Analysis of Component-Based Systems – An Automated Theorem Proving Approach [*]

Murali Rangarajan
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418.
mrangara@htc.honeywell.com

Perry Alexander
EECS Dept., Info. and Telecomm. Tech. Center,
The University of Kansas
2291 Irving Hill Rd, Lawrence, KS 66044-7321.
alex@ittc.ukans.edu

ABSTRACT

As systems become increasingly complex, there is an increasing thrust towards designing systems at the requirements level. This approach enables the analysis of various system properties such as functional correctness, constraint satisfaction, et cetera at a very early stage in systems development, thus enabling faster design of systems with fewer design flaws. Analyses can be performed at various levels of rigor. For mission critical systems, analysis using formal techniques is highly preferable as it provides the highest level of rigor. But the problem with using formal analysis techniques is that they are either intractable for large designs, or require highly specialized knowledge possessed by a few select people. This prevents the majority of the design population from using such formal analyses in their design process. In this paper, we describe our approach for analysis of component-based systems for functional correctness using theorem proving techniques. The components and the design are specified using the VSPEC specification language. The model is translated into an equivalent model in the PVS specification language, and various correctness properties are automatically extracted from the model and their proofs are proofs are automatically attempted using specialized, dynamically generated, proof macros. Results of applying our technique to various modeling problems are provided, and the results are discussed.

1. INTRODUCTION

Hardware and software systems are becoming increasingly complex. Hardware systems consisting of millions of transistors and software systems requiring hundreds of thousands of lines of code are now commonplace. Along with the complexity of systems, methodologies have evolved to handle the complexity. While these methodologies provide ease of design, even for large systems, they do not have sufficient integrated support for analyzing correctness of designs. For example, top-down and bottom-up hierarchical design strategies are widely used to help design large systems. These by themselves do not ensure correctness of design. What is needed is some mechanism for analyzing functional correctness of systems during the design process.

*Support for this work was provided in part by the CEENSS Technology Program, contract number F33615-93-C-4304, and the RASSP Technology Program, contract number F33615-93-C-1316.

In recent times, a number of new techniques and methodologies have been proposed to handle systems design complexity. These involve specifying requirements in a Requirements Specification Language and analyzing specifications to arrive at the correctness of designs. A number of semi-formal and formal analyses have been proposed. Semi-formal methods and formal methods employing techniques such as model-checking are easy to apply since a high degree of automation can be achieved using these techniques. But semi-formal methods do not provide the requisite rigor to be applicable for safety/mission critical systems. Model-checking based analyses are restricted by the problem of state-space explosion (even after the use of state-folding), especially at the high levels of abstraction frequently encountered at the design phases. Moreover, compositional analysis using model-checking is not straightforward. Formal methods employing theorem proving have a wide range of applicability and can provide the mathematical rigor required for analyzing safety/mission critical systems. But they are difficult to apply as they require in-depth knowledge of formal notations and theorem proving techniques.

Application of theorem proving for analyzing designs could, according to the Formal Methods community, have a great impact in the industry. However, modern designs are large and complex, and formal analysis of such designs directly is not practical. Therefore, a simplified model is analyzed for correctness, from which the correctness of the original model is assumed. But this need not be always true. Theorem proving has the capability to directly analyze large and complex designs. Such direct analysis provides more accurate information about the correctness of designs. Thereby, use of theorem proving techniques can lead to better designs.

Theorem proving is not currently practical for a number of reasons. The first problem is identifying what needs to be proved. This is a deceivingly complex task. The second problem is that of specifying what needs to be proved in some theorem proving language. The complex and mathematical nature of such languages makes this a daunting task for most designers. The third problem is that of modeling the domain in the same specification language as that of the theorems. The final problem is that of proving the theorems once they have been specified in the language of a theorem prover. The proof process varies wildly with the theorem being proved, the design being verified, and the underlying models on which the theorem is based.

