research (BMBF)

B. Gramlich (Ed.): FroCoS 2005, LNAI 3717, pp. 302–309, 2005.

2 Related Work

The most recent combination external tools into Isabelle was done through input-output automata [4]. In this work the user gives a model and manually defines its abstraction. Then an LTL model checker is used to prove temporal properties of the abstracted model and a μ-calculus model checker is used to check forward simulations between these two models. A drawback is that defining a suitable abstraction for a big model can be a very hard task. On the contrary, our approach does not have this disadvantage because an abstracted model is derived fully automatically.

An interesting ongoing work of L. Paulson's group [5] is the integration of the first oder theorem prover SPASS [6] into Isabelle. A highlight of the integration is that SPASS proofs can be converted into Isabelle proofs and then rechecked by Isabelle. In this approach the user does not have to trust an external tool, and soundness of the translation can be guaranteed. However, at the moment the integration is experimental and has only a very basic functionality.

The UCLID system [7] is another interesting tool that can handle big problems with great automation. It also has a lot of built-in features, e.g. handling of uninterpreted functions, efficient algorithms for term reduction. The UCLID system is mostly used for verification of invariants of a system (safety properties) but liveness properties (directly) are missing [7]. Even though the UCLID system is more powerful than our tool, our approach allows verification of liveness properties directly. Furthermore we believe that *complete* automatic abstraction, as we implemented, is more suitable for an automatic proof tool. Integration with Isabelle increases the domain of application of our tool but we have to pay for that with user's involvement in proof process. Last but not the least: UCLID is distributed as a close source system with a strict license.

The rest of the paper is organized as follows. In the Section 3 we present the used tools. Section 4 provides subset of the HOL we use to specify hardware. In Section 5 the main functionalities of the translation tool are presented. Section 6 reports results of applications of the resulting system to hardware verification.

3 Tools

The Theorem Prover. Isabelle [2] is a generic theorem prover that supports several object logics. We use Isabelle with its instantiation of HOL. We refer to it as Isabelle/HOL.

The Modelchecker. NuSMV [3] is a symbolic model checker for CTL and LTL properties. It can perform bounded model checking using an external SAT solver. We used NuSMV to verify temporal properties of our models and as an external BDD decision procedure.

SAT Solvers. We implemented an algorithm to convert given problems into propositional logic extended by uninterpreted functions and linear arithmetic. Using this algorithm we can easily bind almost any SAT solver.

We bind these tools to Isabelle/HOL through the Oracle interface. Translation of a problem from Isabelle/HOL into language of an external tool is done by the translation tool

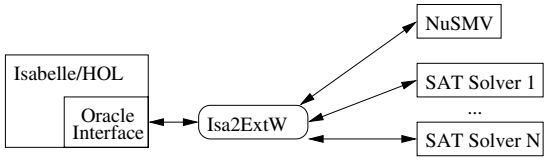

Fig. 1. Isabelle/HOL and External Tools

Isa2ExtW. We implemented Isa2ExtW as a decision procedure in Isabelle/HOL. An overview of the connection is shown in Figure 1. Isa2ExtW performs not only syntax translation but also several semantic transformations, e.g. data abstraction.

4 Subset of HOL for Hardware Design and Verification

Most of our verification problems can be split up into a number of smaller problems which can be solved automatically by one or the other external tool. However as none of these tools can solve our problem entirely, we still need Isabelle/HOL and Isa2ExtW as central instance. To allow translation of our theorems into the language of external tools we need to specify a suitable subset of Isabelle's HOL. Note that this subset has to be translatable into VHDL [8] in order to get synthesizable hardware.

4.1 Types

A fragment of the Isabelle/HOL language to be translated into external tools consists of expressions involving types which are finite. Examples of such types are scalar types, lists of constant length, records constructed from finite types and each other.

4.2 Subtyping

As we have already mentioned we are mostly interested in finite types. However, some infinite data types, namely their subtypes are interesting for us as well[1]. Since there are no subtypes in Isabelle/HOL, we have to define a mechanism for encoding subtype information into our models. For this purpose we created in Isabelle/HOL a library of predicate sets[2]. For the given predicate a *predicate set* defines the set of all elements satisfying this predicate. By means of predicate sets we reduce types in our models to desired finite subtypes. This information is added to the set of assumptions of a theorem we want to prove by external tools. A drawback is that the model will be correct with respect to the specified subtypes. However, the property will be proved automatically. This idea can be easily described by the following example:

Bit vectors in Isabelle/HOL are represented as lists of bits, possibly of infinite length. However, in general we are interested in the lists of a constant length only to allow synthesis of real hardware. Thus, additional information about the length of bit vectors has to be provided. We illustrate our approach on a model of a shifter: suppose

[1] Using data abstraction we can handle infinite types (see Section 5.2).

