A Language and Environment for Architecture-Based Software Development and Evolution

Nenad Medvidovic
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-0781
+1-213-740-5579
neno@usc.edu

David S. Rosenblum
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1-949-824-6534
dsr@ics.uci.edu

Richard N. Taylor
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1-949-824-6429
taylor@ics.uci.edu

ABSTRACT
Software architectures have the potential to substantially improve the development and evolution of large, complex, multi-lingual, multi-platform, long-running systems. However, in order to achieve this potential, specific techniques for architecture-based modeling, analysis, and evolution must be provided. Furthermore, one cannot fully benefit from such techniques unless support for mapping an architecture to an implementation also exists. This paper motivates and presents one such approach, which is an outgrowth of our experience with systems developed and evolved according to the C2 architectural style. We describe an architecture description language (ADL) specifically designed to support architecture-based evolution and discuss the kinds of evolution the language supports. We then describe a component-based environment that enables modeling, analysis, and evolution of architectures expressed in the ADL, as well as mapping of architectural models to an implementation infrastructure. The architecture of the environment itself can be evolved easily to support multiple ADLs, kinds of analyses, architectural styles, and implementation platforms. Our approach is fully reflexive: the environment can be used to describe, analyze, evolve, and (partially) implement itself, using the very ADL it supports. An existing architecture is used throughout the paper to provide illustrations and examples.

Keywords
Software architecture, architecture description language, software environment, evolution, implementation mapping.

1 INTRODUCTION

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are commonly estimated to be as high as 60% of overall development costs [9]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus reducing opportunities for reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

An explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key property is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability: few specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [23]. Each of them may evolve. Our work to date has addressed the evolution of connectors and topologies [19, 22, 29, 37]. This paper discusses an approach for evolving software components, which has resulted from the recognition that an existing software module can evolve in a controlled manner via subtyping, as in object-oriented languages (OOPLs), for example. The novel aspects of this approach are:

- a taxonomy that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces;
- a flexible type theory for software architectures that is domain-, style-, and ADL-independent. By adopting a richer notion of typing, this theory is applicable to a broad class of design and reuse circumstances; and
- an approach to establishing type conformance between interoperating components in an architecture, which is better suited than other existing techniques to support the "large-scale development with off-the-shelf reuse" philosophy on which architecture research is largely based.

To support our approach to architecture-based evolution, we have designed a simple ADL that embodies the principles of the type theory. The ADL is accompanied by an environment

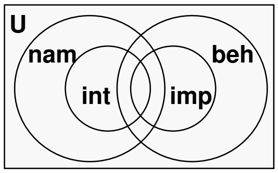

Fig. 1. A framework for understanding OO subtyping relationships as regions in a space, U, of type systems.

for architecture specification, analysis, and evolution. The environment also provides tool support for partial generation of an application from its architecture, which, in turn, facilitates reuse of off-the-shelf (OTS) components. The environment itself is component-based; its architecture was designed to be easily evolvable to support multiple ADLs, types of analysis, architectural styles, and implementation platforms. Our approach is fully reflexive: the environment can be used to describe, analyze, evolve, and (partially) implement itself, using the very ADL it supports.

The remainder of this paper is organized as follows. Section 2 summarizes the general principles of the architectural type theory, as well as the types of evolution and analysis the approach supports [21]. Section 3 introduces the ADL with the help of an example. Section presents and discusses the environment for modeling, analyzing, evolving, and implementing architectures described in the ADL. Section 5 discusses related work. A summary of our results to date, conclusions, and future work round out the paper.

2 THE TYPE THEORY

Motivation

Explicit treatment of types enables *subtyping*, the evolution of a given type to satisfy new requirements, and *type checking*, the determination of whether instances of one type may be legally used in places where another type is expected. A useful overview of PL subtyping relationships is given by Palsberg and Schwartzbach [31]. They describe a consensus in the OO typing community regarding a range of OO typing relationships. *Arbitrary subclassing* allows any class to be declared a subtype of another. *Name compatibility* demands that there exist a shared set of method names available in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass preserve the interface of the superclass, while possibly extending it. *Behavioral conformance* [2, 5, 15, 40] allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. Finally, *strictly monotone subclassing* additionally demands that the subtype preserve the particular implementations used by the supertype.

Protocol conformance goes beyond the behavior of individual methods to specify constraints on the order in which methods may be invoked. Explicitly modeling protocols has been shown to have practical benefits [1, 28, 39, 40]. However, component invariants and method preconditions and postconditions can be used to describe all state-based protocol constraints and transitions. Thus, our notion of behavioral conformance implies protocol conformance, and we do not address them separately.