The purpose of this work is to devise a mechanism for automated analysis of component-based designs for common classes of errors. We use the VSPEC specification language to demonstrate our approach. From VSPEC specifications, equivalent PVS theories are automatically generated. The generated theories contain verification obligations (as theorems) for interfaces and interconnections. Automated proof assistance is provided for the verification activities. This approach was applied to a variety of example problems. The results (presented in section 4) showed that it is possible to generate PVS models corresponding to VSPEC specifications, and that properties about interfaces and interconnections can be generated as theorems. They also showed that it is possible to automatically generate proof scripts for certain categories of models such that the generated theorems can be automatically proved by the PVS theorem prover.

Since the goal was to automate the proof process, a number of simplification strategies were adopted. First, the semantics of VSPEC was kept simple by removing all unnecessary extensions, while at the same time, ensuring the validity of the resulting semantics. Next, the properties to be analyzed were generated from at-most two levels in the design hierarchy of the system. This results in much simpler properties to prove than if we were to consider the design as a whole. Recursive application of this technique to all the levels in the design provides information about the overall correctness of the design. Finally, the proofs themselves are generated as sequences of LISP commands, the same format used by PVS. This enables the proofs steps to be executed automatically to check whether a certain property is applicable to a design. It is to be noted here that the proof macros only increase the possibility of a proof being completed, but they do not guarantee it. Whether the proof is actually completed or not depends upon a number of factors such as the complexity of the design, complexity of the specifications, the property being analyzed, etc.

2. VSPEC

VSPEC is a requirements specification language for VHDL. VSPEC was originally designed as a Larch Interface Language for VHDL. Therefore, VSPEC borrows a number of features from VHDL. The VHDL entities, architectures, and packages are directly used by VSPEC. The information provided by VSPEC is specified in a `specification` construct. VSPEC `specifications` are similar to VHDL architectures in that they provide additional information about an existing entity. VSPEC's declarative specification style complements the traditional VHDL operational style. Together, VSPEC and VHDL support modeling from requirements acquisition through verification and synthesis.

As a working example, a VSPEC description of a sorting component is shown in Figure 1. The entity `sort` is identical to the VHDL entity construct. This provides the interface for the VSPEC specifications. The package construct is also similar to that of VHDL, with the exception of the keyword `mutable`. This type specifier has been added in VSPEC to enable the designer to specify complex types without giving any particular implementation.

The module `sort_spec` constitutes a VSPEC specification of the `sort` entity. The `sensitive to` clause is similar to sensitivity lists and the `wait` statement in VHDL – it defines when the component is active. It is basically a boolean predicate indicating when an entity should begin executing. The functional requirements are defined using the `requires` (pre-condition) and `ensures` (post-condition) clauses. These two clauses define component function as a relationship between current and next state axiomatically. Any implementation that makes the post-condition true in the next state, given that the pre-condition is true in the current state, is a valid implementation of these requirements. The `includes` clause is used to include PVS definitions in a VSPEC description. The sorts and operators defined in the PVS theories named by the `includes` clause can be used in the VSPEC definition. In the example specification from Figure 1, the `sort` component operates correctly in any initial state whenever its input changes and produce an output that is ordered and is a permutation of the input. Note that `event` is a predefined VSPEC predicate that is true whenever its associated signal changes values in the previous state change.

In addition to allowing the designer to describe functional requirements, VSPEC also allows the designer to specify performance constraints using the `constrained by` clause. This clause defines relations over constraint variables such as power consumption, layout area (expressed as a bounding box), heat dissipation, clock speed and pin-to-pin timing. Constraint theories are written in PDL [4], and verified using the associated evaluation tool. Users may define their own constraints and theories if desired [3].

The functional semantics are modeled upon the semantics of VHDL under simulation. Therefore, each entity behaves as an independent process, interacting with the outside world using messages sent and received through its ports. Each entity is modeled as a CSP [2] *process*, and architectures are modeled using CSP's *parallel composition* operator. Component interaction is specified in terms of *events*. Events are instantaneous actions that represent some real-world occurrence of interest. The set of events considered relevant for a particular description of an object is called the object's *alphabet*. A process represents the behavior pattern of an object described in terms of events from the object's alphabet. A sequence of events that a process participates in is called a *trace* of that process. A process is fully defined by its alphabet and the set of all possible traces of that process.

