A Framework for Software Architecture Verification

Kurt Lichtner, Paulo Alencar and Don Cowan
Computer Science Department & Computer Systems Group
University of Waterloo
Waterloo, Ontario
CANADA

Waterloo Technical Report, CS-99-23
July, 1999

Abstract

In this paper we present a framework for software architecture verification using machine-assisted formal proof. Our approach is based on the translation of an existing architecture description language (ADL) based specification to an alternate mathematical representation. We use higher-order logic as mechanized by the Prototype Verification System (PVS) as the formal basis of our framework. Our approach is not tied to any particular ADL. Rather, we define an ADL-independent model of architecture description which formalizes the fundamental design concepts of architecture modeling notations. A key feature of our framework is its flexibility; the architect can choose the design concepts that are modeled. Moreover, since the model is generic to many ADLs, it allows verification of systems that are specified using more than one notation. We introduce our model of architecture description, and illustrate the utility of our approach by verifying properties of an example architecture, a simple compiler specified in a pipe-and-filter architectural style.

1 Introduction

A notable benefit of formally specifying the top-level design, or *architecture* of a software system is that of early analysis or verification [6, 15]. Research activity in architecture specification and analysis has lead to a number of formal modeling notations. However, these *architecture description languages* (ADLs) [10] and accompanying tool-sets offer varying degrees of support for analysis, ranging from parser-based semantic analysis to system simulation. In previous work we outlined an approach for architectural analysis [12]. In this paper we present a theorem-proving based framework for software architecture verification which can augment or extend the analysis capabilities provided though ADLs and their environments.

The role of a formal architecture analysis framework is to assist the architect in answering questions regarding the system before it is constructed. For example, does the architectural specification possess certain critical properties or conform to stated constraints. Undetected errors or inconsistencies in a system's design are costly to correct when discovered at a later point in the development cycle. Architectural analysis may also serve other purposes, such as comprehension, reuse, re-engineering, or impact and change analysis. For example, an architect might need to know if the collection of architectural components and connectors remains well-formed after a new component is connected, or if a property still holds when one component is substituted for another.

In contrast to other approaches, our framework is not specific to a particular ADL; instead, we identify a conceptual model of architecture description languages (i.e., an architectural *meta*-model) that is generic to many currently available languages. This model captures the core, or intersection of design concepts, relationships and constraints fundamental in architecture description. Using the model as a guide, we translate an existing architectural description (specified using an ADL) into an alternate formal representation. Once translated, the representation can be subject to a variety of putative challenges to verify that it meets desired formal requirements. Figure 1 presents a high-level overview of our approach. We are currently re-

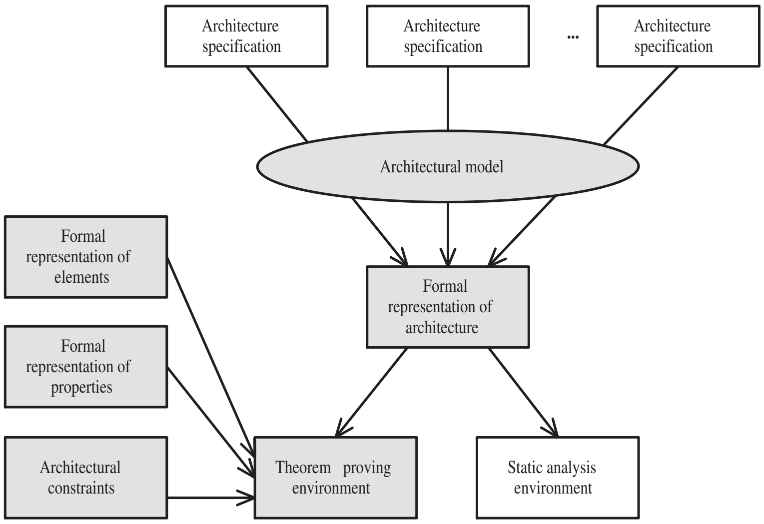

Figure 1: An ADL-based Verification Approach.

stricting our attention to the shaded areas of the diagram. The formal system that we employ is higher-order logic[1] [2], as mechanized by the PVS [14] theorem-proving environment.

