Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events

David Garlan

Carnegie Mellon University, Pittsburgh PA 15213, USA,
garlan@cs.cmu.edu

Abstract. Developing a good software architecture for a complex system is a critically important step for insuring that the system will satisfy its principal objectives. Unfortunately, today descriptions of software architecture are largely based on informal "box-and-line" drawings that are often ambiguous, incomplete, inconsistent, and unanalyzable. This need not be the case. Over the past decade a number of researchers have developed formal languages and associated analysis tools for software architecture. In this paper I describe a number of the representative results from this body of work.

1 Introduction

The field of software architecture is concerned with the design and modeling of systems at a level of abstraction that reveals their gross structure and allows one to reason about key system properties, such as performance, reliability, and security. Typically architectural modeling is done by describing a system as a set of interacting components, where low-level implementation details are hidden, and relevant high-level system level properties (such as expected throughputs, latencies, and reliabilities) are exposed [29, 32].

Software architecture can be viewed as a level of design and system modeling that forms a bridge between requirements and code. By providing a high-level model of system structure it permits one to understand a system in much simpler terms than is afforded by code level structures, such as classes, variables, methods, and the like. Moreover, if characterized properly an architectural description should in principle allow one to argue that a system's design satisfies key requirements by appealing to abstract reasoning over the structure. Finally, an architecture forms a blueprint for implementations, indicating what are the principle loci of computation and data storage, the channels of communication, and the interfaces through which communication takes place.

To illustrate with a simple example, consider a simple pipelined dataflow architecture, in which streams of data are processed in linear fashion by a sequence of stream transformations, or "filters." When annotated with properties such as rates of processing, buffering capabilities of the channels, and expected input rates, one can typically reason about expected throughput and latency of the overall system. Additionally, the architectural structure likely mirrors the implementation structures For example, each filter might be implemented as a separate process communicating over buffered, asynchronous channels provided by the operating system.

Software architecture consequently plays a critical role in almost all aspects of the software development lifecycle.

Requirements specification: Architectural design allows one to determine what one can build, and what requirements are reasonable. Often an architectural sketch is necessary to assess product viability. For example, a preliminary architectural design might tell one whether subsecond response time is a feasible requirement on a new client-server system.

System design: Software architecture is a form of high-level system design. It typically determines the first, and most critical, system decomposition. A system without a well-conceived architecture is doomed to failure.

Implementation: As noted, an architecture is often the blueprint for low-level design and implementation. The components in an architectural description typically represent subsystems in the implementation, where the architectural interfaces correspond to the interfaces provided by an implementation.

Reuse: Most systems exhibit regular structures that represent instances of reusable idioms. For example signal processing systems are often designed as stream processing systems. Data-centric information systems are often designed as 3-tiered client-server systems. More generally, software architectures are a key component of *product lines* and *frameworks*. Those systems exploit architectural (and coding) regularities across a family of systems to make it possible to design and create new systems at low cost by specializing a general framework to create a particular product.

Maintenance: Software architectures facilitate maintenance by clarifying the system design, and enabling maintainers to understand the impact of changes. Since maintenance can account for well over half of a system's lifetime costs, and a substantial portion of maintenance is simply understanding a system in order to make a desired change, software architectures can be play a significant role in maintenance.

Run time adaptation: Increasingly systems are expected to operate continuously. Automated mechanisms for detecting and repairing system faults while a system is running will likely become essential capabilities in future systems. Software architecture can play an key role in supporting self adaptation, by providing a reflective model that can be used as a basis for automated repair.

Unfortunately, the potential uses of software architecture are thwarted by today's relatively informal approaches to architectural representation, documentation, and analysis. Architectural designs are, more often than not, simply informal "box-and-line" diagrams accompanied by prose. While these representations remain useful to practitioners [31] they suffer from their imprecision. Generally, it is not possible to use them for analysis, to determine with confidence whether some property holds of a system, whether a design is complete or consistent, whether an implementation conforms to an architectural design, or whether a proposed change violates an architectural principle.

In an effort to improve this situation many researchers have proposed formal notations and tools to set architectural design on a more solid engineering footing. Indeed, over the past decade dozens of architectural description languages (ADLs), numerous architectural evaluation methods, and many architectural analysis tools have been proposed by researchers [14, 23].

In the remainder of this paper, we outline some of the ways in which formal methods and notations can be brought to bear on software architecture. We begin with a brief introduction to software architecture. Next we consider various formal approaches to modeling and analyzing architectures. Then we briefly consider automated support, and conclude by listing some of the more interesting open research problems.

2 Software Architecture

Before characterizing ways in which we can apply formal modeling and analysis to software architecture, it is important to be clear about what we mean by the term. Definitions of software architecture abound. (The Software Engineering Institute's Web site catalogs more than 90 definitions [8].) A typical one is the following:

> The structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [6].

Unfortunately, as with most definitions of software architecture, this one begs the questions: What structures? What is a component? What kinds of relationships are relevant? What is an externally visible property?

In practice there are a number of kinds of structural decompositions of a system [8, 18]. Each of these has a legitimate place in the design and description of a complex software system, and each has its associated uses with respect to modeling and analysis.

One of these is a code decomposition, in which the primary elements are code modules (classes, packages, etc.). Relationships between these elements typically determine code usage and functionality relationships (imports, calls, inherits-from, etc.). Typical analyses include dependency analysis, portability analysis, reuse analysis.