The semantics for VSPEC is given in the PVS specification language [1]. This includes definitions for all the operators and types used in VSPEC, and the meanings for the various VSPEC clauses. This formal definition of all the aspects of the language in PVS enables the formal analysis of models, as detailed in the next section.

```
package sort_pkg is
    type integer_array is mutable;
end sort_pkg;

use sort_pkg;
entity sort is
    port (input: in integer_array;
          output: out integer_array);
end sort;

use sort_pkg;
specification sort_spec of sort is
    includes SortPredicates;
    sensitive to input'event;
    requires true;
    ensures
        permutation(output'post, input)
        and inorder(output'post);
    constrained by
        power <= 5mW and size <= 3um * 5um
        and heat <= 10mW and clock <= 50MHz
        and input<->output <= 5 ms;
end sort_spec;
```

Figure 1: VSPEC description of a sorting component.

3. TRANSLATION

An example VSPEC file is shown in figure 2, and its corresponding generated-PVS file is shown in figures 3 and 4. The generation process is purely syntactic and is completely automated by the VSPEC parser. The heart of the transformation to PVS involves: (i) transforming ports and state variables into Store representation; and (ii) manipulating the `requires`, `ensures` and `sensitive to` clauses. The general structure of all generated PVS theories are similar, with the basic differences being in the parameters to the theories, and in the right hand sides of the various axioms.

```
entity m3 is
port ( in1 : in integer; inout1 : inout integer;
       out1 : out integer);
end m3;

specification m3_spec of m3 is
begin
    state state1 : integer;
    sensitive to in1'event;
    modifies inout1;
    requires in1 > inout1;
    ensures out1'post = state1 and
            in1 = inout1'post;
end m3_spec;
```

Figure 2: Example VSPEC file

The state of any VSPEC system is defined using a *Store*. The store is a simple abstraction of the record structure containing all the ports and state variables. Each theory representing a specification must have access to the type `Store`. Multiple definitions of a type in PVS results in each definition being a *different* type, thereby making proofs over stores impossible. To get around this problem, the `Store` type and the constant `empty` are passed as parameters to all the entity theories as their first two parameters. For the same reason, the type `Component` is also passed as a parameter. Since there always is a 'root' theory from which the analysis starts, this process ensures that only one `Store` and `Component` are visible throughout the system.

If the original VSPEC file had imported any PVS theories, they would be imported at this point. The reason for importing theories at this point is that types defined in those theories may be needed for the remaining parameters to the theory. The locations of the included theories are specified by the corresponding `LIBRARY` declarations. In our example, since there are no included theories, no `importing` statements are generated here. The remaining parameters are the declarations of the port variables. They are declared as functions from a `Store` to their corresponding type.

The body of the theory starts by declaring a constant `comp` that represents the current component. All the properties of components are defined over their respective `comps`. This enables the linkage of various properties to specific components. Next, the `OneComponent` and the `TypeSpecificInfo` theories are included. The `OneComponent` theory specifies all aspects of the process (associated with the VSPEC entity) independently from any VSPEC component. Theories representing specific components specialize `OneComponent`. The advantage of this approach is that verification of `OneComponent` need only be performed once. The basic theorems need not be reproven each time. The theory `TypeSpecificInfo` defines some operators that are common to all types. This theory is included once for each data type used in the system.

The state variables are all declared in a manner analogous to the port variables. They are declared as constant functions from a `Store` to their corresponding types. Next, some generic variables used in the axioms and theorems are declared.

The `sensitive to`, `requires` and `ensures` clauses are each transformed into axioms over stores. The `sensitive to` clause is defined by the axiom `sensitive_ax`, the `requires` clause by the axiom `requires_ax` over the `pre` state, and the `ensures` clause by the axiom `ensures_ax` over the states `pre` and `post`. The transformation of these clauses involves two basic activities: (i) combining the various occurrences of each clause into one; and (ii) replacing variable references with functions over Store.

The `modifies_event_ax` axiom and the `input_event_ax` axioms are part of the semantics of component activation and define when the functions `modSet_event` and `input_event` are true. The former is true when there is an event on one of the `modifies` variables (including the OUT and STATE variables). The latter is true when there is an event on an IN or INOUT variable.