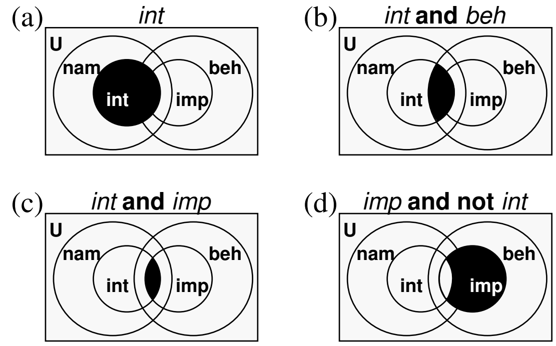

Fig. 2. Examples of component subtyping relationships we have encountered in practice.
(a) *interface conformance* — useful in interchanging components that communicate, e.g., via asynchronous message passing, without affecting dependent components in an architecture;
(b) *behavioral conformance* — guarantees correctness during component substitution;
(c) *strictly monotone subclassing* — enables extension of existing component's behavior while preserving correctness relative to the rest of the architecture;
(d) *implementation conformance with different interfaces* — allows a component to be fitted into an alternate domain of discourse (e.g., by using software adaptors [39]).

We have developed a framework for understanding these subtyping relationships as regions in a space of type systems, shown in Fig. 1. Each subtyping relationship described in [31] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the *int* and *beh* regions (Fig. 2b).

This framework was motivated by our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [18, 22]. Several examples are shown in Fig. 2. Note that we referred to the first three examples (Fig. 2a-c) using the terminology from the Palsberg-Schwartzbach taxonomy. However, while in OOPLs three different subtyping mechanisms would be provided by three separate languages, in architectures such heterogeneous subtyping mechanisms may need to be supported by the same ADL and may all be applied to components in a single architecture. A new type may also be created by subtyping from several existing types using different subtyping mechanisms. Finally, the example in Fig. 2d does not have a corresponding OOPL mechanism, further motivating the need for a flexible type theory for software architectures.

At the same time, by giving a software architect more latitude in choosing the direction in which to evolve a component, we allow some potentially undesirable side effects. For example, by preserving a component's interface, but not its behavior, the component and its resulting subtype may not be interchangeable in a given architecture. However, it is up to the architect to decide whether to preserve architectural type correctness, in a manner similar to America [2], Liskov and Wing [15], Leavens et al. [5], and others (depicted in Fig. 2b), or simply to enlarge the palette of design elements in a controlled manner, in order to use them in the future.

Component Subtyping

Every component specification at the architectural level is an *architectural type*. We distinguish architectural types from *basic types* (e.g., integers, strings, arrays, records, etc.). Unlike OOPLs, in which objects communicate by passing around other objects, in software architectures components are distinguished from the data they exchange during communication. In other words, a "component" in the sense in which we use it here is never passed from one component in an architecture to another.

A component has a *name*, a set of *interface elements*, an associated *behavior*, and (possibly) an *implementation*. Each interface element has a direction indicator (*prov*ided or *req*uired), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type.

A component's behavior consists of an *invariant* and a set of *operations*. The invariant is used to specify properties that must be true of all component states. Each operation has preconditions, postconditions, and (possibly) a result. Like interface elements, operations can be *prov*ided or *req*uired. Only provided operations will have an implementation in a given component. The pre- and postconditions of required operations express their *expected* semantics.

Since we separate the interface from the behavior, we define a mapping function from interface elements to operations. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation's result type.[1] This property directly enables a single operation to export multiple interfaces. An interface element and its corresponding operation comprise a *component service*.

Subtyping Relationships

Given this definition of a component, it is possible to specify component subtyping relationships. We summarize those relationships here. Their formal specification is given in [21].

Name conformance requires that a subtype share its supertype's interface element and parameter names. We have encountered name conformance in practice only as part of the stronger *interface* conformance relationship. Component C_2 is an interface subtype of C_1 if and only if it specifies *at least* the provided and *at most* the required interface elements from C_1, preserving contravariance of parameters and covariance of result.

Behavior conformance requires that the invariant of the supertype be ensured by that of the subtype. Furthermore, each provided operation of the supertype must have a corresponding provided operation in the subtype, where the subtype's operation has the same or weaker preconditions, same or stronger postconditions, and preserves result covariance. This relationship is reversed for required interface elements.

1. We do not explicitly model the semantics of basic types. However, we do allow the specification of subtyping relationships among them.

The subtyping relationship that results from the combination of behavior and interface conformance represents a point in the region depicted in Fig. 2b. This relationship is similar to other notions of behavioral subtyping [2, 5, 15] since it guarantees substitutability between a supertype and a subtype component in an architecture. Our approach is unique in that it establishes this relationship for *required* as well as provided services.

Finally, *implementation* conformance may be established with a simple syntactic check if the operations of the subtype have identical implementations as the corresponding operations of the supertype. On the other hand, establishing semantic equivalence between syntactically different implementations is undecidable in general. Techniques for making this problem tractable are outside the scope of our research.