An important aspect of the framework is its overall flexibility. As the model is intended to be generic across a broad category of ADLs, it can provide a unified verification environment for systems specified with more than one ADL. This allows the architect to leverage the specific strengths of a collection of notations, while providing an environment to combine them meaningfully.

The model can also be extended to incorporate other architectural design categories, including those from other architectural *views* [3]. This freedom allows architects to tailor the amount of detail extracted to that required by their specific design verification needs. This can include design concepts that might not be available for analysis with the original ADL and tool-set.

The framework is presented in the remainder of the paper as follows. In Section 2, we introduce our model of architectural description. As a rhetorical aid, we present it first in a high-level graphical modeling notation, and then fully define it using the PVS language. In Section 3 we illustrate our approach by defining and translating an architectural specification. We discuss mechanical verification in Section 4 by constructing formal descriptions of a set of properties, and proving that they follow from the architecture specification. We finish with conclusions and an outline of our future directions.

2 Introduction to the Model

In order to reason about an architecture we need its description in a suitable formal language. While the majority of architectural design environments are accompanied by text-based formal notations (i.e., ADLs), these languages are not in a form that is directly amenable to formal proofs. As a result, we introduce our own formal model of architecture description. Our model captures the categories of information that may

[1] Essentially first order predicate logic augmented with typed lambda calculus.

be present in an ADL-based architectural specification as well as the constraints and relationships between these categories.

2.1 Modeling Structure

It is generally accepted that the mainstay of architectural description is a hierarchically defined configuration of components and connectors [6], or architectural *elements*. The relative importance of system structure is reflected in current ADLs, most of which provide similarly extensive support for modeling this aspect of the architecture. As it is widely applicable, we focus on structure in this paper.

Figure 2 illustrates the main elements of the structural model. We first present the main design categories and relationships in a high-level graphical form (OMT [17]). This formalization is intended primarily to introduce and assist in understanding the model without overwhelming the reader with details. We then discuss the PVS formalization of the model. For this discussion we introduce a subset of our actual model, with enough detail to illustrate our approach. The reader is invited to reference [11] where a complete formalization is presented.

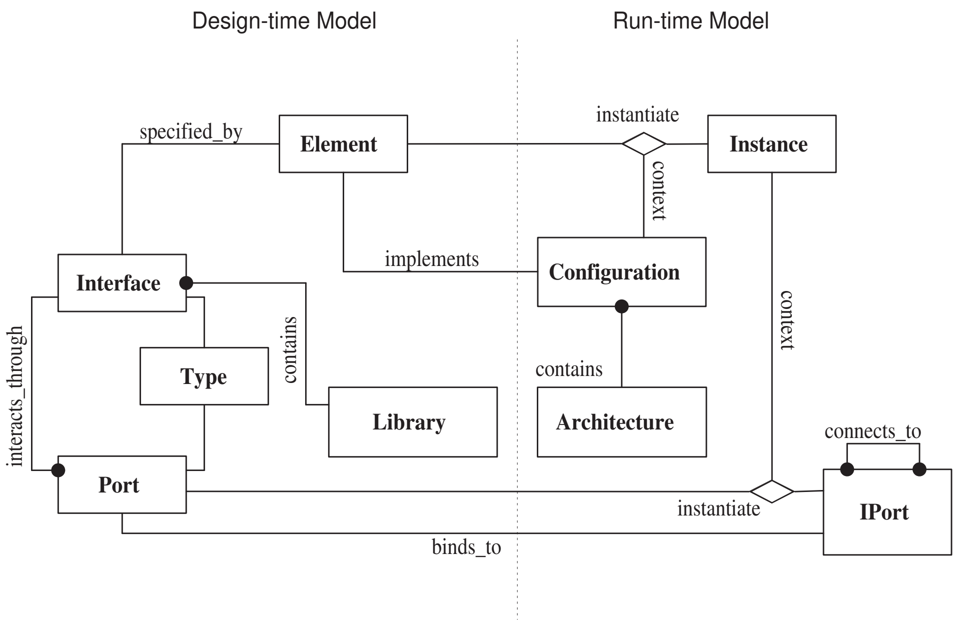

Figure 2: ADL Structural Model.

