A Survey of Architecture Description Languages

Ariel D. Fuxman

February 14, 2000

Abstract

In the last few years, a considerable number of architecture description languages (ADLs) have been proposed for describing and reasoning about software architectures. In this work, we classify and compare several ADLs placing emphasis on their features for formal modelling and analysis.

1 Introduction

There is no standard, universally-accepted definition of software architecture. A clear testimony of this is the recopilation of 18 definitions from the literature in the web page [def]. We will use as our working definition the one provided by Garlan and Perry [GP95]: a software architecture is "the structure of the components of a program/system and their interrelationships together with principles and guidelines governing their design and evolution over time."

A conceptual basis for software architecture was originally presented by Shaw and Garlan [GS93]. In their early work they do not attempt to supply specific mechanisms or notations but rather identify an emerging field and call for research in the area. They present a model of architecture based on three abstractions: *components*, *connectors* and *configurations*. Components are independent units of computation; connectors represent interactions among components; and configurations are instances of an architectural description represented as a connected graph of connectors and components. They also suggest that architectural *styles* form around component and connectors, repeating common patterns of computation and interaction.

Unfortunately, with few exceptions, current exploitation of software architecture and architectural style is informal and ad hoc. Architectural configurations are typically described using informal box and line diagrams in design documentation, providing little information about the actual computations represented by boxes, the interface of the components, or the nature of their interactions. This lack of information severely limits the usefulness of these diagrams for reasoning and analyzing a system's architecture.

In the last decade, there have been several proposals for providing a more sound basis for describing and reasoning about software architecture. In particular, a number of Architecture Description Languages (ADLs) have been proposed. ADLs provide constructs for specifying architectural abstractions in a descriptive notation. They provide mechanisms for decomposing a system into components and connectors, specifying how these elements are combined to form a configuration and defining families of architectures or styles.

The original intention of this report was to survey the application of formal methodology to architecture description languages. Unfortunately, it turned out that most current ADLs, with the notable exception of Wright [All97], do not directly support formal modelling or analysis. However, they present a number of features of importance for describing architectures, some of which might form the basis for future applications of formal methods. Therefore, this report gives a more general coverage of ADLs, though still emphasizing aspects such as formal semantic models and automated verification of architectural properties. In particular, we will survey some representative ADLs such as Rapide [LAK+95], Darwin [MDEK95], Aseop [GAO94], Unicon [SDK+95], Wright [All97] and ACME [GMW97].

The report is organized as follows. In section 2 we will compare the way the different ADLs support the main architectural elements: components, connectors, configurations and styles. In section 3 will we concentrate on formal aspects of ADLs, particularly semantic models and analysis of properties. Finally, in section 4 we will study other topics of ADLs such as code generation and architecture interchange.

2 Support of architectural elements

2.1 Components

According to [SDK+95], "components are loci of computation and state" . In this section, we present the aspects of components that need to be modeled in an ADL and assess existing ADLs with respect to them.

Interface

The *interface* of a component is a set of interaction points between itself and the external world. Interfaces enable a certain, though limited, degree of reasoning about component semantics.

All ADLs support specification of component interfaces. They differ in the terminology and the kinds of information they specify. For example, each interface point in ACME, Aesop, and Wright is called a *port*. In UniCon, an interface point is a *player*, and in Rapide a *constituent*. In Darwin, the interface consists of a collection of *services* that are either provided or required.

Types

ADLs can support reuse by modeling abstract components as types and instantiating them multiple times in an architectural configuration.

All of the surveyed ADLs distinguish component types from instances. However, UniCon and Darwin lack an explicit means of introducing new component types, supporting only a predefined set. Four ADLs make explicit use of parameterization: ACME, Darwin, Rapide and Wright.

Semantics

Modeling of component semantics enables analysis, constraint enforcement, and mapping of architectures across levels of abstraction. However, several languages do not model component semantics beyond interfaces. For instance, Darwin and UniCon define the semantics of components via their implementations, and ACME provides only property lists to model the semantics of components.