A second class of decomposition characterizes the run-time structures of a system. Elements in such descriptions include the principal components of a system that exist as a system is running (clients, servers, databases, etc.). Also important in such descriptions are the communication channels that determine how the components interact. Relationships between these elements determine which components can communicate with each other and how they do so. Analyses of these structures address run-time properties, such as potential for deadlocks and race conditions, reliability, performance, and security. Whether a particular analysis can be performed will usually depend on the kind of system. For example, a queueing theoretic analysis might only be valid for a system composed of components that process streams of requests submitted by clients. Or, a schedulability analysis might only be valid for a system in which each component is treated as a periodic process.

Other structural representations might emphasize the physical context in which a system is deployed (processors, networks etc.), or developed (organizational teams or business units).

In this paper we focus on the second of these classes of structure: run-time decompositions emphasizing the principal computational elements and their communication channels. Sometimes this is referred to as the "component and connector" viewtype [8]. Indeed, in what follows, unless otherwise indicated, when we refer to the software architecture a system, we will mean a component and connector architectural view of it.

While systems can in principle be described as arbitrary compositions of components and connectors, in practice there are a number of benefits to constraining the design space for architectures by associating an *architectural style* with the architecture. An architectural style typically defines a vocabulary of types for components, connectors, interfaces, and properties together with rules that govern how elements of those types may be composed.

Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [11, 34, 7]. Moreover, the notion of style often maps well to widely-used component integration infrastructures (such as EJB, HLA, CORBA), which prescribe the kinds of components allowed and the kinds of interactions that may take place between them.

3 Formal Approaches to Software Architecture

Since architectural description is a multi-faceted problem, it is helpful to classify the properties of interest into several broad categories:

Structure: What are the principal components and the connectors that allow those components to communicate? What kinds of interfaces do components provide? What are the boundaries of subsystem encapsulation? Do the structures conform to any constraints on topology? Is the design complete?

Design Constraints: What design decisions should not change over time? What assumptions are being made that should be preserved in the face of future modification, or dynamically evolving architectures?

Style: What are the constraints implied by the architectural style? Does a given system conform to constraints of a given architectural style? What analyses are appropriate for a particular architectural style. What are the relationships between different architectural styles? Is it possible to combine two styles to produce a third one?

Behavior: What is the abstract behavior of each of the components? What are the protocols of communication that are required for two components to interact? Are the components behaviorally compatible? How does a system evolve structurally over time? Can we guarantee that all possible structures that emerge at run time will satisfy some property?

Refinement: Does a more detailed representation, and in particular a concrete implementation, respect the structure and properties of an architectural design?

Let us now consider how formal representations of software architecture can address many of these questions.

3.1 Formalizing Architectural Structure

Over the past decade there has been considerable research devoted to the problem of providing more precise ways to characterize the structure of software architectures, and to derive properties of those structures. Indeed, more than a dozen Architecture Description Languages (or *ADLs*) have been proposed. These notations usually provide both a conceptual framework and a concrete syntax for modeling software architectures. They also typically provide tools for parsing, unparsing, displaying, compiling, analyzing, or simulating architectural descriptions written in their associated language.

Examples of ADLs include Aesop [11], Adage [9], C2 [22], Darwin [20], Rapide [19], SADL [26], UniCon [30], Meta-H [7], and Wright [4]. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Adage supports the description of architectural frameworks for avionics navigation and guidance; Aesop supports the use of architectural styles; C2 supports the description of user interface systems using an event-based style; Darwin supports the analysis of distributed message-passing systems; Meta-H provides guidance for designers of real-time avionics control software; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that support a mixture of heterogeneous component and connector types; Wright supports the formal specification and analysis of interactions between architectural components.

Although there is considerable diversity in the capabilities of different ADLs, all share a similar conceptual basis [23], that determines a common foundation for architectural description. The main elements are:

– *Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases. In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.
– *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. That is, they provide the “glue” for architectural

designs, and intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors also have interfaces that define the roles played by the various participants in the interaction represented by the connector.
– *Systems* represent configurations (graphs) of components and connectors. In modern ADLs a key property of system descriptions is that the overall topology of a system is defined independently from the components and connectors that make up the system. (This is in contrast to most programming language module systems where dependencies are wired into components via import clauses.) Systems may also be hierarchical: components and connectors may represent subsystems that have "internal" architectures.
– *Properties* represent semantic information about a system and its components that goes beyond structure. As noted earlier, different ADLs focus on different properties, but virtually all provide *some* way to define one or more extra-functional properties together with tools for analyzing those properties. For example, some ADLs allow one to calculate overall system throughput and latency based on performance estimates of each component and connector [33].
– *Constraints* represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.
– *Styles* represent families of related systems. An architectural *style* typically defines a vocabulary of design element types and rules for composing them [32]. Examples include dataflow architectures based on graphs of pipes and filters, blackboard architectures based on shared data space and a set of knowledge sources, and layered systems. Some architectural styles additionally prescribe a framework[1] as a set of structural forms that specific applications can specialize. Examples include the traditional multistage compiler framework, 3-tiered client-server systems, the OSI protocol stack, and user interface management systems.