[2] Full description of the library can be found at `http://busserver.cs.uni-sb.de`

we want to perform an equality test of two functions sh_impl and sh_spec which are implementation and specification of the shifter respectively. The desired lemma could be formulated in Isabelle/HOL as follows:

```
∀ op. ∀ d. ∀ r. ∀ a.
sh_impl(op, d, r, a) = sh_spec(op, d, r, a)
```

This theorem would state that for all bit vectors to be shifted (op), for all shift distances (d), and for all boolean flags r, a the implementation and the specification return the same result. In this theorem the bit vectors op and d can have arbitrary length. However, for our purpose it is enough to show the correctness for op of length 32 and for d of length 5. To describe these subtypes we use parametrized predicate set bv_n from our library. For the given natural number n it defines the set of all bit vectors of the length n. We add an assumption that inputs are from the desired subsets and the theorem in Isabelle/HOL is formulated as follows:

```
∀ op ∈ bv_n(32). ∀ d ∈ bv_n(5). ∀ r. ∀a.
sh_impl(op, d, r, a) = sh_spec(op, d, r, a)
```

This goal is easily discharged by an external tool through a call of Isa2ExtW as a proof method. An advantage of this approach is that the original model is not influenced by any subtyping information, and can be easily reused for other goals. In a similar fashion we handle arrays, records etc., i.e. for each of them we defined a predicate set.

4.3 Operators

There are several operators which are substituted by analogous operators in a target external tool (e.g. boolean connectives) or interpreted by Isa2ExtW (e.g. basic operations on lists as head, tail). The list of such operators can be roughly described as follows: boolean connectives, bit operations, linear arithmetic operators, basic operations on lists, and update of variables of function types.

4.4 Functions

A drawback of the absence of subtypes in Isabelle/HOL is that we can not restrict function inputs to desired subtypes. There are two solutions (i) handle cases of undesirable inputs in the definition of a function; (ii) formally guarantee absence of that inputs.

The first solution allows using functions with any input and functions will return expected results. However, a drawback is that extra handling can be inconvertible into external tools or into VHDL, e.g. in physical sense the behaviour of a circuit for an empty bit vector is unclear. With the second solution the user does not handle undesired input. In this case for some inputs a circuit will have undesired but well-defined behaviour. The absence of undesirable inputs is guaranteed by the use of a function, namely we prove properties about a model for a clearly defined set of inputs (see Section 4.2).

Non-recursive functions will be translated into external tools as they are, i.e. using their definitions. The returned type of a function is computed on the fly by Isa2ExtW and must be one of the supported types, otherwise an error will be raised.

Recursive Functions. For translation of recursive definitions into external tools we support recursions on natural numbers and on the length of a list. Based on earlier experience in hardware design and verification [9,10] these two types of recursion are enough to describe all constructions we need to build and verify a processor. Our translation algorithm unrolls a recursive definition in a set of non-recursive definitions. Initial value for recursion is taken from the current input of a function. Unrolling rules are taken from the original recursive definition.

Uninterpreted Functions. Sometimes it is very useful to abstract a functional unit as an uninterpreted function, e.g. while verifying datapaths of a processor. We defined a simple mechanism to specify uninterpreted functions in our models. To force Isa2ExtW to translate a function as an uninterpreted function, the user has to specify input/output interfaces of the function. It includes the name of the function and predicate sets for inputs and outputs of the function, e.g. "$\forall$ `a` $\in$ `bv_n(32). foo(a)` $\in$ `bv_n(5)`". For the given example Isa2ExtW will not look up the definition of function `foo`. The tool will replace it by an uninterpreted function `foo` which takes a bit vector of length 32 and returns a bit vector of length 5. The restriction is that such a function can not be updated. The usage of such a function is controlled by Isa2ExtW.

5 Isa2ExtW

In this section we describe some functionalities of Isa2ExtW.