For clarity we have further subdivided the structural model into two sub-models, a *design*- and *run*-time formalization. The design-time model captures the primitive categories (both finite and infinite) from which we can build configurations. That is, they represent the elements that can be "instantiated" to form an architecture. The run-time formalization captures the arrangement and connection of instantiated elements within a specification. Since it represents a specific topology of interconnected components and connectors, the ability to reason about properties of the run-time model is critical. However, by also specifically modeling the collection of design-time concepts and their relationships, we gain the ability to challenge the specification with regard to the sets of primitive entities out of which the system is constructed.

2.2 The Prototype Verification System

PVS specifications are composed of a collection of *theories*. Theories are the basic units of modularity used to package the various elements of the specification, including its types, axioms, constants, and theorems. The PVS language is based on a strongly typed higher-order logic. It includes a rich collection of basic types, including booleans, integers, strings, enumerated types (`{red, blue, green}`), records (`[# field1:int, field2:string #]`), tuples (`[int,int]`)[2], functions (`[domaintype->rangetype]`), and sets (`setof[int]`)[3].

2.3 Basic Definitions

Our model is partitioned into three main theories, *basic_defs*, *design_time*, and *run_time* which formalize the basic definitions, design-time model and run-time model respectively.[4] We begin by defining the basic type definitions used in our model.

```
ELEMENT   : type = int
PORT      : type = int
INSTANCE  : type = int
IPORT     : type = [INSTANCE,PORT]
```

An `ELEMENT` represents the basic template which defines a component or connector[5]. Architectural elements interact with each other through specific connection points defined in their interface. These interaction points, modeled with the `PORT` type, define the services that an element offers or expects.

Within a configuration, an instance of an `ELEMENT` is represented by the type `INSTANCE`. Instantiating an `ELEMENT` also causes its ports to be instantiated. Instantiated ports are modeled as the tuple type `IPORT`, which pairs instances with the ports defined by their element type (i.e., it has type `INSTANCE` $\times$ `PORT`). As multiple instances of the same element type share the same set of basic `PORT` definitions, the role of instantiated ports is to partition the connection points of instances into disjoint sets.

2.4 Design-time Formalization

The remaining design-time entities are formalized in the `design_time` theory:

```
design_time [PORTTYPE : type, INTERFACETYPE: type] : THEORY
```

This theory is parametric in two types, `PORTTYPE` and `INTERFACETYPE`. The parameters serve as a means of integrating ADL-specific type information into the model. When instantiated for a particular ADL, type variables whose values represent the predefined port and interface "types" must be supplied. For example, some ADLs, such as UniCon [18], provide a rich "vocabulary" of architectural types, including *pipes*, *filters*, and *processes*, as well as a variety of legal port types. Abstracting specific details of the languages' type system out of the model significantly improves its genericity, making it applicable to a broader class of ADLs.

The design-time entity `Interface`, defines an element's type and the set of connection points through which it interacts:

```
Interface : type = [#
  interacts_through: setof[PORT],
  port_type:[(interacts_through)->PORTTYPE],
  type_of : INTERFACETYPE
#]
```

[2] The nth component of a tuple expression, `t`, is accessed as `proj_`n`(t)`.

[3] Sets can also be declared as predicates, i.e., functions mapping the set element type to `bool`. PVS allows a choice in syntax.

[4] The PVS source files are available at `ftp://csg.uwaterloo.ca/pub/kurt/arch_verification.zip`.

[5] In this paper we represent both components and connectors with the single category, `ELEMENT`.

The first field of the record, `interacts_through`, represents an element's set of ports. Each port is mapped to its type through the `port_type` function. The interface itself may be defined to have type information associated with it. This is captured by the `type_of` field.

The complete package of elements and interfaces available within the specification is modeled by the type `Lib`[6].

```
Lib : type = [#
  contains : setof[Interface],
  defines  : setof[ELEMENT],
  specified_by : [(defines)->(contains)]
#]
```

In addition to containing the set of interfaces and elements, `Lib` also defines the mapping, `specified_by`, which associates elements with their interfaces.