On the other hand, Wright and Rapide do model the behaviour of their components. In Wright there is a special section in the specification called `Computation` to describe component functionality. Rapide defines component types (called interfaces) in terms of a collection of communication events, which are either *observed* (Rapide calls these extern actions) or *initiated* (public actions). Rapide interfaces define the computational behaviour of a component by relating the observation of external actions to the initiation of public actions.

Evolution

ADLs should support component evolution through subtyping and refinement. Only a subset of existing ADLs provide this support. Even within those ADLs, evolution support is limited and often relies on the chosen implementation language.

Rapide supports inheritance. UniCon defines component types by enumeration, allowing no subtyping, and hence no evolution. ACME supports structural subtyping via its `extends` feature, while Aesop supports behavior preserving subtyping to create substyles.

2.2 Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors might not correspond to compilation units in implemented systems. They may be separately compilable message routing devices, or shared variables, table entries, buffers, procedure calls, initialization parameters, client-server protocols, etc. In this section, we present the aspects of connectors that we believe need to be modeled in an ADL and compare existing ADLs with respect to them.

Model of interconnection

In [Med96], languages that model connectors as first-class entities are called *explicit* configuration languages, as opposed to *in-line* configuration languages. The first class includes languages such as Wright, UniCon and ACME. On the other hand, Rapide and Darwin are examples of in-line configuration languages. In these languages, connectors cannot be named, subtyped, or reused. It is generally easier to evolve architectures described in explicit configuration ADLs than in in-line configuration ADLs: connectors in the latter are described solely in terms of the components they connect; adding new components may require modification of existing connectors.

Interface

In order to enable proper connectivity of components and their communication in an architecture, a connector should export as its interface those services it expects. Therefore, a connector's interface is a set of interaction points between it and the components attached to it. It enables reasoning about the well-formedness of a configuration.

Only those ADLs that model connectors explicitly support specification of connector interfaces. Wright, Aesop, ACME, and UniCon refer to connector interface points as *roles*. Explicit connection of component ports and connector roles is then required in an architectural configuration.

Darwin has a very weak notion of connectors. Connections are specified with *bindings* between the provided service of a component and the required service of another component. This model of connection enforces an asymmetrical model of interaction that implies that there is always one party in an interaction that is responsible for definition of the protocol. Thus, interaction patterns can not be described independently of the component that provides it.

UniCon supports explicit, symmetrical and asymmetrical connectors. Each connector has a collection of *roles* that define what participants are expected in a given interaction. Component interfaces, rather than providing or requiring services, are defined by *players* which have a type and a set of properties.

In Rapide, interaction between components is described for a particular configuration in one of two ways. First, events can be connected, in which case they are aliased to the same event. Second, an architecture can declare constraints which specify causal relationships between events on different components. Rapide's connections are similar to Darwin bindings in that they define an asymmetrical, primitive relation between two components. The language of actions permits the definition, essentially, of function calls between components (events have parameters and possibly return values.) As such, they, like Darwin bindings, are not adequate for the introduction of new interaction types and do not support symmetric interaction patterns. Rapide's constraints might appear to provide better support for an explicit connector mechanism, since related constraints can generate quite complex interactions between components. However, because the language does not permit them to be explicitly bundled as connectors, complex interaction patterns (sets of constraints) can not effectively be reused in multiple contexts.

A Wright description of a connector divides it into a set of *roles* and the *Glue*. Each role specifies the behaviour of a single participant in the interaction. The Glue of a connector describes how the participants work together to create an interaction. Like the Computation of a component, the Glue of the connector represents the full behavioural specification. According to the semantics of a Wright configuration, it is the Glue processes that will coordinate the component's behavior to create an interaction. A configuration is completed by describing *attachments* that associate a port of a component with a role of a connector. The component carries out a Computation, part of which is a particular interaction, specified by a Port. That port is attached to a Role, which indicates what rules the port must follow in order to be a legal participant in the interaction specified by the connector. If each of the components, as represented by their respective ports, obeys the rules imposed by the roles, then the connector Glue defines how the Computations are combined to form a single, larger computation.