5.1 Uninterpreted Functions

When verifying processors big storages such as general purpose registers or memories have to be modelled. It is convenient to represent memories as uninterpreted functions. In our library we have a parametrized predicate set which for the two given predicate sets `A` and `B` returns the set of functions `A` $\rightarrow$ `B`. The only restriction is that the input predicate sets themselves can not be sets of functions. Direct translation of such types into the mentioned external decision procedures is unpractical because of the size. To avoid this limitation we implemented a simple but efficient algorithm for elimination of variables of function types. The idea was taken from [11]. This algorithm consists of three parts: (i) representation of the variables of a function type (memories) as uninterpreted functions, (ii) elimination of memories updates by `if-then-else` expressions and (iii) elimination of applications of uninterpreted functions by nested `if-then-else` expressions. For more details we refer the reader to [11].

5.2 Data Abstraction

To counter act the state explosion problem we implemented a data abstraction algorithm based on symmetry reduction [12]. Implementations of Dill's idea usually include a type constructor `scalar type`, e.g. [7,13]. This constructor defines an abstract type on which the symmetry reduction can be done. In contrast, our implementation works *completely automatically*. For the given model it finds, on the fly, all variables which can be abstracted, exploiting data symmetry of the model. The new reduced model is constructed *automatically* as well.

Often implementations of symmetry reduction can not handle constants, e.g. when we compare a variable with a constant we can not apply symmetry reduction to that variable (a drawback of SMV [13]). We solved the problem by introducing a new unique symbolic value for every constant and adding it to the definition of abstracted type. We exploit advantages of this feature in processor verification. In our models of processors we have bit vectors of length 32 and we use a few constants of this type, e.g. 32 zeroes. Application of our symmetry reduction algorithm reduces the state space of such a variable from 2^{32} just to $number\ of\ variables\ +\ number\ of\ constants$.

Another advantage of our implementation is that uninterpreted functions do not break symmetry of the model. Since the result of the application of an uninterpreted function does not depend on input, then, from the symmetry point of view, such application splits the model into two independent parts. These are: arguments of function and the result of application. We apply the abstraction algorithm independently on both parts. The effect is that even if the result cannot be abstracted then arguments may be abstracted and vice versa. The idea behind this is a combination of algorithms for elimination of uninterpreted functions and symmetry reduction. Consider the following example. In equality (1) `f` is supposed to be an uninterpreted function. Bryant et. al. [11] proved that (2) holds. After application of symmetry reduction[3] we come to (3). Where variables `da′`, `db′` and `a′`, `b′` have abstracted types. By applying the Bryant theorem in reverse direction we can conclude that there exists an uninterpreted function `f′` with the same domain/range type as the type of `a′`/`da′` (4).

```
f(a) = f(b)                                                    (1)
f(a) = f(b)     ↔ da = (if a = b then da else db)              (2)
                  da' = (if a' = b' then da' else db')         (3)
f'(a') = f'(b')                                                (4)
```

Our implementation abstracts applications of uninterpreted functions in only one step. This feature *increases significantly* the range of models where the abstraction algorithm can succeed. It allows hiding symmetry-breaking functions by declaring them as uninterpreted functions. It leads not only to a reduced state space of the model but also to a reduced number of terms the model consists of. These all will result in faster verification.

5.3 SAT Solving

All problems which are specified according to the rules presented above can be converted to propositional logic and solved by an external SAT solver. If the model contains linear arithmetic, the external tool should have a decision procedure for it. Ourself we do not require from SAT solvers to support uninterpreted functions, since we can handle them. However, native support of them may speed up the verification process[4].

5.4 Model Checking

Models to be model-checked have to be represented as finite state machines (FSM). To define such a machine the user has to provide a next-state function, a set of states and

[3] Without support of uninterpreted functions.

[4] E.g. by usage of ModuSAT [14] after solving some performance problems.

a set of initial states. To express temporal properties of FSM's we specified CTL and LTL in Isabelle/HOL. FSM and CTL/LTL are combined in simple interface functions $\models_{LTL}$ and $\models_{CTL}$:

for LTL: `(States, Init, NSF)` $\models_{LTL}$ `LTL_formula`
for CTL: `(States, Init, NSF)` $\models_{CTL}$ `CTL_formula`,

where `States` is a predicate set as described earlier, `Init` is a predicate on the state type and `NSF` is a next state function. `LTL_formula` and `CTL_formula` are properties in LTL and CTL respectively. The user has to take care that the `Init` predicate holds at least for one state from `States`. Otherwise the results of verification can be nonsense.