Type Checking a Software Architecture

There is no single accepted set of guidelines for composing architectural elements into an architecture. Instead, topology depends on the ADL in which the architecture is modeled, the application domain, and/or the rules of the chosen architectural style. In specifying architectures, we adhere to the rules of the C2 style [37]: we model connectors explicitly, limit a component's attachments to single connectors on its top and bottom sides, and allow a connector to be attached to multiple components and connectors. None of these specific choices is required by our type theory. It is indeed possible to provide a definition of architecture that reflects other compositional guidelines. However, these kinds of decisions are necessary in order to formally specify and check type conformance criteria.

It is now possible to specify type checking predicates. A service provided by one component will satisfy a service required by another if and only if their interfaces match, the precondition of the required operation implies the precondition of the provided operation, and the reverse of this relationship holds for their postconditions. As already discussed, components need not be able to fully interoperate in an architecture. The two extreme points on the spectrum of type conformance are:

- *minimal type conformance*, where at least one service required by each component is provided by some other component along its communication links; and
- *full type conformance*, where every service required by every component is provided by some component along its communication links.

Depending on the requirements of a given project (reliability, safety, budget, deadlines, etc.), type conformance corresponding to different points along the spectrum may be adequate. Architectural type correctness is thus expressible in terms of a *degree* of conformance, rather than the "all or nothing" approach found in PLs.

Type Conformance and OTS Reuse

Establishing whether two components can interoperate includes matching the specification of what is expected by a required operation of one component against what another component's provided operation supplies. Behavior of an operation is modeled in terms of its interface parameters and component state variables. A component may thus need to refer to state variables that belong to another component in order to specify a *required* operation's expected behavior. However, doing so would be a violation of the "provider" component's abstraction. It would also violate some basic principles of component-based development:

- the designer may not know in advance which components, if any, will contain a matching specification for a required operation and, thus, what the appropriate (types of) state variables are. This is particularly the case when using behavior matching to aid component discovery/retrieval;
- large-scale, component-based development treats an OTS component as a black box, thereby intentionally hiding the details of its internal state. Having to explicitly refer to those details would require them to be exposed.

Existing approaches to behavior modeling and conformance checking have not addressed this problem. The problem does not apply to component subtyping: the designer must know all of existing component's details in order to effectively evolve it. Thus, those approaches that focus on behavioral subtyping (e.g., America [2], Liskov and Wing [15], and Leavens et al. [5]) do not encounter this problem. Zaremski and Wing [40] do address component retrieval and interoperability. However, their approach makes the very assumption that the designer will have access to a "provider" component's state (via a shared Larch trait [11]). Meyer [25] makes a similar assumption: all users of a component (class) have full access to the specification of its behavior. Fischer and colleagues [6, 35] model components at the level of a single procedure. In order to be able to properly specify pre- and postconditions, they include all the necessary variables as procedure parameters. Thus, for example, to implement a stack *push* procedure, the stack itself is passed as a parameter to the procedure.

The solution we propose to this problem is based on two requirements arising from a more realistic assessment of component-based development:

- we do not have access to a "provider" component's internal state (unlike Zaremski and Wing's approach), and
- we cannot change the way many software components, especially in the OO world, are modeled (unlike Fischer et al.).

We model a required operation as if we have access to a "provider" component's state. However, since we do not know the actual "provider" state variables or their types, we introduce a generic type, STATE_VARIABLE; variables of this type are essentially placeholders in logical predicates. When matching, say, a required and provided precondition, we attempt to unify (instantiate) each variable of the STATE_VARIABLE type in the required precondition with a corresponding state variable in the provided precondition. If the unification is possible and the implication holds, then the two preconditions conform.

3 THE LANGUAGE

The basic syntactic constructs needed in an ADL to support our type theory were introduced in [18]. We elaborate on those constructs here. This section introduces the basic concepts of C2SADEL (*S*oftware *A*rchitecture *D*escription and *E*volution *L*anguage), a language designed specifically to support architecture-based evolution. C2SADEL supports component evolution via heterogeneous subtyping and facilitates architectural descriptions that allow establishment of type-theoretic notions of architectural soundness. Before presenting the language constructs, we briefly describe an example architecture used for illustration purposes in the remainder of the paper.