In Figure 1 we show a client-server system specified in Wright. The first part of the description defines component and connector types. The `Client` and `Server` components have each a single port to interact with the environment. The connector `C-S-connector` has two roles, `client` and `server`, that specify their protocol of interaction. As we will see later, the client role might describe the client's behaviour as a sequence of alternating requests for service and receipts of results. The server role might describe the server's behaviour as the alternate handling of requests and return of results. The `glue` specification describes how the activities of the client and server roles are coordinated. It would say that the activities must be sequenced in a particular order: clients request a service, server handles the request, server provides a result, client gets the result.

The second part of the specification is a set of component and connector instances. In the example, there is a single server (`s`), a single client (`c`) and a single connector (`cs`). In the third part, components

System C-S- Example
  component Server =
    port provide *[provide protocol]*
    spec *[Server specification]*
  component Client =
    port request *[request protocol]*
    spec *[Client specification]*
  connector C-S-Connector =
    role client *[client protocol]*
    role server *[server protocol]*
    glue *[glue protocol]*
Instances
  s: Server
  c: Client
  cs: C-S-Connector
Attachments
  s.provide **as** cs.server;
  c.request **as** cs.client
end C-S-Example.

Figure 1: A Client-Server system in Wright

and connector instances are combined by prescribing the attachments that associate ports with roles. In the example, the client `request` and server `provide` ports are "attached as" the `client` and `server` roles, respectively. This means that the connector `cs` coordinates the behaviour of the ports `c.request` and `s.provide`.

Types

Architecture-level communication is often expressed with complex protocols. To make these protocols more abstract and reusable, ADLs should model connectors as types.

Only ADLs that model connectors as first-class entities distinguish connector types from instances. Wright, ACME and Aesop base connector types on protocols. UniCon, on the other hand, currently provides limited mechanisms for defining new connector types. New types can only be added by hand-implementing new "connection experts" that add to the collection of built-in, atomic types.

Semantics

To perform analysis of component interactions, consistent refinements across levels of abstraction, and enforcement of interconnection and communication constraints, architectural descriptions should provide connector protocol and transaction semantics. It is interesting to note that languages that do not model connectors as first-class objects, such as Rapide, model connector semantics.

ADLs generally use a single semantic model for both components and connectors. For example, Rapide uses posets [Pra86] and Wright models connector's Glue and roles with the process algebra CSP[Hoa85]. As is the case with its components, ACME allows specification of semantic information for connectors only in property lists.

2.3 Configurations

Architectural configurations are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior. Descriptions of configurations enable the assessment of important properties of architectures, such as a potential for deadlocks and starvation, performance, etc.

Understandability

A major role of architectures is to facilitate understanding of systems at a high level of abstraction. Therefore, ADLs must model structural information with a simple and understandable syntax, where system structure is clear from a configuration specification alone.

Configuration descriptions in in-line configuration ADLs, such as Rapide and Darwin, tend to be encumbered with connector details, while explicit configuration ADLs, such as UniCon, ACME, and Wright have the best potential to facilitate understandability of architectural structure.

Several languages provide both a graphical and a textual notation. Graphical specification provides another way of achieving understandability but only if there is a precise relationship between the graphical description and the underlying model. UniCon, Aesop, Rapide and Darwin support such graphical notations, while ACME and Wright do not.

Compositionality

An ADL may need to support situations in which an entire architecture becomes a single component in another, larger architecture. Therefore, support for compositionality, or hierarchical composition, is crucial.