As a very simple illustrative example, consider a simple containing a client and server component connected by a RPC connector. The server itself might be represented by a subarchitecture. Properties of the connector might include the protocol of interaction that it requires. Properties of the server might include the

[1] Terminology distinguishing different kinds of families of architectures is far from standard. Among the terms used are "product-line frameworks," "component integration standards," "kits," "architectural patterns," "styles," "idioms," and others. For the purposes of this paper, the distinctions between these kinds of architectural families is less important than the fact that they all represent a set of architectural instances.

average response time for requests. Constraints on the system might stipulate that no more than five clients can ever be connected to this server and that servers may not initiate communication with a client. The style of the system might be a “client-server” style in which the vocabulary of design includes clients, servers, and RPC connectors.

This conceptual basis of ADLs provides a natural way to model the run-time architectures of systems. First, ADLs allow one to describe compositions of components precisely, making explicit the ways in which those components communicate. Second, they support hierarchical descriptions and encapsulation of subsystems as components in a larger system. Third, they support the specification and analysis of non-functional properties. Fourth, many ADLs provide an explicit home for describing the detailed semantics of communication infrastructure (through specification of connector types). Fifth, ADLs allow one to define constraints on system composition that make clear what kinds of compositions are allowed. Finally, architectural styles allow one to make precise the differences between kinds of component integration standards.

To be concrete, we now describe a representative ADL, called Acme [13] Acme supports the definition of four distinct aspects of architecture. First is structure—the organization of a system as a set of interacting parts. Second is properties of interest—information about a system or its parts that allow one to reason abstractly about overall behavior (both functional and extra-functional). Third is constraints—guidelines for how the architecture can change over time. Fourth is types and styles—defining classes and families of architecture.

Structure Architectural structure is defined in Acme using seven core types of entities: *components, connectors, systems, ports, roles, representations, and rep-maps.* Consistent with the vocabulary outlined earlier, Acme *components* represent computational elements and data stores of a system. A component may have multiple interfaces, each of which is termed a *port.* A port identifies a point of interaction between the component and its environment, and can represent an interface as simple as a single procedure signature. Alternatively, a port can define a more complex interface, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multicast interface.

Acme *connectors* represent interactions among components. Connectors also have interfaces that are defined by a set of *roles.* Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors may have more than two roles. For example an event broadcast connector might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles.

Acme *systems* are defined as graphs in which the nodes represent components and the arcs represent connectors. This is done by identifying which component ports are *attached* to which connector roles.

Figure 1 contains an Acme description of the simple architecture described above. A *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is defined by listing a set of *attachments* that bind component ports to connector roles. In this case, the client's requesting port is bound to the rpc's caller role, and the servers's request-handling port is bound to the rpc's callee role.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc  = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Fig. 1. Simple Client-Server System in Acme.

To support hierarchical descriptions of architectures, Acme permits any component or connector to be represented by one or more detailed, lower-level descriptions. Each such description is termed a *representation*.

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A *rep-map* (short for "representation map") defines this correspondence. In the simplest case a rep-map provides an association between internal ports and external ports (or, for connectors, internal roles, and external roles).[2] In other cases the map may be considerably more complex.

Figures 2 illustrates the use of representations in elaborating the simple client-server example. In this case, the *server* component is elaborated by a more detailed architectural representation.

Properties The seven classes of design element outlined above are sufficient for defining the *structure* of an architecture as a graph of components and connectors. However, there is more to architectural description than structure. But what exactly? Looking at the range of ADLs, each typically has its own forms of auxiliary information that determines such things as the run-time semantics of the system, protocols of interaction, scheduling constraints, and resource consumption. Clearly, the needs for documenting extra-structural properties of a system's architecture depend on the nature of the system, the kinds of analyses required, the tools at hand, and the level of detail included in the description.

[2] Note that rep-maps are not connectors: connectors define paths of interaction, while rep-maps identify an abstraction relationship between sets of interface points.

```
System simpleCS = {
  Component client = { ... }
  Component server = {
        Port receiveRequest;
        Representation serverDetails = {
          System serverDetailsSys = {

            Component connectionManager = {
               Ports { externalSocket; securityCheckIntf; dbQueryIntf } }

            Component securityManager = {
               Ports { securityAuthorization; credentialQuery; } }

            Component database = {
               Ports { securityManagementIntf; queryIntf; } }

            Connector SQLQuery = { Roles { caller; callee } }
            Connector clearanceRequest = { Roles { requestor; grantor } }
            Connector securityQuery = {
               Roles { securityManager; requestor } }
            Attachments {
               connectionManager.securityCheckIntf to clearanceRequest.requestor;
               securityManager.securityAuthorization to clearanceRequest.grantor;
               connectionManager.dbQueryIntf to SQLQuery.caller;
               database.queryIntf to SQLQuery.callee;
               securityManager.credentialQuery to securityQuery.securityManager;
               database.securityManagementIntf to securityQuery.requestor; }

       }
       Bindings { connectionManager.externalSocket to server.receiveRequest }
    }
  }
  Connector rpc  = { ... }
  Attachments { client.send-request to rpc.caller ;
                server.receive-request to rpc.callee }
```

Fig. 2. Client-Server System with Representation.

To accommodate the open-ended requirements for specification of auxiliary information, Acme supports annotation of architectural structure with arbitrary lists of properties. Figure 3 shows the simple client-server system elaborated with several properties. In the figure, properties document such things as the client's expected request rate and the location of its source code. For the *rpc* connector, properties document the protocol of interaction described as a Wright specification [4] (described in Section 3.4).