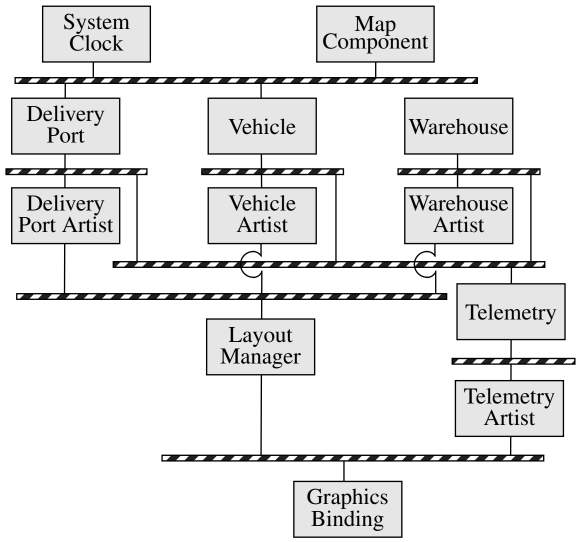

Fig. 3. Architecture of the cargo routing system in the C2 style.

Example Architecture

The architecture we use is a variant of the logistics system for routing incoming cargo to a set of warehouses, first introduced in [29] and shown in Fig. 3. The *DeliveryPort*, *Vehicle,* and *Warehouse* component types are objects that keep track of the state of a port, a transportation vehicle, and a warehouse, respectively. Each of them may be instantiated multiple times in a system. The *DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist* components are responsible for graphically depicting the state of their respective objects to the end-user. The *Layout Manager* organizes the display based on the actual number of port, vehicle, and warehouse instances. *SystemClock* provides consistent time measurement to interested components, while the *Map* component informs vehicles of routes and distances. The *Telemetry* component determines when cargo arrives at a port, keeps track of available transport vehicles at each port, and tracks the cargo during its delivery to a warehouse. *TelemetryArtist* allows entry of new cargo as it arrives at a port and informs the *Telemetry* component when the end-user decides to route cargo. The *GraphicsBinding* component renders the drawing requests sent from the artists using a graphics toolkit, such as Java's AWT.

C2SADEL

We encountered a tension between formality and practicality in designing C2SADEL. Our goal was a language that was simple enough to be usable in practice, yet formal enough to adequately support analysis and evolution. For this reason, we kept the syntax simple and reduced formalism to a minimum.

A C2SADEL architecture is specified in three parts: component types, connector types, and topology. The topology, in turn, defines component and connector instances for a given system and their interconnections.[2] A partial description of the architecture shown in Fig. 3 is given in Fig. 4. The *DeliveryPort* component type is specified *extern*ally, i.e., in a different file (*Port.c2*). The *GraphicsBinding* component type

2. C2SADEL also supports hierarchical composition, allowing an entire architecture to be used as a single component in another architecture. We do not discuss this feature here due to space constraints. Its thorough treatment is given in [17].

```
architecture CargoRouteSystem is {
  component_types {
    component DeliveryPort is extern {Port.c2;}
    component GraphicsBinding is virtual {}
    ...
  }
  connector_types {
    connector FiltConn is {filter msg_filter;}
    connector RegConn is {filter no_filter;}
  }
  architectural_topology {
    component_instances {
      Runway : DeliveryPort;
      Binding : GraphicsBinding;
      ...
    }
    connector_instances {
      UtilityConn : FiltConn;
      BindingConn : RegConn;
      ...
    }
    connections {
      connector UtilityConn {
        top SimClock, DistanceCalc;
        bottom Runway, Truck;
      }
      connector BindingConn {
        top LayoutArtist, RouteArt;
        bottom Binding;
      }
      ...
    }
  }
}
```

Fig. 4. Cargo routing system architecture specified in C2SADEL.

is specified as a *virtual* type: it can be used in the definition of the topology, but it does not have a specification and does not affect type checking of the architecture; furthermore, a virtual type cannot be evolved via subtyping. The concept of virtual types is useful in the case of components, such as *GraphicsBinding*, for which implementations are known to already exist, but which are not specified in C2SADEL.

A partial specification of the *DeliveryPort* component is given in Fig. 5. Component invariants and operation pre- and postconditions in C2SADEL are specified as first-order logic expressions. *DeliveryPort*'s invariant specifies that the current capacity of the port will always be between zero and the

```
component DeliveryPort is
  subtype CargoRouteEntity (int \and beh) {
    state {
      cargo    : \set Shipment;
      selected : Integer;
      ...
    }
    invariant {
      (cap >= 0) \and (cap <= max_cap);
    }
    interface {
      prov ip_selshp: Select(sel : Integer);
      req  ir_clktck: ClockTick();
      ...
    }
    operations {
      prov op_selshp: {
        let  num : Integer;
        pre  num <= #cargo;
        post ~selected = num;
      }
      req or_clktck: {
        let  time : STATE_VARIABLE;
        post ~time = time + 1;
      }
      ...
    }
    map {
      ip_selshp -> op_selshp (sel -> num);
      ir_clktck -> or_clktck ();
      ...
    }
}
```

Fig. 5. DeliveryPort component type specified in C2SADEL. Interface and operation labels (e.g., ip_selshp) are a notational convenience. "~" denotes the value of a variable after the operation has been performed, while "#" denotes set cardinality.

maximum allowed capacity. Examples of a provided and a required operation are given. The required *ClockTick* operation should be provided by the *SystemClock* component in the *CargoRouteSystem* architecture, such that *DeliveryPort*'s variable placeholder *time* can be unified with the appropriate *SystemClock* state variable to satisfy the postcondition of the operation. As discussed in Section 2, *DeliveryPort*'s interface is separated from its behavior and a surjective function is provided to map between the two.