For the component to ever become active, its initial state must be a part of the set of active states of the process. This fact is ensured by the first theorem (`initstates_th`) in the generated theory. Since, currently, there is no mechanism

```
%% PVS representation of specification m3_spec of entity m3
m3_spec [ Store: TYPE+, empty: Store, Component: TYPE+,
  in1 : [Store -> integer], inout1 : [Store -> integer],
  out1 : [Store -> integer]] : THEORY
BEGIN
% The component corresponding to this entity
comp: Component

IMPORTING jvsp@OneComponent[Store, empty, Component, comp]
...
%% The state variables in this entity
state1 : [Store -> integer]

%% Variables used in theorems and axioms
pre, post, any: VAR Store

%% Part of definition for semantics for event
modifies_event_ax : AXIOM modSet_event(comp)(any) =
  ( event(out1, any) OR event(inout1, any)  )
...
%% Requires Clause defines I
requires_ax : AXIOM I(comp)(pre) = (   in1(pre) > inout1(pre) )

% Ensures clause defines O
ensures_ax : AXIOM O(comp)(pre,post) =
  (  out1 (post)  = state1(pre) AND in1(pre) =  inout1 (post)  )
% Sensitive to clause
sensitive_ax : AXIOM member(pre,Psi(comp)) = ( event(in1, pre))
```

Figure 3: Partial PVS translation of specification m3_spec, part 1 – Variables and axioms

to specify the initial values of the various port and state variables, the axiom `initstates_ax` asserts that the initial state is equal to `Psi`. This automatically ensures us that `InitStates` is a subset of `Psi`.

The remaining theorems represent the single-component proof obligations. The completeness proof obligation is generated as the theorem `complete_th`, the witness for incompleteness obligation as theorem `incomplete_th` and the inconsistency obligation as the theorem `inconsist_th`. This completes the generated PVS theory for the specification `m3_spec`.

Abstract architectures are defined by: (i) specifying communication paths between components; and (ii) defining activation conditions to indicate when components should process inputs. The state of an architecture is defined to be the union of its components states. Component communication is achieved when their states share objects. Activation is the VSPEC dual of VHDL's sensitivity lists and indicate when a component should process its input. Together, communication and activation define the semantics of architecture specifications.

The VSPEC architecture for the `find` component is shown in figure 5. The VSPEC architecture is identical to the VHDL architecture, except for the use of the key word `VSPEC` to denote VSPEC specifications rather than VHDL components during instantiation. The variables in the signal declaration represent internal connections between the components in the architecture. Following the signal declarations, the components in the architecture are instantiated with appropriate parameters. The parameters represent connections with other components, or with the interfaces of the architecture, based on the name of the parameter. This concludes an architecture description. In our example, the `find_arch` architecture has two components – the `sorter` component, which is an instance of the `sort` component, and the `searcher` component, which is an instance of the `bin_search` component.

The PVS representation for this architecture (Figures 6, 7) has, as usual, the parameters `Store, empty` and `Component`. This enables the architecture to be imported in other architectures. Following this, the theory `Architecture` is imported to provide semantics to the various architecture operators. The ports of the higher-level component (`find`, in our example) are then declared. These form the ports of the architecture too, and provide inputs to and obtain outputs from the components in the architecture. The signals, which provide connections between the components in the architecture, are declared next.

The specifications of the higher-level component and the components in the architecture are imported with appropriate (depending upon how the component is connected) instantiations. The sole axiom of the theory (theorem `arch _comps_Axiom` in our example) defines the set `arch_comps` to be composed of the `comps` of all the components in the architecture. This declaration is essential for the definition of the architecture process, the process representing the parallel composition of these components.

```
%% Possible initial value in all traces of entity_process.
initstates_ax: AXIOM InitStates(comp) = Psi(comp)

%% Value of outputs and state variables does not change between post &
%% any. Initstates must be a subset of Psi for the model to be valid!
initstates_th: THEOREM subset?(InitStates(comp), Psi(comp))

%% Completeness obligation
complete_th: THEOREM (member(pre, LegalStates(comp)) AND
  member(pre, Psi(comp))) => I(comp)(pre)

%% Witness for incompleteness
incomplete_th: THEOREM EXISTS (x: Store): (member(x,LegalStates(comp))
  AND member(x, Psi(comp))) => NOT I(comp)(x)

%% Inconsistency obligation
inconsistent_th: THEOREM (member(pre, LegalStates(comp)) AND
  member(pre, Psi(comp))) => NOT I(comp)(pre)

END m3_spec

%% End of PVS representation of entity
```

