A Heirarchical Formal Verification Tool for Simple HDLs

Annette Bunker, Michael D. Jones, Trent N. Larson and Phillip J. Windley*
{bunker, jones, larson, windley}@cs.byu.edu
Computer Science Department, Brigham Young University, Provo UT 84602, USA

Abstract

We describe a tool to aid in the hierarchical verification of hardware components. This tool encourages hierarchical verification by using abstract thoeries and predicate types to ensure that an implementation will be usable whenever its specification is usable. Implemented devices are described in the BOLT HDL and verfied using the HOL proof assistant.

Useful hardware components are complex and often designed by more than just one person building on existing components. Hardware verification on the other hand is typically performed by one person building everything by hand-even if other models exist. The wide variety of models and techniques used by verification practitioners make the resuse of prior work difficult and unatural. If design proceeded like verification currently does, designers would always work alone and never reuse their own or anyone else's work. This mismatch in methodologies limits the complexity of devices that can be verified using formal methods.

We address the mismatch in methodologies by creating tools to support hierarchical verification and component sharing among collaborating researchers. Our approach uses abstract theories and predicate types to build libraries of verified components. The information is organized to ensure that an implementation is usable whenever its specification is. We use the BOLT hardware description language to describe specific implementations.

The next section contains an overview of related work followed by a section containing a short tutorial on abstract theories and predicate types and their use in heirarchical hardware

*This work was sponsored by the National Science Foundation under NSF grant MIP–9412581

verification. Section three covers the tools we created in the HOL system to allow the use of abstract theories and predicate types in HOL. Section four presents a more detailed description of the use and contents of the library itself. Section five contains several examples of the library in use. We conclude with a concise statement of our results and plans for future work.

1 Related Work

Our work ties together many ideas from the fields of subtyping using predicate types, abstract theories in automated theorem proving, embedding HDLs in theorem provers and secure sharing of the theorems. In this section, we discuss related work that proved influential in our own.

Predicate subtypes are the subject of a large and varied research community. [?] discuss the notion of type in great detail and [?] builds on this notion with his discussion of typechecking, type inferencing, and subtyping. The PVS theorem prover contains a widely known and quite succesfull predicate subtyping system [?] which was very influential. While the PVS type system is precisely what we need, PVS' closed system and lack of a meta-language make it unusable for our purposes. However, Nuprl and UmptyProver also have predicate types built-in but we didn't use them because what?

Using HDLs as input languages to formal verification tools in theorem provers has been widely explored in the HOL research community. [?] describes the authors' experiences with embedding several hardware description languages in HOL, including ELLA, Silage, and VHDL.

[?] describes our recent efforts to create a general HDL embedding framework called `GHDL_EVAL` for reasoning about simple heirarchical netlist description languages. The `GHDL_EVAL` project resulted in a deep embedding of an abstract syntax which gains the ability to reason about languages in general but is slow and awkward for large examples. [?] describes past efforts to integrate BOLT into a hardware verification environment. A tool to automaticaly generate language embeddings in HOL is presented in [?] and used in [?] to embed a significant subset of VHDL.

[?] build a fairly sophisticated proof heirarchicy using Nuprl's predicate and dependent subtyping system along with a proof tactic [?] for verifying components built using the tool. This work presented promising early results but has not been tied together in a usefull way.

Sharing theorems across an insecure network presents trustworthiness problems. Grundy addresses those problems using PGP encryption to digitaly sign and decrypt theorems [?]. Grundy's work is carefully thought out and interesting but not implemented for the HOL system.

2 Foundational Ideas

In this section, the notions of predicate types and abstract theories are discussed along with their uses in hardware verification. As these ideas may be unfamiliar to researchers in other fields, we present a brief discussion of each highlighting their application to hardware verification.

2.1 Predicate Types

Modern programming (and theorem proving) uses simple types to constrain the values a variable may contain. For example, the type *integer* contrains variables of type *integer* to contain only integer values. In most cases, type languages are kept simple to maintain decidable type checking algorithms. However, simple type systems have limited expressability and can only be used to partialy specify a given device. A device's intended behavior is only partialy specified in a partial specification of correctness. For example, a partial full-adder specification might state that a full-adder takes 3 signals and retuns 2 signals but does not specify the relationship between the input and output signals. Although the partial specification is not expressive, it can be used to create a type containing full-adders.