The *DeliveryPort* component is a subtype of the more general *CargoRouteEntity* component, which is evolved by preserving both its interface and behavior, as shown in Fig. 5.

4 THE ENVIRONMENT

The concepts of component subtyping and type checking of architectures were initially motivated by specific problems encountered in exploring architecture-based development of distributed applications and OTS reuse in the context of the C2 style [18, 22, 37]. We have since developed a deeper understanding of many of the relevant ideas, including the formal underpinnings of the type theory [21]. To make the approach practical, we also needed to provide tool support for architecture-based subtyping and type checking, as well as for transferring the properties of an architecture to its implementation. This section describes the environment, called DRADEL (*D*evelopment of *R*obust *A*rchitectures using a *D*escription and *E*volution *L*anguage), which was developed to support modeling, analysis, evolution, and implementation of architectures described in C2SADEL.

The conceptual architecture of the DRADEL environment is shown in Fig. 6. Just like the application architectures it is built to support, the architecture of DRADEL itself adheres to C2 style rules. The environment is built using the C2 implementation infrastructure, an extensible framework of abstract classes for C2 concepts such as components, connectors, and messages [19]. The framework implements component interconnection and message passing protocols, and supports a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system process.[3]

Specifically, we have used the Java version of the C2 framework in the implementation of DRADEL. DRADEL's implementation consists of 13,000 source lines of code, in addition to the 7,000 lines of code in the base framework and C2 *GraphicsBinding*. In the current implementation, each DRADEL component executes in a separate thread of control within the same process.

DRADEL's Architecture

The *Repository* component from Fig. 6 stores architectures modeled in C2SADEL. The *Parser* receives via C2 messages specifications of architectures or sets of components, parses each specification, and requests that the *InternalConsistency-Checker* component check the consistency of the specification. The *InternalConsistencyChecker* builds its own representation of the architecture and ensures that components and

3. Note that implementation framework concepts, such as "abstract class," refer to the programming language used to implement an architecture, and are in no way related to our architectural type theory.

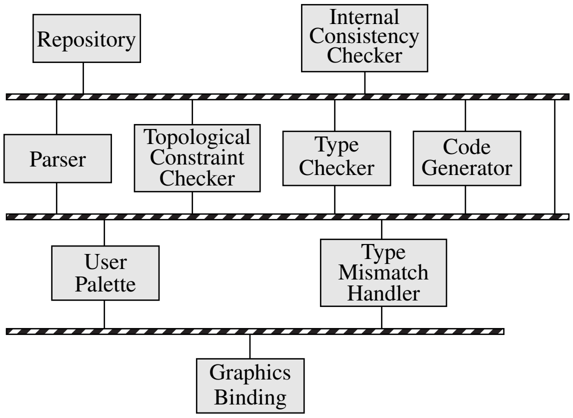

Fig. 6. Architecture of the DRADEL environment in the C2 style.

connectors are properly specified (e.g., two interface elements in a component cannot be identical), instantiated, and connected; that component interface elements are correctly mapped to operations (as specified in Section 2); that variables referenced in an operation are defined either as local variables for that operation or as component state variables; and that operation pre- and postcondition expressions are type correct (e.g., a set variable is never used in an arithmetic expression, although its cardinality may be). Furthermore, the *InternalConsistencyChecker* computes communication links for every component in an architecture: two components can interoperate if and only if they are on the opposite sides of the same connector (e.g., *Repository* and *Parser* in Fig. 6) or are on the opposite sides of two connectors which are, in turn, connected by one or more connector-to-connector links (e.g., *Repository* and *UserPalette* in Fig. 6).

Once the entire specification is parsed and its consistency ensured, its internal representation is broadcast to the *TopologicalConstraintChecker*, *TypeChecker*, and *CodeGenerator* components. If the specification in question is an architecture, the *TopologicalConstraintChecker* ensures that the topological rules of the C2 style are satisfied. The specification may contain a set of components instead, in which case no topological constraints are enforced.

The *TypeChecker* performs two kinds of analysis:

- given a specification of an architecture, it analyzes each component to establish whether its requirements are (partially or fully) satisfied by the components along its communication links;
- given a set of component specifications, the *TypeChecker* ensures that their specified subtyping relationships hold.