Properties serve to document details of an architecture relevant to its design and analysis. However, from Acme's point of view properties are uninterpreted values—that is, they have no intrinsic semantics. Properties become useful, however, when tools use them for analysis, translation, display, and manipulation.

```
System simple_cs = {
  Component client = {
        Port sendRequest;
        Properties { requestRate : float = 17.0;
                     sourceCode : externalFile = "CODE-LIB/client.c" }}

  Component server = {
        Port receiveRequest;
        Properties { idempotent : boolean = true;
                     maxConcurrentClients : integer = 1;
                     multithreaded : boolean = false;
                     sourceCode : externalFile = "CODE-LIB/server.c" }}

  Connector rpc  = {
        Role caller;
        Role callee;
        Properties { synchronous : boolean = true;
                     maxRoles : integer = 2;
                     protocol : WrightSpec = "..." }}

  Attachments {
     client.send-request to rpc.caller ;
     server.receive-request to rpc.callee }
}
```

Fig. 3. Client-Server System with Properties.

3.2 Formalizing Architectural Design Constraints

One of the key ingredients of an architecture model is a set of design constraints that determine how an architectural design is permitted to evolve over time. Acme uses a constraint language based on first order predicate logic. That is, design constraints are expressed as predicates over architectural specifications. The constraint language includes the standard set of logical constructs (conjunction, disjunction, implication, quantification, and others). It also includes a number of special functions that refer to architecture-specific aspects of a system. For example, there are predicates to determine if two components are connected, and if a component has a particular property. Other functions return the set of components in a given system, the set of ports of a given component, the set of representations of a connector, and so forth. Figure 4 lists a representative set of example functions. (For a detailed description see [25].)

Constraints can be associated with any design element of an architectural model. The scope of the constraint is determined by that association. For example, if a constraint is attached to a system then it can refer to any of the design elements contained within it (components, connectors, and their parts). On the other hand, a constraint attached to a component can only refer to that compo-

Connected(comp1, comp2)	True if component comp1 is connected to component comp2 by at least one connector
Reachable(comp1, comp2)	True if component comp2 is in the transitive closure of Connected(comp1, *)
HasProperty(elt, propName)	True if element elt has a property called propName
HasType(elt, typeName)	True if element elt has type typeName
SystemName.Connectors	The set of connectors in system SystemName
ConnectorName.Roles	The set of the roles in connector ConnectorName

Fig. 4. Sample Functions for Constraint Expressions.

nent (using the special keyword *self*, and its parts (that is, its ports, properties, and representations).

To give a few examples, consider the following constraints that might be associated with a system:

connected(client, server)

will be true if the components named *client* and *server* are connected directly by a connector.

Forall conn : connector in systemInstance.Connectors @ size(conn.roles) = 2

will be true of a system in which all of the connectors are binary connectors.

```
Forall conn : connector in systemInstance.Connectors @
    Forall r : role in conn.Roles @
        Exists comp : component in systemInstance.Components @
            Exists p : port in comp.Ports @ attached(p,r) and (p.protocol
= r.protocol)
```

will be true when all connectors in the system are attached to a port, and the attached (port, role) pair share the same protocol. Here the port and role protocol values are represented as properties of the port and role design elements.

Constraints can also define the range of legal property values, as in

self.throughputRate >= 3095

and indicate relationships between properties, as in

comp.totalLatency =
(comp.readLatency + comp.processingLatency + comp.writeLatency)

Constraints may be attached to design elements in one of two ways: as an *invariant* or a *heuristic*. In the first case, the constraint is taken to be a rule that cannot be violated. In the second case, the constraint is taken to be a rule that should be observed, but may be selectively violated. Tools that check for consistency will naturally treat these differently. A violation of an invariant makes the architectural specification invalid, while a violation of a heuristic is treated as a warning.

Figure 5 illustrates how constraints might be used for a hypothetical *MessagePath* connector. In this example an invariant prescribes the range of legal buffer sizes, while a heuristic prescribes a maximum value for the expected throughput.

```
System messagePathSystem = {
  ...
  Connector MessagePath = {
      Roles {source; sink;}
      Property expectedThroughput : float =  512;
      Invariant (queueBufferSize >= 512) and (queueBufferSize <= 4096);
      Heuristic expectedThroughput <= (queueBufferSize / 2);
    }
}
```

Fig. 5. *MessagePath* Connector with Invariants and Heuristics.

3.3 Formalizing Architectural Style

An important general capability for the description of architectures is the ability to define styles—or families—of systems. Styles allow one to define a domain-specific or application-specific design vocabulary, together with constraints on how that vocabulary can be used. This in turn supports packaging of domain-specific design expertise, use of special-purpose analysis and code-generation tools, simplification of the design process, and the ability to check for conformance to architectural standards.

The basic building block for defining styles in Acme is a type system that can be used to encapsulate recurring structures and relationships. Using Acme one can define types of components, connectors, ports, and roles. Each such type provides a type name and a list of required substructure, properties, and constraints.