Two other entities are defined in the theory: the constant `initial_lib` of type `Lib`, which represents the initial (i.e., empty) library, and the operation, `add_element` which has the signature

```
[ELEMENT, Interface, Lib -> Lib]
```

and has the effect of adding the element to the library. Similarly, other library operations such as `delete_element`, or `update_element` may be provided. For succinctness, we have not included them here.

2.5 Run-time Formalization

The two main entities in the run-time formalization are *configurations* and *architectures*. A configuration consists of a set of instances, connections, and bindings:

```
Configuration : type = [#
  instances   : setof[INSTANCE],
  elements    : [(instances)->ELEMENT],
  ports       : setof[IPORT],
  connections : setof[[IPORT,IPORT]],
  bindings    : setof[[IPORT,PORT]]
#]
```

The function `elements` maps instances within the configuration to their corresponding element type. As previously mentioned, the instantiation of an element includes the instantiation of its ports. These are contained within the set, `ports`. The set, `connections`, defines the topology of the configuration by associating the ports of one instance with that of another. All of the architectural connections are modeled as a set of tuples of type `IPORT` $\times$ `IPORT`.

Our model allows hierarchical descriptions. Specifically, an entire configuration can be treated as a single element. These composite elements, like their primitive counterparts can then be instantiated within other configurations. To model these nested architectural descriptions, we must specify how the ports of the element are mapped to the instantiated ports of the configuration. This mapping is modeled by the field, `bindings`, which is a set of pairs of type `IPORT` $\times$ `PORT`. An example of a hierarchically defined element is shown in Section 3.

The final category formalized by the run-time model is an *Architecture*. An architecture represents all of the components of a complete specification:

```
Architecture : type = [#
  specLib     : Lib,
  composed_of : setof[Configuration],
  mainConfig  : Configuration
#]
```

[6] `library` is a reserved word in PVS.

An instance of this type consists of the library of elements and interfaces, and the set of configurations defined in the specification. From this set of configurations, only one is labeled as "main", or the configuration representing the system itself. The others are used in the definition of composite elements. Since most of the properties that we wish to prove will involve the architecture's main configuration, for convenience we also define a collection of operations which provide a "shorthand" notation for accessing its elements. For example:

```
instances(arch)    : setof[INSTANCE] = instances(mainConfig(arch))
elements(arch)     : [(instances(arch))->ELEMENT] = elements(mainConfig(arch))
ports(arch)        : setof[IPORT] = ports(mainConfig(arch))
...
```

Using these operations makes the conjectures (and proofs) less verbose, and thus more readable.

There are a number of other entities defined as part of the `run_time` theory. The constant `initial_configuration` represents the empty configuration. Two operations are defined on configurations: `connect`, which has the signature:

```
[IPORT, IPORT, Configuration -> Configuration]
```

and has the effect of adding a connection between the two specified ports, and `bind`, which has the signature:

```
[IPORT, Interface, PORT, Configuration -> Configuration]
```

and adds a binding between an instantiated port of the configuration and a port of the interface.

3 An Example Architecture

The example we present is that of a simple compiler specified in a *pipe-and-filter* [19] architectural style. The

```
architecture compiler {
  library {
    element filter is Component {
      type Filter;
      interface {
        port input  : dataIn
        port output : dataOut
      }
    }

    element pipe is Connector {
      type Pipe;
      interface {
        port input  : sink
        port output : source
      }
    }
  }
```

```
  configuration compiler_config implements filter {
    filter : lexer, parser, codegen;
    pipe   : p1, p2;

    connect lexer.output  to p1.input;
    connect p1.output     to parser.input;
    connect parser.output to p2.input;
    connect p2.output     to codegen.input;

    bind lexer.input    to input;
    bind codegen.output to output;
  }
}
```

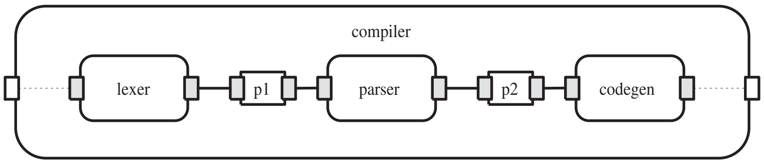

Figure 3: ADL-based compiler specification.

system is defined in both a pseudo-ADL and a graphical notation, and presented in Figure 3. The remainder of this section will discuss the translation of this specification into the PVS language. This phase is currently performed manually. However, automating the process should be relatively straightforward as most ADLs have publicly available grammar specifications.