It performs these functions by establishing the relationships, discussed in Section 2, among component interface elements, invariants, and operations.

The *CodeGenerator* component generates application skeletons for the specified architecture or set of components. Like DRADEL itself, the application skeletons are built on top of the Java C2 framework. The "main program," containing component and connector instances and their interconnections, is automatically generated. For each specified component, the *CodeGenerator* creates the corresponding C2 component with the canonical internal architecture, shown in Fig. 7 [37]. The *InternalObject* of every generated component is a Java class

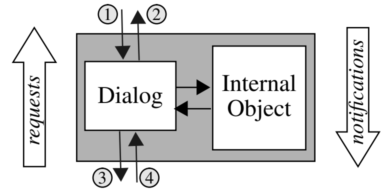

Fig. 7. Internal architecture of a canonical C2 component: (1) incoming notifications; (2) outgoing requests; (3) outgoing notifications; (4) incoming requests.

corresponding to the C2SADEL specification of the component. For example, the generated internal object for the *DeliveryPort* component from Fig. 5 is shown in Fig. 8: the state variables, state variable access methods, and provided component service declarations are generated.

Each method corresponding to a component service (e.g., *Select* in Fig. 8) is implemented as a null method in the generated class,[4] and is preceded by a comment containing the method's precondition and followed by one containing its postcondition. In general, these individual, application-specific methods are the only parts of a component for which the developers will have to provide an implementation. In the case of entirely new functionality, the pre- and postcondition comments serve as an implementation guideline to the developer; otherwise, they serve as an indicator of whether OTS functionality may be reused in the given context.

Except for the Java defined types (e.g., *Integer* or *String*), the *CodeGenerator* also supplies class skeletons for all other basic types, which include constructors and access methods. The

```
package c2.CargoRouteSystem;
import java.lang.*;
public class DeliveryPort extends Object
{
  // COMPONENT INVARIANT: cap >= 0 \and cap <= max_cap
  private Shipment_SET cargo;
  private Integer max_cap;
  private Integer cap;
  private Integer selected;

/***** Class Constructor *****/
  public DeliveryPort() {
    cargo = null;       // or: = new Shipment_SET(<init>);
    max_cap = null;     // or: = new Integer(<init>);
    cap = null;         // or: = new Integer(<init>);
    selected = null;    // or: = new Integer(<init>);
  }

/***** ADL Specified Methods *****/
  // PRECONDITION: num <= #cargo
  public void Select(Integer num)
  {
    /*** METHOD BODY ***/
  }
  // POSTCONDITION: ~selected = num

/***** State Variable Access Methods *****/
  public void SET_cargo(Shipment_SET new_value) {
    cargo = new_value;
  }
  public Shipment_SET GET_cargo() {
    return cargo;
  }
}
```

Fig. 8. Generated internal object class skeleton for the DeliveryPort component. The class is shown only partially due to space limitations.

4. The exception are non-void functions: Java requires their results to be initialized and returned. DRADEL initializes the results to an arbitrary value.

actual data structures must be specified in these classes by the developers. C2SADEL `set` types are currently implemented by subclassing from Java's *Vector* class, which provides a reasonable implementation of set abstractions.

The *Dialog* portion of a C2 component from Fig. 7 is responsible for all of component's message-based interaction. The *CodeGenerator* can generate a component's dialog almost completely from its C2SADEL specification: the provided services correspond to the notifications a component emits (message pathway `3` in Fig. 7) and the requests to which it is capable of responding (pathway `4`), while the required services are used as a basis for specifying the requests the component issues (pathway `2`). The dialog class also contains a specification of the component's message interface in the form needed by the underlying class framework to support various protocols of communication, e.g., message broadcast, registration, or point-to-point [37].

The only portion of the dialog that cannot be generated based on the information currently modeled in a C2SADEL specification is what the dialog should do in response to the notifications it receives (message pathway `1` in Fig. 7). This information could easily be specified in the ADL, as was shown in the prototype design language for C2-style architecture that preceded C2SADEL [24]; however, we have chosen to remove those constructs in the interest of language simplicity.

Given that a component's dialog is generated almost entirely from its C2SADEL specification, the internal object may, in fact, be completely replaced by an OTS component that does not communicate via messages. Essentially, the OTS component is modeled in C2SADEL, so that DRADEL can be used to check its compliance with the rest of the architecture and/or other components in its type hierarchy, as well as to generate its C2 dialog. This is the basic approach to OTS reuse we have used successfully in the past [19, 22].[5]

The remaining components in DRADEL's architecture, *UserPalette*, *TypeMismatchHandler*, and *GraphicsBinding* (see Fig. 6) handle the user's interaction with DRADEL. The *UserPalette* component drives the entire environment. It also displays the current execution status. The *TypeMismatch-Handler* component informs the user of the results of all component subtype matching and architectural type checking. Finally, as with any C2-style application, the *GraphicsBinding* renders DRADEL's user interface on the screen. The user interface is shown in Fig. 9, with examples of *nam*, *int*, and *beh* type conformance violations in the *CargoRouteSystem* architecture specified in Fig. 4.