Figure 6 illustrates the definition of a *Client* component type. The type definition specifies that any component that is an instance of type *Client* must have at least one port called *Request* and a property called *request-rate* of type float. Further, the invariants associated with the type require that all ports of a *Client* component have a *protocol* property whose value is *rpc-client*, that no client more than 5 ports, that a component's request rate is larger greater than 0. Finally, there is a heuristic indicating that the request-rate should be less than 100.

```
Component Type Client = {
    Port Request = {Property protocol: CSPprotocolT};
    Property request-rate: Float;
    Invariant Forall p in self.Ports @ p.protocol = rpc-client;
    Invariant size(self.Ports) <= 5;
    Invariant request-rate >= 0;
    Heuristic request-rate < 100;
}
```

Fig. 6. Component Type "Client."

An Acme style, or *family*[3] is defined by specifying a set of types and a set of constraints. The types provide the design vocabulary for the style. The constraints determine how instances of those types can be used.

Figure 7 illustrates the definition of a "Pipe and Filter" style, together with a sample system declaration using the style. The style defines two component types, one connector type, and one property type. The single invariant of this family prescribes that all connectors must be pipes. The system *simplePF* is then defined as an instance of the style. This declaration allows the system to make use of any of the types in the style, and it must satisfy all of the style's invariants.

But what does it mean for an instance to satisfy a type? In Acme, types are interpreted as predicates, and asserting that an instance satisfies a type is the same as asserting that it satisfies the predicate denoted by the type. The predicate associated with a type is constructed by viewing declared structure as asserting the *existence* of that structure in each instance. In other words, a type defines the *minimal* structure of its instances.[4] (Hence, in the example of Figure 7 it is essential to include the invariant asserting that all connectors have type *pipe*.)

The use of a predicate-based type system has several important consequences. First, design elements (and systems) can have an arbitrary number of types. For example, the fact that a structural element is declared to be of a particular type, does not preclude it from satisfying other type specifications. This is an important property since it permits, for example, a system to be considered a valid instance of a style, even though it was not explicitly declared as such.

Second, the use of invariants fits smoothly within the type system. Adding a invariant to a structural type or family simply conjoins that predicate with the others in the type. This means that the type system becomes quite expressive – essentially harnessing predicate logic to create useful type distinctions.

[3] For historical reasons a "style" in Acme is termed a "family."

[4] The semantics of the Acme type system is similar to – but considerably simpler than – that of other predicate-based type systems, such as the one used by PVS [28]. For a formal treatment of the semantics, see [25].

```
Family PipeFilterFam = {

  Component Type FilterT = {
        Ports { stdin; stdout; };
        Property throughput : int;
  };
  Component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
  };
  Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
  };
  Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
  Invariant Forall c in self.Connectors @ HasType(c, PipeT);

}

System simplePF : PipeFilterFam = {

    Component smooth : FilterT = new FilterT
    Component detectErrors : FilterT;
    Component showTracks : UnixFilterT = new UnixFilterT extended with {
        Property implementationFile : String = "IMPL_HOME/showTracks.c";
    };

    // Declare the system's connectors
    Connector firstPipe : PipeT;
    Connector secondPipe : PipeT;

    // Define the system's topology
    Attachments { smooth.stdout to firstPipe.source;
                  detectErrors.stdin to firstPipe.sink;
                  detectErrors.stdout to secondPipe.source;
                  showTracks.stdin to secondPipe.sink; }
}
```

Fig. 7. Definition of a Pipe-Filter Family.

Third, the process of type checking becomes one of checking satisfaction of a set of predicates over declared structures. Hence, types play two useful roles: (a) they encapsulate common, reusable structures and properties, and (b) they support a powerful form of checkable redundancy.

The use of predicates does, however, raise the issue that, in general, checking for satisfaction of predicates is not decidable. Therefore, systems that rely on predicate-based type systems usually do so with the aid of a theorem prover (for example, PVS [28]). In Acme, however, we constrain the expressiveness of types so that type checking remains decidable by ensuring that quantification is only over finite sets of elements. (Finiteness comes from the fact that Acme structures can only declare a finite number of subparts – components, ports, representations, and others.)

3.4 Formalizing Architectural Behavior

In addition to formal modeling of architectural structure, properties, constraints and styles, it is also useful to be able to model and analyze architectural behavior. By associating behavior with architectures, we are able to express much richer semantic models, capturing things such as the fact that a pipe provides buffered, order-preserving data transmission, or that a given component will call the services of another component in some particular order. This in turn allows us to attach analyze important properties, such as system deadlocks, race conditions, and interface incompatibilities.

In principle there are many possible ways one might specify behavior of the elements in an architectural model. Indeed, almost any formalism can be used, and researchers have experimented with formal techniques ranging from pre-post conditions [1], process algebras [4, 20], statecharts [5], POSets [19], rewrite rules [17], and many others.

However, all of these have a similar flavor: (1) they document the individual elements with behavior characterized in terms of abstract events, states and transitions, and (2) they then perform various composition checks or simulations to test for aggregate behavior, mismatches, deadlocks, and other anomalies.

Wright. To illustrate how this can be done, consider the Wright architecture specification language [4]. Wright adopts an approach based on the process algebra CSP [16]. Specifically it associates a CSP-like process with each component, each component interface (port), each connector, and each connector interface (role). The overall behavior is then a set of interacting protocols.

The notation used is a subset of CSP, containing the following elements:

– **Processes and Events:** A process describes an entity that can engage in communication events.[5] Events may be primitive or they can have associated data (as in `e?x` and `e!x`, representing input and output of data, respectively). The simplest process, STOP, is one that engages in no events. The event $\surd$ is used represent the "success" event. The set of events that a process, P, understands is termed the "alphabet of P," or αP.
– **Prefixing:** A process that engages in event `e` and then becomes process `P` is denoted $e{\rightarrow}P$.