3.1 Translation to PVS

The PVS representation of the architecture is constructed by introducing typed logical constants for each of the language constructs formalized by the model. We create a new PVS theory for the translated specification. The theories corresponding to the model are incorporated using the `IMPORTING` statement of PVS. We need only import the *run_time* theory, as it is built upon and imports the remaining theories. As the design- and run-time theories of our model are instantiated with ADL-specific type information, we define two enumerated types corresponding to the predefined element and port types supported by our pseudo-ADL. In this example we assume the built-in element types, *pipes*, and *filters*, and four possible types of ports, *dataIn, dataOut, sink*, and *source*:

```
compiler_arch  : THEORY
BEGIN
  elementTypes : type = {Pipe, Filter}
  portTypes    : type = {dataIn, dataOut, sink, source}

  IMPORTING run_time[portTypes,elementTypes]
```

3.2 Basic Definitions

We introduce a constant representing the filter element and its two ports:

```
  filter        : ELEMENT = 0
  filter_input  : PORT = 0
  filter_output : PORT = 1
```

Also as part of the translation phase, we generate unique values for the basic types modeled as integers. Note that we have prepended the name of the element to the name of its ports. As different elements may use identical names for their ports, relabeling eliminates the need to explicitly formalize the notion of "scope" within our models[7].

3.3 Design-time Formalization

The interface of the filter element is declared as a constant of the type, `Interface`:

```
  filter_interface : Interface = (#
    interacts_through := filter_interacts_through,
    port_type         := filter_port_type,
    type_of           := Filter
  #)
```

where `interacts_through` and `port_type` are given the following values:

```
  filter_interacts_through : setof[PORT] =
    add(filter_output, add(filter_input, emptyset))

  filter_port_type : [(filter_interacts_through)->portTypes] =
    (lambda (p:(filter_interacts_through)) :
      cond
        p = filter_input  -> dataIn,
        p = filter_output -> dataOut
      endcond)
```

[7] Which increases the complexity of the model.

The interface of the pipe element is declared in an analogous manner. Once the elements and interfaces are defined they can be added to the library using the `add_element` operation:

```
lib : Lib = add_element(filter, filter_interface,
            add_element(pipe, pipe_interface, initial_lib))
```

3.4 Run-time Formalization

The configuration of the compiler is specified as a series of instantiations, connections, and bindings on the initial, or empty configuration:

```
compiler_config : Configuration =
  bind((lexer,   filter_input),  specified_by(lib)(filter), filter_input,
  bind((codegen, filter_output), specified_by(lib)(filter), filter_output,

  connect((lexer, filter_output),  (p1, pipe_input),
  connect((p1, pipe_output),       (parser, filter_input),
  connect((parser, filter_output), (p2, pipe_input),
  connect((p2, pipe_output),       (codegen, filter_input),

  instantiate(p1,      pipe,   lib,
  instantiate(p2,      pipe,   lib,
  instantiate(lexer,   filter, lib,
  instantiate(parser,  filter, lib,
  instantiate(codegen, filter, lib, initial_configuration)))))))))))
```

Finally, the complete specification of the compiler is represented by an instance of type `Architecture`:

```
compiler : Architecture = (#
  specLib := lib,
  composed_of := add(compiler_config,emptyset),
  mainConfig  := compiler_config
#)
```

4 Mechanical Verification

Once the architectural description has been translated, we can mechanically verify that it meets certain formal requirements. The first step is to construct a mathematical description of the desired property and then demonstrate that it follows from (i.e., is a consequence of) the description. This entails proving theorems of the form: $\vdash$ *description* $\supset$ *property*.