The Process

The *UserPalette* component, rendered as the top pane of Fig. 9, enforces our "architecting" process. Prior to parsing a file, the *CheckConstr*, *TypeCheck*, and *GenCode* buttons are grayed out. Only once an architectural description or a set of components is successfully parsed and its internal consistency established can the user perform other functions. If the parsed file contains an architecture, the *CheckConstr* button is enabled, and the topological constraints must then be ensured. This must be

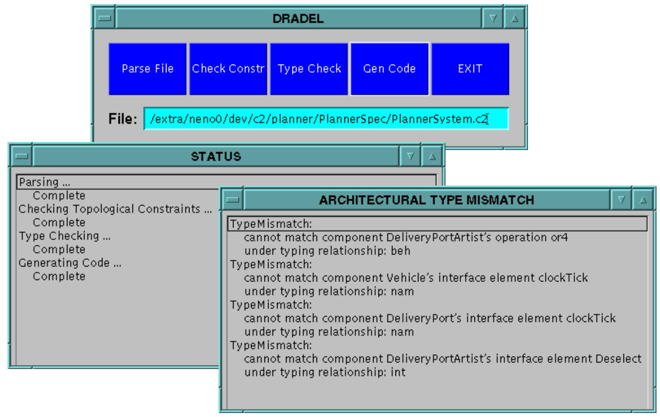

Fig. 9. DRADEL system's user interface.

done before either type checking the architecture or generating the application skeleton. If we are evolving design elements (i.e., ensuring the specified subtyping relationships among components), the *TypeCheck* and *GenCode* buttons are automatically enabled. This process is repeated every time the user decides to parse a new file.

If any errors are found during parsing, consistency checking, or topological constraint checking, a message will appear in the *Status* window informing the user of the exact error and its location. The user will not be able to proceed until the error is corrected. This is not the case with architectural type checking: the user can still generate the application, even if there are type errors, or the user may decide to skip the type checking stage altogether. The reasons for this are twofold.

Unlike a PL, which requires complete type conformance, architectures may describe meaningful functionality even if there are interface or behavioral mismatches. This has certainly been the case with C2-style applications, in which components communicate via asynchronous messages and are substrate independent [37]: C2 architectures are robust (hence the "R" in "DRADEL") in that a type mismatch may result in degraded functionality, or it may have no ill effects on the system, if it affects a part of the system that is not used in a given setting. It is the architect's responsibility to decide whether a given type mismatch is acceptable.

The other reason for allowing generation of type-mismatched components and architectures is the *TypeChecker* itself. To establish behavioral conformance between two components, the *TypeChecker* attempts to find mappings between their variables such that the correct (implication) relationships between their invariants, and pre- and postconditions hold. Since our approach does not explicitly model *basic types* (see Section 2), the *TypeChecker* essentially performs symbolic evaluation of logical expressions. For this reason, there are cases in which the *TypeChecker* is unable to correctly determine whether the implication indeed holds.

For example, assume that one component's operation precondition is

PRE1: $n <= 10$

and a candidate *beh*avior subtype's corresponding operation precondition is

PRE2: $n^2 <= 100$

where `n` in both expressions is declared to be of the (basic)

5. Prior to DRADEL's development, OTS component dialogs were implemented manually.

type *Natural*. To establish that the *beh*avior subtyping relationship holds, PRE1 must imply PRE2. This is obviously true: if a natural number is less than or equal to `10`, its square will be less than or equal to `100`. Note this would not be true of integers, for example (e.g., if `n = -20`). The *TypeChecker* has no knowledge of the non-negative property of natural numbers and cannot establish that the relationship between the two preconditions is true. It is pessimistically inaccurate: it treats those cases for which it cannot determine type conformance as errors. If the architect either judges the type mismatch not to be critical or discovers that the error is a result of *TypeChecker*'s inaccuracy, the architect has the choice to override the *TypeChecker* and proceed with code generation.

Discussion

DRADEL has been designed to be easily evolvable. Its components can be replaced to satisfy new requirements. For example, the file system-based *Repository* can be replaced with a database to keep track of design elements and architectures more efficiently. Another *Parser* can be substituted to support a different ADL, while a new *TopologicalConstraintChecker*, e.g., Armani [26], can be used to ensure adherence to a different architectural style. The *TypeChecker* may be replaced with a component that supports a different notion of architectural evolution and analysis; also, additional analysis tools may be added, even at runtime, using techniques described in [29]. Finally, the existing *CodeGenerator* may be replaced or new generation components added to support different implementation infrastructures.