[5] It should be clear that by using the term "process" we do not mean that the implementation of the protocol would actually be carried out by a separate operating system process. That is to say, processes are logical entities used to specify the components and connectors of a software architecture.

- **Alternative:** ("deterministic choice") A process that can behave like P or Q, where the choice is made by the environment, is denoted $P \,\square\, Q$. ("Environment" refers to the other processes that interact with the process.)
- **Decision:** ("non-deterministic choice") A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Unlike CSP, however, we restrict the syntax so that only a finite number of process names can be introduced. We do not permit, for example, names of the form $Name_i$, where i can range over the positive numbers.

In process expressions $\rightarrow$ associates to the right and binds tighter than either $\square$ or $\sqcap$. So $e \rightarrow f \rightarrow P \,\square\, g \rightarrow Q$ is equivalent to $(e \rightarrow (f \rightarrow P)) \,\square\, (g \rightarrow Q)$.

In addition to this standard notation from CSP we introduce three notational conventions. First, we use the symbol § to represent a successfully terminating process. This is the process that engages in the success event, $\surd$, and then stops. (In CSP, this process is called SKIP.) Formally, $\S \overset{\text{def}}{=} \surd \rightarrow STOP$. Second, we allow the introduction of scoped process names, as follows: **let** $Q = expr1$ **in** R. Third, as in CSP, we allow events and processes to be labeled. The event e labeled with l is denoted $l.e$. The operator ":" allows us to label all of the events in a process, so that $l : P$ is the same process as P, but with each of its events labeled. For our purposes we use the variant of this operator that does not label $\surd$. We use the symbol Σ to represent the set of all unlabeled events.

This subset of CSP defines processes that are essentially finite state. It provides sequencing, alternation, and repetition, together with deterministic and non-deterministic event transitions.

Connector Description. To see how this is used let us consider first how a connector is specified. A connector type is specified by a set of *roles* processes and a *glue* process. The roles describe the expected local behavior of each of the interacting parties. For example, the client-server connector illustrated earlier would have a client role and a server role. The client role process might describe the client's behavior as a sequence of alternating requests for service and receipts of the results. The server role might describe the server's behavior 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 the order: client requests service, server handles request, server provides result, client gets result.

This is how it would be written using the notation just outlined.

connector Service =
    role Client = request!x→ result?y → Client ⊓ §
    role Server = invoke?x→ return!y → Server □ §
    glue = Client.request?x→ Service.invoke!x
            →Service.return?y→Client.result!y→**glue**
            □ §

The Server role describes the communication behavior of the server. It is defined as a process that repeatedly accepts an invocation and then returns; or it can terminate with success instead of being invoked. Because we use the alternative operator ($\square$), the choice of invoke or $\surd$ is determined by the environment of that role (which, as we will see, consists of the other roles and the glue).

The Client role describes the communication behavior of the user of the service. Similar to Server, it is a process that can call the service and then receive the result repeatedly, or terminate. However, because we use the decision operator ($\sqcap$) in this case, the choice of whether to call the service or to terminate is determined by the role process itself. Comparing the two roles, note that the two choice operators allow us to distinguish formally between situations in which a given role is *obliged* to provide some services – the case of Server – and the situation where it may take advantage of some services if it chooses to do so – the case of Client.

The **glue** process coordinates the behavior of the two roles by indicating how the events of the roles work together. Here **glue** allows the Client role to decide whether to call or terminate and then sequences the remaining three events and their data.

The example above illustrates that the connector description language is capable of expressing the traditional notion of providing and using a set of services – the kind of connection supported by import/export clauses of module interconnection.

As another illustration, consider two examples of a shared data connector.

```
connector Shared $Data_1$ =
   role $User_1$ = $set \to User_1 \sqcap get \to User_1 \sqcap \S$
   role $User_2$ = $set \to User_2 \sqcap get \to User_2 \sqcap \S$
   glue = $User_1.set \to glue \;\square\; User_2.set \to glue$
            $\square\; User_1.get \to glue \;\square\; User_2.get \to glue \;\square\; \S$

connector Shared $Data_2$ =
   role Initializer =
      let A = $set \to A \sqcap get \to A \sqcap \S$
      in $set \to A$
   role User = $set \to User \sqcap get \to User \sqcap \S$
   glue = let Continue = $Initializer.set \to Continue$
                          $\square\; User.set \to Continue$
                          $\square\; Initializer.get \to Continue$
                          $\square\; User.get \to Continue \;\square\; \S$
          in $Initializer.set \to Continue \;\square\; \S$
```

The first, Shared $Data_1$, indicates that the data does not require an explicit initialization value. The second, Shared $Data_2$, indicates that there is a distinguished role Initializer that must supply the initial value.