The PVS environment includes a highly-automated proof checker, or interactive theorem prover, which assists the user in carrying out the proof steps. The user guides the proof checker with a series of commands corresponding to the deductive rules higher-order logic. At each step, the prover provides not only assistance in managing the tedious details of the proof, but also feedback as to what remains to be proven. While the primary role of the PVS prover is that of proof "assistant", it is capable of automating significant portions of the proof through a set of powerful, high-level proof commands. Many of the properties we have proved are easily demonstrated with the single command, `(grind)`. This command, among other things, repeatedly expands definitions, instantiates universal and existential quantifiers, and performs propositional simplifications until the result is either trivially true or some further direction from the user is required.

Properties relating to an architecture are expressed in higher-order logic using the same vocabulary of entity sets and relationships as previously defined in the architectural model (see Figure 2). This lets us prove putative challenges relating to any aspect of the model. The following challenges are examples of three important categories of structural properties:

- stylistic properties[8], such as "all instances are either of type `Pipe` or `Filter`":

[8] So named because they relate the system to a particular *architectural style*. See reference [19] for an introduction to style.

```
PipeFilterStyle : conjecture forall (i:(instances(compiler))) :
    (type_of(specified_by(compiler)(elements(compiler)(i))) = Pipe or
     type_of(specified_by(compiler)(elements(compiler)(i))) = Filter)
```

This conjecture is shown to be true with `(grind)`.

- properties relating to topology, or the interconnection of components and connectors, such as "pipes are only connected to filters and vice-versa":

```
Topology : conjecture forall (c:(connections(compiler))) :
  (proj_2(proj_1(c)) = filter_output => proj_2(proj_2(c)) = pipe_input) and
  (proj_2(proj_1(c)) = pipe_output   => proj_2(proj_2(c)) = filter_input)
```

Proving this conjecture takes slightly more effort as PVS divides the work into a set of subgoals, each of which must be proved separately with a series of manual substitutions and simplifications; however, each subgoal is proved in an identical manner:

```
(""
 (GRIND)
 (("1" (REPLACE -1 (-2 1) RL) (ASSERT)) ("2" (REPLACE -1 (-2 1) RL) (ASSERT))
  ("3" (REPLACE -1 (-2 1) RL) (ASSERT)) ("4" (REPLACE -1 (-2 1) RL) (ASSERT))
  ("5" (REPLACE -1 (-2 1) RL) (ASSERT)) ("6" (REPLACE -1 (-2 1) RL) (ASSERT))
  ("7" (REPLACE -1 (-2 1) RL) (ASSERT)) ("8" (REPLACE -1 (-2 1) RL) (ASSERT))))
```

- completeness[9] constraints, such as "the ports of all instantiated elements are either connected or bound to ports of an interface (that is, there are no unattached ports in the configuration)":

```
PortsAttached: conjecture
       forall (ip: (ports(compiler))):
         (exists (connection: [IPORT, IPORT]):
            member(connection, connections(compiler))
                AND (ip = PROJ_1(connection) OR ip = PROJ_2(connection)))
             OR
           (exists (binding: [IPORT, PORT]):
              member(binding, bindings(compiler)) AND (ip = PROJ_1(binding))))))
```

Again, this conjecture is easily proved. However a more informative directive must be given to the prover, `(grind :if-match all)`[10].

While the example architecture we have presented is simple, the properties we have demonstrated are illustrative of those that a software architect would find valuable. We have found that these putative challenges are useful for not only demonstrating that the specification of the system adheres to some formal requirements, but also for uncovering inconsistencies in the model itself.

5 Related Work

As previously discussed, architectural analysis is typically provided through ADLs and their environments. Each ADL is typically built upon a formal semantic foundation, which predisposes it to a particular class of properties and style of analysis. For example, the analysis provided by Rapide [13], which is based on system simulation, focuses on execution trace analysis. Wright [1], which is based on a subset of CSP [7],

[9] There are many types of *completeness*; we use the term to refer to completeness of the configuration with respect to some externally defined criteria.

[10] This simple fact took us several hours to discover.

can analyze a system for properties such as connector deadlock freedom. UniCon [18], which provides a rich and expressive architectural vocabulary, primarily supports enforcement of type constraints but also allows analysis of certain non-functional properties, such as real-time scheduability.