In our work, we use types as a complete, rather than partial, specification of correctness. A complete specification of correctness describes every facet of the intended behavior and can not be used to define a type in simple type systems. A full-adder complete specification would state not only that a full-adder takes three inputs and returns two outputs but would also state that the sum of the bit-values of the inputs is equal to the sum of output plus the carry out times two.

To use types as a complete specification of correctness, we expand the type language to allow the use of arbitrary predicates to describe type spaces. In the full-adder example, the specification would be expressed as a predicate and used to define a full-adder type populated with devices that behaved according to the specification.

The expressiveness of predicate types comes at the cost of decidability. Type checking algorithms in systems using predicate types return a set of conditions which imply the type-correctness of the expression being checked. Type-checking a full-adder would generate a condition stating that the adder behaves according to the specification predicate used to define the type. The user would then be left to prove that this condition holds before the expression could be declared type-correct.

Assuming the type checking system is safe, predicate types as a total specification of correctness ensures that a given implementation of a given type will be usable whenever the predicate defining the type is usable.

2.2 Trustworthy Theorem Sharing

3 Theorem Prover Infrastructure Development

Existing HOL implementations do not support hardware verification as discussed in the last section. In this section, we briefly discuss tools developed for use in the HOL system to aid in heirarchical verificaiton based on predicate types, abstract theories and the BOLT hardware description language.

3.1 BOLT Translator

To use BOLT specifications in HOL we first translate BOLT expresions into HOL terms. We built a shallow embedding of BOLT in HOL. In a shallow embedding the translation is described outside the framework of the logic. logic. Shallow embeddings preclude reasoning about the language in general but are (in our experience) more efficient, easier to develop and easier to use than deep embeddings.

In our BOLT shallow embedding, the translation functions attempt to infer as much information as possible from the BOLT description. This prevents the user from entering redundant information. For example, the translation tools use set intersection and difference to calculate the internal wires from the BOLT description rather than having the user type them explicitly. The internal wires are then hidden using existential quantification in the HOL term denoting the device.

3.2 Predicate Types

In order to get the expressive power of predicate types in HOL we implemented a predicate type system outside the HOL logic that behaves like the PVS type system–with some minor extensions. Implementing predicate types inside the HOL logic entails making significant modifications to the HOL logic and was clearly beyond the scope of our project. By building predicate types outside the HOL logic, we avoid the complexity of changing the HOL term langauge and gain the expressiveness of predicate types.

In our enhanced HOL type system, the user associates predicates with logical constants which restrict the range of values to which a constant may be applied. The system then generates correctness conditions from uses of restricted definitions in terms. Correctness conditions are either proven automaticaly using a built in tactic or returned to be proven by the user. A tool for saving restrictions and unproven conditions with the current theory is also provided. The built in tactic for automaticaly proving correctness conditions can be extended by the user.

For example, the restricted type system can be used to define a MOD function which takes a non-zero second parameter. Suppose the MOD function were defined as follos:

```
! a b . mod a b = (if (a > b) then mod (a - b) else a)
```

If $b \leq 0$ then the recursion will never terminate so we restrict b to be greater than 0 using the `new_type_restriction` function.

```
new_type_restriction (``mod'',
      **`==:num== -> --λ x . x > 0-- -> ==:num==`**)
```

In this example, `mod` is now restricted to operate on ...

Using MOD in a HOL term now results in a restriction correctness condition which must be proven by the user. The term

```
mod 3 5 = 3
```

results in the condition

```
5 > 0
```

which is easily proven using existing HOL arithmetic procedures.

3.3 Abstract Theories

4 Component Library

The library contains specifications and implementations. SML functions generate library objects from specificaion predicates and BOLT descriptions of implementations. Libraries can be loaded as HOL parent theories or generated using files of executable SML code.

Specifications in the library are comprised of type definitions, type constructors, type destructors and type restrictions on the constructors. The type definition is done in HOL using the inductive definition package developed by Melham [?] and contains the external, structural specification of the device. Type constructors and destructors are used to insert and extract implementations from the type. The type constructor restriction captures the behavioral specification of the device and is used to keep the type space pure by only allowing well-behaved devices to be injected into the type space.