To take a more complex example, consider the following specification of a pipe connector.

```
connector Pipe =
   role Writer = write→Writer ⊓ close→§
   role Reader =
      let ExitOnly = close→§
      in let DoRead = (read→Reader
                        [] read-eof→ExitOnly)
      in DoRead ⊓ ExitOnly
   glue = let ReadOnly = Reader.read→ReadOnly
                        [] Reader.read-eof
                           →Reader.close →§
                        [] Reader.close→§
         in let WriteOnly = Writer.write→WriteOnly
                           [] Writer.close→§
         in Writer.write→glue
            [] Reader.read→glue
            [] Writer.close→ReadOnly
            [] Reader.close→WriteOnly
```

It might appear to be a simple matter to define a pipe: both the writer and the reader decide when and how many times they will write or read, after which they will each close their side of the pipe. In fact, the writer role is just that simple. The reader, on the other hand, must take other considerations into account. There must be a way to inform the reader that there will be no more data.

Connector Semantics. The intuition behind a connector description is that the roles are treated as independent processes, constrained only by the glue, which serves to coordinate and interleave the events. To make this idea precise we use the CSP parallel composition operator, $\|$, for interacting processes. The process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2. That is, for the events in the intersection of the processes' alphabets, both processes must agree to engage in the event. We can then take the meaning of a connector description to be the parallel interaction of the glue and the roles, where the alphabets of the roles and glue are arranged so that the desired coordination occurs.

Hence, the *meaning of a connector description* with roles R_1, R_2, …, R_n, and glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1\mathbf{:}R_1 \parallel \mathrm{R}_2\mathbf{:}R_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}R_n)$$

where R_i is the (distinct) name of role R_i, and

$$\alpha Glue = \mathrm{R}_1\mathbf{:}\Sigma \cup \mathrm{R}_2\mathbf{:}\Sigma \cup \ldots \cup \mathrm{R}_n\mathbf{:}\Sigma \cup \{\surd\}.$$

In this definition we arrange for the glue's alphabet to be the union of all possible events labeled by the respective role names (*e.g.* Client, Server), together with the $\surd$ event. This allows the glue to interact with each role. In contrast, (except for $\surd$) the role alphabets are disjoint and so each role can only interact with the glue. Because $\surd$ is not relabeled, all of the roles and glue can (and must) agree on $\surd$ for it to occur. In this way we ensure that successful termination of a connector becomes the joint responsibility of all the parties involved.

Describing Components. Thus far we have concerned ourselves with the definition of connector types. To complete the picture we must also describe the ports of components and how those ports are attached to specific connector roles in a complete software architecture.

In Wright, component ports are also specified by processes: The port process defines the expected behavior of the component at that particular point of interaction. For example, a component that uses a shared data item only for reading might be partially specified as follows:

component DataUser =
    port DataRead = get→DataRead ⊓ §
    other ports...

Since the port protocols define the actual behavior of the components when those ports are associated with the roles, the port protocol takes the place of the role protocol in the actual system. Thus, an attached connector is defined by the protocol that results from the replacement of the role processes with the associated port processes. More formally, the meaning of attaching ports $P_1 \ldots P_n$ as roles $R_1 \ldots R_n$ of a connector with glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1\mathbf{:}P_1 \parallel \mathrm{R}_2\mathbf{:}P_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}P_n).$$

Note that this definition of attachment implies that port protocols need not be identical to the role protocols that they replace. This is advantageous because it allows greater opportunities for reuse. For instance, in the above example, the DataUser component should be able to interact with another component (via a shared data connector) even though it never needs to set. As another example, we would expect to be able to attach a File port as the Reader role of a pipe (as is commonly done in Unix when directing the output of a pipe to a file).

But this raises an important question: when is a port “compatible” with a role? For example, it would be reasonable to forbid DataRead to be used as the Initializer role for the Shared Data_2 connectors, since it requires an initial set; clearly DataRead will never provide this event.

Analyzing Architectural Behavior. Once one has a formal definition of behavior there are a number of analyses that one can perform. The most obvious one is checking that a connector is well-formed. That is to say, that the Glue in combination with the roles does not lead to deadlock. Another useful check is to investigate race conditions. This can be done by checking whether certain events can ever occur out of order.

Yet another check is to answer questions like "what ports may be used in this role?" At first glance it might seem that the answer is obvious: simply check that the port and role protocols are equivalent. But as illustrated earlier, it is important to be able to attach a port that is not identical to the role. On the other hand, we would like to make sure that the port fulfills its obligations to the interaction. For example, if a role requires an initialization as the first operation (*cf.*, the shared data example), we would like to guarantee that any port actually performs it.

Informally, we would like to be able to guarantee that an attached port process always acts in a way that the corresponding role process is capable of acting. This can be recast as follows: When in a situation predicted by the protocol, the port must always continue the protocol in a way that the role could have.

In CSP this intuitive notion is captured by the concept of refinement. Roughly, process P_2 refines P_1 (written $P_1 \sqsubseteq P_2$) if the behaviors of P_1 include those of P_2. Technically, the definition is given in terms of the failures/divergences model of CSP [16, Chapter 3]. For various technical reasons, however, the actual definition of compatibility is a little more complex to define, although it captures the same essential idea of refinement. (See [4] for details.)

As another check, one can investigate whether a port can be left unattached. This can be done by seeing if the port will deadlock when connected to a "do nothing" connector. Other checks are described in detail in [2].

Analyzing Reconfigurable Architectures Thus far the analysis has assumed a *static* architecture: that is, the structure of the architecture does not change during the execution of a system. While this is often a useful approximation to systems, clearly in the general case systems do evolve structurally. At the very least, during initialization the system must be created, and this is not likely to be an atomic operation.