Figure 4: PVS translation of specification m3_spec, part 2 – Theorems

```
architecture find_arch of find is
  signal sig_out1, sig_out2, sig_out3, sig_out4: integer;
begin
  sorter: VSPEC entity sort
          port map (in1, in2, in3, in4, sig_out1, sig_out2, sig_out3,
          sig_out4);
  searcher: VSPEC entity bin_search
            port map (sig_out1, sig_out2, sig_out3, sig_out4, key,
                      output);
end architecture find_arch;
```

Figure 5: VSPEC specification of `find` component's architecture

The generated theorems represent the various proof obligations for architectures. The input consistency proof obligation is generated as the `input_interface_th` theorem, the output consistency proof obligation as the `output_interface_th` theorem, strong liveness proof obligation for `sort` component as the `strong_live_sort_th1` theorem, and weak liveness and inconsistency proof obligations of the `bin_search` component as the theorems `weak_live_bin_search_th2` and `inconsist_bin_search_ th3`. The numbers at the end of theorem names are used to disambiguate between the same proof obligations of the same component used multiple times in the architecture. The `sort` component does not have theorems corresponding to the weak liveness and inconsistency proof obligations as there are no other components providing inputs to it. All its inputs are obtained from the interface of the architecture. Similarly, the `bin_search` component does not have a strong liveness theorem as all its outputs are part of the architecture interface.

4. PROOF AUTOMATION MECHANISM

Automation is critical for the success of any new methodology, and application of theorem proving is not an exception. But automation of theorem proving, in the general case, is not possible. Our approach has been to generate proof steps that have a high probability of successful completion. This approach is facilitated by the fact that both the generated theorems, and the semantics required for their proofs, have been written by us. This enables us to fine-tune the proof steps for individual theorems.

The proof steps generated for individual theorems are dependent upon the terms in the theorem. The general approach is to `LEMMA` the relevant axioms and theorems, instantiate them with appropriate constants, perform appropriate replacements of terms, and finally, to use PVS's built-in macro `GRIND` to attempt completion of the proof. Powerful prover commands provided by PVS are used in order to generalize the generated proof. For example, the `(INST?)` command provided by PVS attempts to automatically instantiate universally quantified variables in the antecedent with appropriate skolem variables.

The proof process starts by using the VSPEC parser to parse a component specification or architecture. The parser's `--pvs` flag generates the PVS equivalent of the parsed module. While generating theorems for the PVS module, the

```
input_interface_th: THEOREM member(pre,
        Psi(find_spec[Store, empty, Component, in1, in2, in3, in4,
        key, output].comp)) =>
    (member(pre, Psi(sort_spec[Store, empty, Component, in1, in2, in3,
        in4, sig_out1, sig_out2, sig_out3, sig_out4].comp))
    or member(pre, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp)))
...

strong_live_sort_th1: THEOREM
    O(sort_spec[Store, empty, Component, in1, in2, in3, in4, sig_out1,
        sig_out2, sig_out3, sig_out4].comp)(pre,post)
    => ( member(post, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp)) )

weak_live_bin_search_th2: THEOREM
    (member(pre, Psi(sort_spec[Store, empty, Component, in1, in2, in3,
        in4, sig_out1, sig_out2, sig_out3, sig_out4].comp))
    and O(sort_spec[Store, empty, Component, in1, in2, in3, in4,
        sig_out1, sig_out2, sig_out3, sig_out4].comp)(pre, post))
    => member(post, Psi(bin_search_spec[Store, empty, Component,
        sig_out1, sig_out2, sig_out3, sig_out4, key, output].comp))
...
END find_arch
% End of PVS representation of architecture.
```

Figure 7: Partial PVS representation of find architecture, part 2

corresponding proof scripts and LISP files for use with the PVS proof checker are also simultaneously generated. After completion of the parsing, the PVS proof checker is invoked in the batch mode. Since PVS uses a LISP interface, a LISP file, containing a sequence of commands for execution by PVS, can be passed to PVS in the batch mode.