Optionally the user can turn off the algorithm for elimination of applications of uninterpreted functions. In this case such a function will be represented in NuSMV as an additional state variable. The type of that variable is an array of the domain type of the function. The size of the array is computed based on the type of the range of the function. We do not put any transition constraints on the behaviour of that variable and it can change non-deterministically along the time. It captures behaviour of an uninterpreted function. If the abstraction algorithm does not succeed then application of NuSMV may be inefficient because of the size of the state space. However, usually[5] the abstraction algorithm succeeds and we can use model checking technique even for *models containing uninterpreted functions and memories*.

6 Results and Future Work

We used the Isa2ExtW tool for verification of many combinational circuits. For example we automatically verified all hardware components of our processors such as shifters, decoders, encoders, parallel prefix or-operation etc. by an external SAT solver and NuSMV. We automatically verified a simple sequential DLX [15] processor using uninterpreted functions.

Verification of a pipelined DLX processor featuring 3-stage forwarding and stalling was a more challenging task. We took proofs of the processor from [16,10] as a base. We built models of specification and implementation in Isabelle/HOL. Then the proof was started in Isabelle/HOL. We interactively distinguished major cases of the proof and got subgoals which were discharged by a SAT solver. Uninterpreted functions were heavily used to simplify verification of datapaths of the processor. Through instantiation of uninterpreted functions by concrete verified functions in verified datapaths, the correctness proof could be completed. The liveness of the processor was verified completely automatically. Applying our method as front-end to NuSMV, we were even able to model-check models with big storages (e.g. data memory, general purpose registers) and uninterpreted functions (e.g. arithmetic-logical unit). In this manner we got a completely verified processor on the gate level. Our method significantly improves verification of a similar processor in [10] because the user is no longer required to prove "simple" subgoals manually.

[5] For the models we verified, see section 6.

We incorporated the NuSMV model checker to verify automata for the memory management unit [17] and automata for a replacing policy in a cache system [9].

In order to get synthesizable hardware we aim towards a tool which will translate our hardware specifications from Isabelle/HOL into VHDL. The next benchmark for our method is verification of an out-of-order DLX processor featuring Tomasulo algorithm, precise interrupts and memory management unit [16,10,9].

References

1. The Verisoft Consortium: The Verisoft Project. `http://www.verisoft.de/` (2003)
2. Paulson, L.C.: Isabelle - A generic theorem prover. LNCS **828** (1994)
3. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Marco Pistore, M.R., Sebastiani, R., Tacchella, A.: NuSMV 2: An open source tool for symbolic model checking. In: CAV '02, Springer-Verlag (2002) 359–364
4. Müller, O.: A Verification Environment for I/O Automata Based on Formalized Meta-Theory. PhD thesis, Techn. Univ. Munich (1998)
5. Larry Paulson: Larry Paulson's home page. (`http://www.cl.cam.ac.uk/users/lcp/`)
6. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic', D.: SPASS version 2.0. In Voronkov, A., ed.: Automated deduction, CADE-18. Volume 2392 of Lecture Notes in Artificial Intelligence., Kopenhagen, Denmark, Springer (2002) 275–279
7. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and verification of out-of-order microprocessors in uclid. In: FMCAD '02, London, UK, Springer-Verlag (2002) 142–159
8. Ashenden, P.J.: The Designer's Guide to VHDL. Morgan Kaufmann Publishers Inc. (1999)
9. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Putting it all together formal verification of the VAMP, to appear in STTT, Springer-Verlag (2005)
10. Kröning, D.: Formal Verification of Pipelined Microprocessors. PhD thesis, Saarland University, Computer Science Department (2001)
11. Bryant, R.E., German, S.M., Velev, M.N.: Microprocessor verification using efficient decision procedures for a logic of equality with uninterpreted functions. In: TABLEAUX '99, Springer-Verlag (1999) 1–13
12. Ip, C.N., Dill, D.L.: Better verification through symmetry. Form. Methods Syst. Des. **9** (1996) 41–75
13. McMillan, K.L.: The SMV language. Technical report, Berkeley Labs (1999)
14. Prevosto, V.: ModuProve Developer and User Manual. Max-Planck Institut für Informatik – Verisoft Project. (2005)
15. Patterson, D.A., Hennessy, J.L.: Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc. (1995)
16. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer-Verlag New York, Inc. (2000)
17. Dalinger, I., Hillebrand, M., Paul, W.J.: On the verification of memory management mechanisms. Technical report, Saarland University (2005)