As another example, consider a simple client-server system, such as the one illustrated earlier, but that allows for the possibility that a server may crash. In such cases the system might reconfigure itself so that the client uses a backup server. This can be done by adding a new connector during run time. One of the things we would like to guarantee for such a system is that no client requests are lost. This requires some constraints on when reconfiguration can happen.

Some work has been done to address these issues, although comparatively that work is relatively sparse. In our own work we showed how to extend Wright to handle dynamically changing topologies [3]. Others have looked at ways to use the Pi Calculus to specify such things [20]. Others have looked at graph grammars [24] and category-theoretic approaches [35]. Unfortunately, in all of these cases the complexity of the specification becomes drastically higher, and the models become much less tractable for static analysis.

4 Automated Support

For all of the formal approaches outlined earlier, researchers have developed numerous tools to aid in the modeling and analysis process for architects. Broadly speaking there are three general categories of tools:

1. **Design Assistants:** These tools tend to focus on providing a graphical front end to allow architects to develop designs. Typically they provide a pallet of component and connector types that can be instantiated to create system descriptions. Typical examples are environments such as C2 [22], MetaH [7], Aesop [11], and Darwin [20].
2. **Design Checkers:** While automated support for architectural creation and browsing is valuable, to be effective one must also provide analysis capabilities. Hence, a number of tools have been created to perform various checks. For example AcmeStudio [25] checks for violations of design constraints. Wright provides a tool for performing the checks outlined earlier. Those checks are based on the use of the FDR [10] model checker for CSP. Kramer and Magee demonstrate how to use their LTSA tool to check specifications written in their process algebra, FSP [21].
3. **Code Generators:** In many cases a formal definition of an architecture can be used to generate system code. For example, the UniCon system handles the generation of connector code for a wide variety of connector types [30]. Similarly C2 can generate partial implementations in using various infrastructures to handle component interaction.

5 Conclusion and Future Prospects

As we have tried to illustrate, software architecture is a field in which formal modeling and analysis can have a major impact. While the state of practice continues to rely on informal and semi-formal descriptions, considerable research has been done to develop good formal models and associated tools for analyzing them.

But the story is far from complete and there a number of areas in which further research is needed. Here are a few.

– Scalability: Although some large case studies have been carried out (e.g., [5]), there are relatively few demonstrated success stories for large, complex industrial systems. When systems have thousands of components, it is not clear how well the representation techniques (particularly graphical ones) scale. Nor is it clear whether analyses remain tractable. For example, many analysis tools are based on model checkers, which have significant limitation on the size of the model that can be checked.
– Dynamism: As noted earlier a key issue is modeling systems whose structure changes at run time.
– Code conformance: One of the big problems is guaranteeing that an implementation conforms to its architectural specification. In situations where a

code generator is used it is often possible to guarantee conformance by construction. But more generally, given an architecture and body of code, there has been very little work on finding ways to make sure they are consistent. The main problem is that architectures (as we have discussed them) represent run-time models, whereas code is obviously a design-time artifact. In general it is undecidable whether a given body of code will generate a given architecture.

There are also some intriguing new directions being explored in the area of self-adaptive systems. Increasingly systems are required to run continuously. Moreover they must often do this in the context of environments whose resources are constantly changing (e.g., wireless bandwidth), or whose components may be changing dynamically (e.g., web services). One approach that is being investigated by a number of researchers is the incorporation of self-adaptation or self-healing into a system. The interesting question is how should one do this?

One approach is to use architectural models as the basis for system monitoring and repair [12, 15, 27]. The idea is that the architectural model becomes available at run-time in order to understand whether a system is performing optimally, and if not it can be used model to reason about reasonable repair strategies at a high level of abstraction. While work is just beginning in this area, it appears to be a promising avenue for future research.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[2] Robert Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[5] Robert Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of of the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM Press.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1998. ISBN 0-201-19930-0.

[7] Pam Binns and Steve Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[8] Paul Clements, Felix Bachmann, Len Bass, David GArlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison Wesley, 2002.

[9] L. Coglianese and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[10] *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, 1.2β edition, October 1992.

[11] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[12] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In A. Romanovsky R. de Lemos, C. Gacek, editor, *Architecting Dependable Systems.* Springer-Verlag, 2003.

[13] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, page 47. Cambridge University Press, 2000.

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

[15] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, 2002.

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

[17] Paola Inverardi and Alex Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):373–386, April 1995.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[19] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

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

[21] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* Wiley, 1999.

[22] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering.* ACM Press, October 1996.

[23] Nenad Medvidovic and Richard N. Taylor. Architecture description languages. In *Software Engineering – ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, Zurich, Switzerland, September 1997. Springer.

[24] Daniel Le Metayer. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering.* ACM SIGSOFT, October 1996.

[25] Robert T. Monroe. *Rapid Develpomentof Custom Software Design Environments.* PhD thesis, Carnegie Mellon University, July 1999.

[26] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[27] P. Oriezy et al. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[28] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[29] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[30] 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, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[31] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Volume 1000. Springer-Verlag, 1995.

[32] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[33] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco, CA, June 1998.

[34] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[35] Michel Wermelinger. Formal specification and analysis of dynamic reconfiguration of software architecture. In *Proceedings of the 20th International Conference on Software Engineering*, volume 2, pages 178–179. IEEE Computer Society Press, 1998.