Several ADLs provide explicit features to support hierarchical composition: ACME templates, composite elements in Darwin and UniCon, and Rapide maps. Other ADLs, such as Wright, allow hierarchical composition in principle, but provide no specific constructs to support it. [Iva98] proposes a model for hierarchical connectors in Wright.

Dynamism

Explicit modeling of architectures is intended to support the development and evolution of large and potentially long-running systems. It may be necessary to evolve such systems during execution. Configurations exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or subarchitectures. The majority of existing ADLs view configurations statically. The exceptions are Darwin and Rapide. Darwin allows runtime replication of components via dynamic instantiation. Rapide's `where` clause supports a form of architectural rewiring at runtime.

2.4 Styles

A style defines a set of properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations.

UniCon provides no means for describing or delineating families of systems. Darwin's support for architectural style is limited to the description of parameterized configurations. In Aseop, in accordance to its object-oriented focus, the vocabulary of an architectural style is described by defining subtypes of the basic architectural types: Component, Connector, Port, Role, Configuration, and Binding.

In Wright, common vocabulary is introduced by declaring a set of component and connector types and a set of constraints. Each of the constraints declared by a style represents a predicate that must be satisfied by any configuration declared to be a member of the style. The notation for constraints is based on first-order logic. The constraints can refer to the sets of components, connectors and attachments, to the ports and computations of a specific component and to the roles and glue of a particular connector. It is also possible to define substyles that inherit all the constraints of their parents.

3 Formal methods

By providing a precise semantics for the system at the abstract level of architecture, a formal model can provide the basis for rigorous, justifiable analysis of critical system properties. A formal model of the architecture can also be used as a basis for verification of an implementation. Because the constraints that must be met by the system are precisely defined, it is possible to determine whether a system conforms to an architecture and whether a given architecture conforms to a style.

3.1 First approaches

Even before the first ADLs were proposed, some researchers tried to apply general formal methods to modelling and analyzing architectures and architectural styles.

Inverardi and Wolf applied the CHAM (Chemical Abstract Machine) model [BB92] to the description of the architecture of a compiler. They argued that CHAM provides a useful basis for the description of architecture because of its ability to compose specifications from parts and to explicitly describe computation rules. However, the formalization is unique and hand-crafted, so there are no clear criteria for analyzing the consistency or completeness of the configuration descriptions.

There have been a number of efforts to apply formal methods to the characterization of architectural styles. Most of them have used Z [Spi92] as their notation, and describe the structure and semantics of a single class of systems, such as event systems or pipe-and-filters. These formalizations have the advantage of being applicable to a class of systems rather than a single system and to establish properties that systems in the described style share. However, each formalization is described in isolation, and the abstractions and techniques that they share must be re-invented for each formalization effort. Because they are developed independently of a general architectural model, it is not clear how to establish important properties such as the style's consistency.

Observing that many individual style formalization efforts shared a common set of characteristics, Abowd *et al.* [AAG93] defined a common framework in Z for the definition of an architectural style. They define a style by three parts : a concrete, *style-specific vocabulary* for the description of instances, a *semantic model* for reasoning about systems, and a set of *interpretation mappings* from the configuration syntax into the semantic model. These mappings provide interpretations for component, connector and configuration descriptions. Their common model has a number of advantages over stand-alone style characterizations. It clarifies the relationship between a configuration and a style, by showing how a style defines a semantic model (a collection of Z schemas) and a configuration is an element of that model (an instance of the schemas). It provides an interpretation of configuration consistency, since its description must be in the domain of an interpretation mapping. In summary, their work contributes to the theory of software architecture by providing a possible model for what a style is, and how a style model relates to descriptions of configurations. However, its practical application is limited mainly because since it uses unique, style-specific syntax and semantics there is no less effort involved in describing a style using these techniques than there is for the previously described stand-alone efforts.