There are two kinds of implementations in the library: primitive implementations and parameterized implementations. In primitive implementations, the device is defined using only the specification with no reference to how the device will actualy be built. Parameterized implementations contain the inter-connections of the subcomponents which will (hopefully) result in the behavior specificied by the specification. Inside the definition, subcomponent parameters are extracted from their datatype and used in the defintion of the device.

Our approach to generating objects in the library of components is rather simplistic. Rather than create an object description language and an interpreter to translate descriptions back to HOL terms, we decide to represent library objects with executable SML code. In this approach, the user calls an SML function to create an object in the library. The SML function then generates the SML code to create, execute and store the object. The SML function used by the user is more abstract than the code generated by the library itself. We don't gain any more functionality in this approach we just make it easier on the user.

For example, to create a new specification for a hardware device the user makes a call to `create_hw_type` with the name of the specification, the names and types of the inputs and outputs and an HOL term denoting the intended behavior in abstract terms. The automated library creation tools then use the information supplied by the user to create a datatype representing the device, type constructors and type destructors along with type restrictions on the type constructor that limit the constructor to apply only to well-behavied devices. More concretely, a call to

```
create_hardware_type
{name = ``fulladder'',
 library = ``walter'',
 output_tys = [==`:bool`==,==`:bool`==],
 output_vars = ["b3", "co"],
 input_tys =  [==`:bool`==,==`:bool`==,==`:bool`==],
 input_vars = ["b1", "b2", "ci"],
 spec = (--`bv b1 + bv b2 + bv ci =
    bv b3 + (2 * (bv co))`--)};
```

creates a fulladder type along with

Libraries of devices are stored and retrieved in two ways: by using existing theory storage and retrieval systems and by using executable files of SML code.

- In the first scenario, the HOL mechanism for storing the definitions made and theorems proved is used along with the restricted types mechanism for storing restrictions and unproven RCCs. The saved theory and type information is then loaded as an ancestor theory in the curent HOL session using the HOL new_parent and add_to_SML functions.
- In the second scenario, the executable SML code generated to create library objects is simply written to a file. This file can then be loaded in an HOL session to recreate the objects in the current theory using the built-in use command in SML.

5 Examples

Several examples are presented which demonstrate heirarchical verificaiton using the library infrastructure and illustrate how the library tools are used. The following example contains the specification and creation of primitive gates which are then used in the creation of two half-adder implementations which are then combined to create a full-adder.

5.1 Primitive Gates

Several calls to `create_hardware_type` creates the library objects representing several simple devices that will be used as primitives in this example. The following SML function application

```
create_hardware_type
{name = "or_2",
 library = "walter",
 output_tys = [==`:bool`==],
 output_vars = ["c"],
 input_tys =  [==`:bool`==,==`:bool`==],
 input_vars = ["a", "b"],
 spec = (--`c = (a  b)`--)};
```

creates the library object representing the specification for a two input or gate. Similar function applications are used to create specifications for two input `and` , `xor` , `nor` and `nand` gates.

The library tools are then used to create primitive implementations of these specifications. Primitive implementations contain no structural information and are simply devices assumed to behave according to the specification.

```
create_EZ_primitive library = "walter",
     name = "or_2_prim",
     spec = or_2_spec;
```

creates a primitive library object named `or_2_prim` which is assumed to meet the `or_2_spec` and has type `or_2_ty`.

5.2 Adders

Once the library of primitive devices is created, they can be loaded and used to create half-adder implementations. In this example, the primitive library theory is made an ancestor of the current theory and loaded using the `add_to_sml` Command.

The first step is to create a library object specifying the half-adder behavior and defining the half-adder type space. SML function `create_hardware_type` creates the requisite HOL definitions, types and type restrictions then saves them in a library file of SML code. Loading the library file regenerates the library objects in the current HOL session.

```
create_hardware_type
{name = "fulladder",
 library = "walter",
 output_tys = [==`:bool`==,==`:bool`==],
 output_vars = ["b3", "co"],
 input_tys =  [==`:bool`==,==`:bool`==,==`:bool`==],
 input_vars = ["b1", "b2", "ci"],
 spec = (--`bv b1 + bv b2 + bv ci =
    bv b3 + (2 * (bv co))`--)};
```