The loader file is generated by the parser, and has a sequence of commands for execution by the PVS proof checker. The `load` commands loads a file called `prfobs_fns.lisp`, which defines two main LISP functions – `mytc` and `myprove`. The `mytc` function instructs PVS to typecheck a file. The `myprove` function first installs the generated proof script, then uses it to prove a specification. The results are stored in a file called `proof_status`, which is printed out at the end of the proof process.

5. RESULTS AND EVALUATION

In this section, we present the results of analyzing model systems using our approach. The main aim of performing this evaluation was to identify the factors that affect the automatability of proofs. Towards this end, the example systems incorporate a wide variety of situations. They include control-based and data-based activation of components; linear, branching and feed-back architectures; and systems with intentional bugs that result in incompleteness (necessitating proofs over existential quantifiers).

The automated theorem-proving analysis has been applied to a number of systems. The `find` system has been described throughout this dissertation. It was chosen for illustration as it is a conceptually simple example that demonstrates most of the features of our approach. Apart from the `find` example, the `AlarmClock` system (a synthesis benchmark developed by Synopsis), the `PIP` system (a Digital Signal Processing System developed by our sponsors), and the `CruiseControl` system (a standard modeling problem) were also analyzed.

The results of our analyses are presented in table 1. The first column of the table lists the various modules that were analyzed. The second column indicates whether a module is a single component or an architecture. The third column lists the number of theorems that were automatically proved. The fourth column lists the number of theorems with *possible* proofs in that module. This is critical for statistical analysis of the effectiveness of our approach. A theorem has a possible proof if it is not preempted by some other theorem. For example, successful proof to the Completeness proof obligation automatically implies that the Incompleteness and Inconsistency proof obligations cannot be proved. At the same time, successful proof of the Incompleteness proof obligation invalidates the Completeness proof obligation, but does not necessarily mean that the Inconsistency proof obligation should hold. It may or may not, depending on the model. Such cases are handled by the eighth column, where we list the number of (invalidated) theorems that are not provable (even by hand) in a given model. We are mainly interested in seeing how many of the provable (that is, possible minus invalidated) theorems are automatically proved. The fifth through seventh columns list the various causes (activation style employed, specification style employed, or presence of existential quantifiers) for why certain provable theorems could not be proved automatically. The specification style column also includes cases in which the problems are caused by deficiencies in the source specification language. The final column lists genuine bugs identified by the

Module Name	Comp/ Arch	Auto- mated	Pos- sible	Act. Style	Spec. Style	Exist. Quant.	Not Prov.	Bug
`find`	Comp	2	2					
`sort`	Comp	2	2					
`bin_search`	Comp	1	2	1				
`find_arch`	Arch	3	4		1			
`AC`	Comp	2	2					
`comparator`	Comp	2	2					
`counter`	Comp	2	2					
`mux`	Comp	2	2					
`AC_arch`	Arch	1	5		4			
`PIP`	Comp	2	2					
`PulseDetector`	Comp	1	3			1	1	
`InterrogatorDecoder`	Comp	1	2	1				
`PulseGenerator`	Comp	1	2	1				
`PIP_arch`	Arch	2	6	4				
`CruiseControl`	Comp	2	2					
`SystemState`	Comp	2	2					
`CDS`	Comp	2	2					
`CTS`	Comp	2	2					
`CruiseControl_arch`	Arch	4	6				1	1

Table 1: Results of applying automated proof obligations to various systems

```
%% PVS representation of architecture find_arch of
%% entity find
find_arch [Store: TYPE+, empty: Store,
           Component: TYPE+] : THEORY
BEGIN

IMPORTING jvsp@Architecture[Store, empty, Component]

...
%% Ports of the Higher Level Component
in1: [Store -> integer]
in2: [Store -> integer]
in3: [Store -> integer]
in4: [Store -> integer]
key: [Store -> integer]
output: [Store -> integer]

sig_out1: [Store -> integer]
sig_out2: [Store -> integer]
sig_out3: [Store -> integer]
sig_out4: [Store -> integer]

...
%% Variables used in theorems and axioms
pre, post, any: VAR Store
```

Figure 6: Partial PVS representation of `find` architecture, part 1

system.