The style model of Moriconi *et al.* [MQR95] is similar to that of Abowd *et al.* with two main differences: they use first-order logic for the definition of both configuration and style structures, and they provide a specific model of style-based refinement of configurations. Instead of providing interpretation mappings that map distinct configuration entities into a style's semantic space they define a configuration as a collection of predicates in first-order logic, which refer to the atomic predicates defined and related by a style. Thus, they view a configuration as a logical theory that includes a style's theory. They define refinement through interpretation mappings from one logical theory (*i.e.* a configuration or a style) to another. A refinement mapping is considered legal if all properties that are provable in the abstract theory are provable in the concrete theory, and all properties that are expressible but not provable in the abstract theory are not provable in the concrete one. They thus require a *completeness* assumption about the specifications that states that all properties not provable in a theory are false. This means that if an architect wishes to permit a property to be unspecified in a given system description, he or she must select a semantic theory in which is it impossible to state that property. Otherwise, failure to specify a property supplies a default answer. It is simpler and more practical in formalization to use *non-determinism* to indicate when a given property is not constrained. In this approach, if a property cannot be proved to be either true or false it is assumed that a refinement may make either choice. Some work in this direction is done by [PR99]. They define a refinement calculus for the pipe-and-filters style which supports underspecification using a refinement relation that is essentially defined as logical implication.

All these approaches based on formal methods provide strong analytic power, abstraction and independence from implementation details, but they have the weakness that they fail to provide basic architectural constructs and abstractions, such as connectors and styles, directly and, therefore, cannot be considered ADLs. The lack of predefined structure of a general formalism means that architectural abstractions must be defined from first principles for each new description. Most ADLs, on the other hand, provide a much better match to architecture by defining notations that are ready to apply to the problem of constructing complex real-world systems but are weak in analytic power, overly-specific to a subset of systems of interest, tied to specific implementations, and express the architecture at an overly low level of abstraction.

3.2 Semantic models for ADLs

Underlying semantic models vary across ADLs. For example, Rapide uses partially-ordered event sets ("posets") [Pra86], while Darwin uses π-calculus [MPW92]. UniCon and ACME only supply certain kinds of semantic information in property lists. Aesop assumes that the semantics of an architecture may be arbitrarily different for any style. As a result, it provides no built-in support for the description of a style's or a configuration's semantics but rather leaves placeholders to provide this information implicitly.

Darwin provides a semantics for its structural aspects through the π-calculus. Each provided service is modelled as a channel name, while each binding declaration is a process that transmits the name of that channel to a component that requires the service. This model has been used to prove the correctness of Darwin configurations [MDEK95]. Since π-calculus has been specifically designed for mobile processes, its use as semantic model confers Darwin configurations a number of dynamic properties.

The behaviour of architectural elements in Wright is specified in CSP [Hoa85]. CSP is a process algebra for specifying patterns of behavior and interaction. In CSP, behaviours are specified by combining events into patterns called *processes*. There is a process for each of the elements of a Wright description, one for each port, role, computation and connector glue. Abstractly, the behaviour of a Wright configuration can be described from the behaviour of individual components, each operating independently except that they are coordinated by the glue of the connectors to which they are attached. The computation of each component forms part of the overall behavior, where the order in which the computations occur and the transfer of data from one to the other is coordinated by the connectors. Continuing the previous example, the following are the process descriptions for the roles and glue of the client-server connector:

connector C-S-Connector =
    role Client = $(request!x) \rightarrow result?y \rightarrow Client) \sqcap \oint$
    role Server = $(invoke?x \rightarrow return!y \rightarrow Server) \Box \oint$
    glue = $(Client.request?x \rightarrow Server.invoke!x \rightarrow Server.return?y$
        $\rightarrow Client.result!y \rightarrow glue) \Box \oint$

The `Server` role describes the communication behaviour of the server. It is defined as a process that can repeatedly accept an invocation an then return a value; or it can terminate with success (denoted as $\S$) instead of being invoked. Because of the use of CSP's external choice operator ($\Box$) the choice between engaging in `invoke` and successful termination is determined by the environment of the role. The server is not allowed to terminate unless the environment in which it is operating also permits termination.