The first implementation will be based on `xor` , `and` and `or` gates.

```
create_bolt_module
{library = "walter",
 devs = [("xor_2_prim", xor_2), ("and_2_prim", and_2), ("or_2_prim", or_2)],
 spec = fulladder_spec,
 bolt_desc = (  `MODULE b3 co .fulladder_2 b1 b2 ci;
         BEGIN
 p  .xor_2_prim b1 b2;
 b3 .xor_2_prim p ci;
 q  .and_2_prim b1 b2;
 r  .and_2_prim p ci;
 co .or_2_prim  q r;
 END;`  )}
```

Function `creae_bolt_module` attempts to inject this implementation into the half adder type space. Doing so results in a type restriction correctness condition requiring that the implementation implies the specification. This particular RCC is easily proven using the `ABS_HW_VERF_TAC` tactic. The second implementation is built only from `nand` gates.

```
create_bolt_module
{library = "walter",
 devs = [("nand_2", nand_2_spec)],
 spec = fulladder_spec,
 bolt_desc = (  `MODULE b3 co .fulladder_32 b1 b2 ci;
         BEGIN
 p  .nand_2 b1 b2;
 q  .nand_2 b1 p;
 r  .nand_2 b2 p;
 s  .nand_2 q r;
 t  .nand_2 s ci;
 v  .nand_2 t ci;
 u  .nand_2 t s;
 b3 .nand_2 u v;
 co .nand_2 t p;
 END;`  )};
```

Verifying (or type checking) the second implementation procedes as in the first.

The two full adder impplementations can then be instantiated with their theory parameters and used interchangably. We illustrate this by building a two bit adder using a full adder and an or gate. The specification is as follows:

```
someday, a specification will appear here.
```

The implementation is built using a half-adder and an or gate wired together as shown:

```
someday, an implementation in BOLT will appear here.
```

Because the 2-bit adder is built and verified using the full adder specifications (rather than a specific implementation) either previous full adder implementation can be used to instantiate the 2-bit adder.

The 2-bit adder can then be used any where a 2-bit adder is required regardless of which full adder implementation is used to build the 2-bit adder.

6 Results and Future Work

Our work resulted in tools for creating libraries of hardware devices built using abstract theories and organized in predicate type spaces. Abstract theories allow heirarchical techniques to be applied to hardware verification. Predicate types organize theory parameter information concisely in a way that can be heavily automated. We have used the library tools to create a library of adders built from a library of primitive devices. Modular compents in the library use subcomponent implementations interchangeably.

We are currently working on building larger verification examples using our library tools. We are also exapanding the scope of our verification to include othe HDLs and problem domains (such as floating-point arithmatic algorithms).

One of our remaining goals is to make devices construced with our tools available on the WWW. There are a few reasons for this:

- The www is a common medium for accessing information. Any method we have for sharing information via WWW pages can be adapted by almost anyone else for use in their organization.
- The www is distributed worldwide, so researchers in any parts of the world may collaborate. All this requires is that they share the initial specifications.
- In addition, since our tool is made to help with distributed work, the web is an ideal place for making up-to-date information available to fellow researchers.

In the current library, abstract theories are used implicitly rather than explicitly. A package exists for creating and using abstract theories but we find that our use of abstract theories is simple enough to obviate the need for this tool. In future work, we intend to make more extensive use of abstract theories.

There are problems to overcome when dealing with the web. The main problem is trustworthiness. If we maintain a library for researchers at our university in which anyone can upload results, how do other users authenticate that the particular proof someone submitted was actually done? How can we check that the person who claims authorship actually did the proof? ... what else?

We'd like to try our approach to prove theorems in other domains as well. Someone has suggested we try to verify properties about floating point numbers, and we think our tool may help the proof effort by structuring it appropriately.

We would like to make our tool handle different HDLs and primitive libraries. We have built in the ability to change the HDL and the primitive components that developers use. We want to see exactly how easy this is to do, and provide support for as many HDLs as possible. We'd also like to make it simple to use whatever library of primitive components is desired. ..... expand that?

Finally, there are many parts of our tool that can be optimized. Verifications are currently done with a case split on the possible inputs to a device, and even for small combinations of devices (especially sequential devices), the cases can be incredibly large. Also, the meaning of the implemented circuit (translated from the HDL) is represented in our theorem prover with an operational semantics. This basically means that we execute the implementation to check it (thus the case split). There may be better ways to represent circuits in our theorem prover, while still automating the translation from the HDL.

Oh then there's iterated hardware devices built and verified using dependent types. We should mention that.