There are four main factors that affect the provability of generated theorems. The most significant of these factors is the structure of the architecture. Linear architectures are the easiest to perform automated analysis. Branching architectures increase the complexity of generated theorems and require additional proof steps, such as proof by cases, in some cases. The most complicating structure is feedback, which has its greatest impact on bisimulation proofs. Systems with feedback require induction proofs, which, in the general case, cannot be automated. Therefore, systems with feedback cannot be automatically analyzed. A special case of systems with feedback is the use of local store by components. For all practical purposes, a system with a local store is similar to that same system generating outputs to a component that feeds the same values back to the original component during the next activation period. Therefore, systems with local store cannot be automatically analyzed.

The activation style employed plays a crucial role in the provability of the proofs. When using control-based activation, it becomes necessary to do an inductive proof over all traces of the system in order to identify the states in which a component can be activated. The validity of the pre-condition is checked in all such active states. On the other hand, when using data-based activation, the pre-condition can be verified directly from the activation condition, and so is automatable.

The third factor that affects the complexity of proofs is the specification style employed by the user to specify the user-defined operators in PVS. Specifications using *conservative extension* are more amenable to automated proofs as the prover can automatically expand relevant terms, whereas other specification styles require explicit introduction of relevant axioms into the proof process.

The final factor is the presence of existential quantifiers in the theorem proved. At present, there is no automated mechanism to provide correct instantiations to existential quantifiers. Therefore, while the semantics allow such proofs to be manually performed, they cannot be automated. However, one of the new engines being implemented for the PVS theorem prover is aimed at rectifying this deficiency. This engine, once fully implemented, is expected to find suitable instantiations (in most cases) if they exist. This engine would allow automated proofs to many more proof obligations than currently possible.

6. SUMMARY AND CONCLUSIONS

In this work, we presented an approach to using theorem proving for automatic analysis of abstract, component-based, designs. It involved the use of two different languages, one that is easy for the designers to write specifications in, and another in which it is easy to prove theorems. For abstract designs, the VSPEC specification language was used, while the PVS theorem prover was used for proving properties about the design.

The two-language approach necessitated the writing of the semantics of the specification language in the theorem proving language and translating specifications into that semantics. A number of interesting properties generic to component-based hardware-like systems were identified. These analyze the interfaces and interconnections of an architecture with respect to a specification. Since interfaces and interconnections were identified as the sources of most errors in systems design and implementation, these were expected to have the most impact. Translation is a straightforward process, with one PVS theory being generated for each VSPEC entity. The various clauses are generated as axioms within the theory. The proof obligations for the analysis of interfaces and interconnections within the model are generated as theorems within the theory.

The final part of our approach involved the generation of proof steps to attempt automated proofs to the generated theorems. The proof steps were generated simultaneously with the generation of theorems. The proof steps make use of powerful proof macros provided by the PVS proof checker so as to make the proofs generic.

Our methodology was evaluated with the goal of identifying the conditions under which it is or is not effective. Four models were analyzed using our approach – the `Find` model, the `AlarmClock` model, the `PIP` model and the `CruiseControl` model. Most of the provable theorems were shown to be automatically proved using the generated proof scripts. A number of factors affecting the automatability of the proofs were also identified. These include structure of the architecture, activation style, specification style and presence of existential quantifiers.

In conclusion, our approach to the problem is promising. The use of theorem proving facilitates appropriate handling of abstraction during the early design stages, while the automation makes this approach easily applicable. Our use of design abstraction and small theorems makes our approach tractable.

7. REFERENCES

[1] Judy Crow, John Rushby, Natarajan Shankar, and Mandayan Srivas. *A Tutorial Introduction to PVS*. SRI International, Menlo Park, CA, June 1995. Presented at WIFT'95.

[2] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978.

[3] Amitvikram Rajkhowa. Vspec constraints modeling, evaluation and verification. Master's thesis, University of Cincinnati, 1999.

[4] Ranga Vemuri, Ram Mandayam, and Vijay Meduri. Performance modeling using PDL. *Computer*, 29(4):44–53, April 1996.