The `Client` role describes the communication behaviour of the client. Similar to the `Server`, it is a process that can request service and receive results repeatedly, or terminate. However, because CSP's internal choice operator ($\sqcap$) is used to describe the alternatives, the choice of whether to request a service or to terminate is determined by the `Client` process rather than by the environment.

The `glue` process coordinates the behaviour of the `Client` and `Server` roles. It allows the `Client` role to decide whether to request a service or to terminate and, in the first case, it sequences the invocation of the service and the transmission of results from the server to the client.

The semantics of Wright specifications themselves is given in CSP. For instance, the meaning of the `C-S-Connector` would be described as the parallel composition of its glue and roles:

$$glue \| (Client : Client \| Server : Server)$$

The colons denote prefixing of the events of a process. Then, for instance, the event `request` of the `Client` role would be renamed as `Client.request`. The symbol $\|$ denotes paralell composition. In CSP, parallel processes interact by synchronously engaging in the same events.

3.3 Analysis and automated verification

The types of analysis for which an ADL is well suited depend on its underlying semantic model. For example, Wright, which is based on CSP, analyzes individual connectors for deadlocks. Rapide's event monitoring and filtering tools also facilitate analysis of architectures.

Despite the presence of a π-calculus model for Darwin's structural descriptions, Darwin does not provide an adequate basis for analysis of the behavior of an architecture. This is because the model does not provide any means of describing the properties of either a component or its services. Component implementations are uninterpreted black boxes, while service types are a platform-dependent collection whose semantics is also uninterpreted in the Darwin framework.

Rapide permits consistency checking and analysis through architectural simulation. In essence, the architecture is simulated, generating a partially-ordered set of events that is compatible with the interface, behaviour, and constraint specifications. Because the generated set explicitly defines causal relations between events rather than simply providing one possible sequence of the events, simulation is useful for detecting execution alternatives such as race conditions. Rapide also provides a toolkit for testing the conformance of an implementation to the architectural specification by simulating the architecture in parallel with the execution of the implementation. However, a given simulation execution will provide only one possible poset, rather than all permitted posets. This means that using simulation to check for architectural consistency is essentially testing the architecture, rather than an exhaustive analysis. Failure to locate a problem provides no guarantee that the inconsistency will not arise on a different execution.

On the other hand, Wright defines consistency checks that can be applied to an architecture statically rather than through simulation, thus providing a stronger assurance of consistency than Rapide's. Some of the tests that can be performed automatically are port-computation consistency for components, port-role compatibility for attachments and deadlock freedom for the whole configuration.

A fundamental test that can be performed on Wright specifications is whether a port is compatible with a role in an specific configuration (port-role compatibility). The intuitive idea is that in a situation predicted by the protocol, the port must always continue the protocol in a way that the role could have. In other words, the port process should be substitutable for the role process in such a way that the rest of the connector interaction cannot detect that the role process has been replaced. In CSP this informal notion is captured formally by means of a refinement relationship between processes. The most important property of refinement is that if $P \sqsubseteq Q$, then Q must respect all of P's obligations to interact with its environment. We will illustrate this notion with the following two examples :

1. Client = $(request!x) \rightarrow result?y \rightarrow Client) \sqcap \oint$ and NonTermi-

natingClient = $(request!x) \rightarrow result?y \rightarrow NonTerminatingClient)$

2. Server = $(invoke?x \rightarrow return!y \rightarrow Server) \Box \oint$ and NonTerminatingServer = $(invoke?x \rightarrow return!y \rightarrow NonTerminatingServer)$

In the first example $Client \sqsubseteq NontTerminatingClient$ because `NonTerminatingClient` exercises the internal choice of requesting a service, but is otherwise consistent with the behaviour of `Client`. In the second example, $Server \not\sqsubseteq NonTerminatingServer$ because `NonTerminatingServer` refuses to terminate even though `Server` specifies that this decision can only be made by the environment.