Our work is related to the ACME [5] interchange language in that it is similarly motivated. ACME is an architectural design syntax which primarily models architectural structure, but supports other design information through uninterpreted property sets. However, ACME is intended primarily as a vehicle for architectural design interchange, allowing the architect to leverage the benefits of using multiple design notations, and not as a basis for verification.

A number of other general frameworks for analyzing architectural designs have been proposed. The CHAM model [8], which is based on an abstract model of chemical reactions has been used for modeling and analysis of descriptions. Abd-Allah [4] has proposed a Z [20] framework as a means of analyzing the composition of architectures of different styles. ASDL [16], uses Z and CSP as a framework for an architectural style description language that also forms the basis of an approach to analysis. Each of these formalisms inherit the strengths and limitations of their underlying semantic foundation for modeling specific types of architectural properties. They are best viewed not as competing, but rather, complementary approaches to analysis.

Finally, our approach is related to work in static program analysis [9]. Both techniques are based on using conceptual models of the specification to extract and analyze the description. In the case of static program analysis, however, the specification is an executable program based on some operational semantics.

6 Summary and Future Work

We have presented an approach to software architecture verification which can complement the capabilities of other architectural tools. It is based on three main ideas:

- a formal model of architecture description
- generating an alternate mathematical representation of an existing architectural specification
- subjecting the translated specification to putative challenges to ensure it meets certain formal requirements

Our framework is characterized by its flexibility: the model is generic to a wide class of ADLs, which means that it can be used to integrate designs specified in different notation. The model can also be readily extended to include arbitrary design concepts.

Our formal proofs are conducted with the interactive theorem prover of the PVS environment. To date, our experience with PVS has been positive; we have found it to be a good match for this type of modeling. With its high-level proof commands, many interesting properties are demonstrated with only a small amount of guidance from the user.

Work is underway on extending the model with behavioral semantics. In addition to a structural model, we have also defined a model of behavior which captures the static (i.e., *syntactic*) aspects of event based component communication. We are working to extend this model with a dynamic framework based on execution sequence semantics [21] to provide a complete behavioral formalization. Related to the behavioral model, we are also investigating the possibility of modeling *dynamic* architectures, or systems whose topologies change as a result of their execution. As there is nothing specifically restricting our model to static architectures, in principle it is possible to extend it with semantics which allow for reconfigurable topologies.

References

[1] Robert Allen and David Garlan. The Wright Architectural Specification Language. Technical Report CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[2] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986.

[3] Paul Clements and Linda Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, Feb. 1996.

[4] Ahmed Abd el Shafy Abd-Allah. *Composing Heterogeneous Software Architectures.* PhD thesis, University of Southern California, August 1996.

[5] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[6] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[8] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[9] S. Jarzabek. Systematic design of static program analyzers. In *Proc. 18th Annual Int. Computer Software & Applications Conf.*, pages 281–286. IEEE Computer Society Press, Los Alamitos, USA, November 1994.

[10] Paul Kogut and Paul Clements. Features of architecture representation languages. Technical Report CMU/SEI-94-TR-tbd, Software Engineering Institute, December 1994.

[11] Kurt Lichtner, Paulo Alencar, and Donald Cowan. Formalizing Architecture Description Languages. Technical Report CS-98-07, University of Waterloo, July 1998.

[12] Kurt Lichtner, Paulo Alencar, and Donald Cowan. Using View-Based Models to Formalize Architecture Description. *Third International Software Architecture Workshop, ISAW-3*, Oct 1998.

[13] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[14] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. *Lecture Notes in Computer Science*, 607:748–752, 1992.

[15] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[16] Michael D. Rice and Stephen B. Seidman. An Approach to Architectural Analysis and Testing. *Third International Software Architecture Workshop, ISAW-3*, Oct 1998.

[17] M. Rumbaugh, M. Blake, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[18] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996. ISBN 0-13-182957-2.

[20] J. M. Spivey. *The Z Notation: A Reference Manual.* ISICS. Prentice-Hall, 2nd edition, 1992.

[21] G. Tredoux. Mechanizing execution sequence semantics in HOL. *The Computer Journal*, 7:81–86, July 1992. Proceedings of the 7th Southern African Computer Research Symposium, Johannesburg, South Africa.