While CSP's notion of refinement might be a good model for characterizing substitutability of ports for roles, in Wright it is not used directly. The reason for this is a concern about the ability to have a port fill as broad a set of roles as possible (or, equivalently, to reuse a connector in as many contexts as possible). This is because it may be that the port process is defined so that incompatible behavior with the role is possible in general, but would never arise in the context of the glue of the connector to which it is attached. For example, suppose a component port has the property that it must be initialized before use, but that it will crash if it is initialized twice. If we put this in the context of a connector that guarantees that at most one initialization will occur, then the anomalous situation will not arise. Thus, to evaluate compatibility it is necessary to restrict to the behaviour of the port over traces described by the role. This is because any trace of the port not described by the role represents a situation that will not arise in the interaction described by the connector.

Another fundamental test performed in Wright is deadlock-freedom verification. In CSP, a process is said to be deadlock-free if there is no situation in which it may refuse to participate in all events, but has not yet terminated successfully. In Wright, a connector is said to be deadlock-free if the roles and the glue work in such a way that if the overall connector process stops, it will be in a situation that is a success state for all the parties. Even though it is not obvious whether a connector remains deadlock-free when instantiated with compatible ports, this is indeed proved as a fundamental result in [All97]. The result is a theorem that proves that compatibility ensures deadlock-freedom of any instantiated well-formed deadlock-free connector. The significance of this theorem is twofold. First, it tells that local connector compatibility checking is sufficient to maintain deadlock-freedom for any configuration. Second, it provides a kind of soundness check for the definition of compatibility: under any execution, a compatibly-instantiated architectural description retains certain properties.

The verification of the consistency and deadlock-freedom properties is perfomed with a commercial model-checker for CSP, called FDR [For93]. Wright tools generate input to FDR from a Wright specification.

A key limitation of automatic exhaustive checking of models is that the models are required to be finite-state. This raises the problem of how to deal with the possibility that some specifications may not be finite-state. In fact, it is not decidable in general whether a process is finite- or infinite-state. Thus, there are two options: restrict the syntax of the language to a subset known not to permit infinite states or leave the syntax unrestricted and accept that it is impossible to guarantee that the checks will terminate. Wright chooses the latter option in order to provide as expressive and flexible a notation as possible.

Recently, there have been proposals for automatic property verification of architectures which rely on theorem proving rather than model checking and therefore allow models to have infinite states. For instance, Herbert *et. al.* [HDRS99] find and mechanize a general semantic framework for architectures that provides tractable models that allows properties to be proved in the higher-order logic proof tool PVS [ORS92].

A final is remark is that although Wright provides good analysis support for compatibility and deadlock-freedom properties it does not support a way to verify important properties such as the conformance of a configuration to an architectural style.

4 Other aspects

4.1 Code Generation

The ultimate goal of software design and modeling is to produce an executable system. An elegant and effective architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. It is, therefore, desirable for an ADL to provide code generation capabilities. Several languages enable system generation directly from architectural specifications. Aesop generates C++ code; UniCon, C code. Rapide can construct executable systems in C, C++, Ada. On the other hand, ACME, and Wright are used strictly as modeling notations and provide no code generation support.

One easy way to automatically generate code from an architecture is to associate architectural elements to implementation constructs. For instance, UniCon allows the specification of source files that correspond to given architectural elements. However, there are several problems with this approach: the assumption that the relationship between architectural elements and those of the resulting implementation is one-to-one may be unreasonable; there is also no guarantee that the specified source modules will correctly implement the desired behaviour or that future changes to those modules will be traced back to the architecture and vice versa.

4.2 Architecture interchange

Developed as a joint effort of several architectural research groups, ACME is intended to serve as a least-common denominator interchange language for architectural descriptions. ACME has focussed on providing a common skeleton through which the benefits of other languages and tools can be combined. It defines a basic, uninterpreted vocabulary of components, connectors, ports, roles, bindings, and configurations. These elements may have specifications associated with them through property lists. In ACME, properties may be specified in any language. The semantics of the properties and even of the overall architectural specification are supplied by these auxiliary languages. The goal is that a specification written in one language, say UniCon, could share a common architectural structure with a specification in another language, say Rapide, and thus the architect will be able to take advantage of the descriptive and analytic tools of multiple ADLs.

5 Conclusions

Our original intention was to survey the application of formal semantic models and automated verification to architectural description languages. At first glance, it might seem disappointing that most of the compared ADLs are not at all based on formal methodology. However, we can conclude from this work that each ADL has been designed with basically one particular goal in mind. For instance, Darwin is designed for dynamic architectures, Unicon is aimed for generating executable code from a description, ACME focuses on architectural interchange, and Rapide on simulation and architecture conformance. So, it is of no surprise that only one of the languages discussed has the features we were looking for. In fact, Wright precisely focuses on the integration of formal methodology to architectural description and effectively applies many of the concepts learned in class, such as process algebras and process refinement, to the automated verification of properties for software architectures.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan. "Using style to understand descriptions of software architecture". In *Proceedings of SIGSOFT '93*, December 1993.

[All97] R. Allen. *"A Formal Approach to Software Architecture"*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1997.

[BB92] G. Berry and G. Boudol. "The chemical abstract machine". *Theoretical Computer Science*, 96:217–248, 1992.

[def] "How do you define software architecture?" http://www.sei.cmu.edu/architecture/definitions.html.

[For93] Formal Systems (Europe) Ltd. *"Failures Divergence Refinement : User Manual and Tutorial"*, August 1993.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. "Exploiting sytle in architectural design environments". *Proceedings of SIGSOFT '94*, December 1994.

[GMW97] D. Garlan, R. Monroe, and D. Wile. "Acme : An architecture description interchange language". In *Proceedings of CASCON '97*, November 1997.

[GP95] D. Garlan and D. Perry. "Introduction to the special issue on Software Architecture". *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[GS93] D. Garlan and M. Shaw. "An introduction to Software Architecture". In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.

[HDRS99] J. Herbert, D. Dutertere, R. Riemenscheider, and V. Stavridou. "A formalization of software architecture". *Proceedings of World Congress on Formal Methods in the Development of Computing Systems FM '99*, 1, September 1999.

[Hoa85] C.A.R. Hoare. *"Communicating Sequential Processes"*. Prentice Hall, 1985.

[Iva98] J. Ivanisevic. "Hierarchical Connectors : A Contribution to Software Architecture". Master's thesis, University of Toronto, Toronto, Canada, 1998.

[LAK+95] D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan, and W. Mann. "Specification and Analysis of System Architecture using Rapide". *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. "Specifying distributed software architectures". *Proceedings of the Fifth European Software Engineering Conference ESEC '95*, September 1995.

[Med96] N. Medvidovic. A classification and comparison framework for software archtiecture description languages. Technical Report UCI-ICS-97-02, University of California, Irvine, February 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. "A calculus of mobile processes". *Journal of Information and Computation*, (100):1–77, 1992.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. "Correct architecture refinement". *IEEE Transactions on Software Engineering*, April 1995.

[ORS92] S. Owre, J. Rushby, and N. Shankar. "PVS : A prototype verification system". *11th International Conference on Automated Deduction*, June 1992.

[PR99] J. Philipps and B. Rumpe. "Refinement of Pipe and Filter Architectures". *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems FM '99*, 1, September 1999.

[Pra86] V. Pratt. "Modeling concurrency with partial orders". *International Journal of Parallel Programming*, (15):33–71, February 1986.

[SDK+95] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. "Abstractions for software architecture and tools to support them". *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[Spi92] J. Spivey. *"The Z notation"*. Prentice Hall, 1992.