DRADEL itself can be used reflexively to model and ensure the consistency of its own evolution. As the diagrams in Figures 3 and 6 and their subsequent discussions indicate, there are *no* fundamental differences between DRADEL and an application modeled, analyzed, evolved, and implemented with its help. Indeed, DRADEL's architecture can be specified in C2SADEL, parsed, checked for internal consistency, type checked, and the environment itself partially generated, as discussed above, using DRADEL.

5 RELATED WORK

This work has been influenced by research in several areas: OO typing, behavioral specification of software, software architectures and ADLs, and software environments. The relationship of our approach to OO typing and behavioral specification techniques is discussed in Section 2 and in more depth in [21]. In this section, we relate our approach to research in software architectures and environments. The unique aspects of our approach are

- component evolution via heterogeneous subtyping;
- formal specification, in type-theoretic terms, of the separation of provided and required functionality;
- analysis of architectures for consistency by unifying corresponding required and provided operations, where the architect possesses the authority to override the analysis tool ("architect's discretion");
- implementation generation; and
- a component-based, evolvable environment.

We look at related approaches mainly along these dimensions.

We have conducted an extensive survey of architectural approaches, which has indicated that their support for specification-time, architecture-based evolution is limited [20, 23]. ADLs that do address evolution typically rely on a chosen implementation language and only support a single form of subtyping/subclassing. Rapide [14] and LEDA [4] support evolution via inheritance. ACME [8] supports structural subtyping via its *extends* feature, while Aesop [7] supports behavior preserving subtyping to create substyles. UniCon [36] focuses on modeling, analyzing, and evolving software connectors; a connector is evolved by changing its properties or the properties of its interaction points (*roles*).

A majority of ADLs support some form of behavioral specification of components, connectors, and entire architectures. Three representative approaches are CHAM [12, 13], Rapide, and Wright [1]. These approaches allow the specification of dynamic behavior of components and/or connectors and the analysis of architectures for behavioral conformance (using the chemical abstract machine formalism, partially-ordered event sets and their patterns, and communicating sequential processes, respectively). Although, as discussed in Section 2, invariants and pre- and postconditions can be used to specify component interaction protocols, this has not been a particular focus or our research. Similarly to C2SADEL, the three approaches differentiate between provided and required functionality. CHAM and, to a limited extent, Rapide employ their equivalents of our variable placeholders to specify component expectations of the surrounding system. Wright addresses this issue indirectly, by interposing a connector between two, potentially independently developed, components. None of the approaches address the possibility of giving the architect the discretion to override a checking tool.

Few existing architectural approaches support mapping of architectures to their implementations. Some, like SRI's SADL [27], provide techniques for refining architectures across levels of abstraction, but do not carry the refinement into implementation. Others, like Rapide, provide a separate executable sublanguage in which systems may be implemented. Darwin [16], UniCon, and Weaves [10] generate "glue" code for composing architectural elements into a system, but assume that individual components will be independently implemented in a traditional PL. Our approach to generating an implementation from an architecture is influenced more directly by domain-specific software architecture (DSSA) approaches: by restricting the software development space to a specific architectural style, reference architecture, and possibly implementation infrastructure, DSSA projects, such as ADAGE [3] and MetaH [38], have been very successful at transferring architectural decisions to running systems. Aesop has been successful in this regard by adopting a similar philosophy: like C2, Aesop provides a class hierarchy for its concepts; this hierarchy, in essence, serves as a domain-specific language for implementing Aesop architectures.

Finally, our work on DRADEL has drawn inspiration from the Inscape environment for software specification, implementation, and evolution [33]. Inscape addressed many of the same issues addressed by DRADEL (scale, evolution, complexity, programming-in-the-large, practicality of the approach), but did so at a level of abstraction below architecture. For example, Inscape requires the semantics of data objects to be modeled, while DRADEL treats them as unelaborated (*basic*) types. Both approaches model operations in terms of pre- and postconditions; Inscape also specifies *obligations*, predicates that must eventually be satisfied after an operation is invoked. Unlike our approach, which uses type-theoretic principles to evolve components, Inscape supports component evolution at a finer level of granularity and in a less systematic manner by adding, removing, and changing pre- and postcondition predicates, and also by changing the implementation itself. Inscape also provides component browsing, retrieval, and version management support. Although it has not been a focus of our work to date, we believe that DRADEL's architecture is flexible enough to include such functionality. Finally, it is unclear whether Inscape could be used reflexively in its own development and evolution.

6 CONCLUSIONS AND FUTURE WORK

Software architectures provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by focusing solely on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools that also must be supported with specific techniques to achieve desired properties. This paper has outlined such techniques and has discussed tool support for evolving software components in a manner that preserves the desired architectural properties and relationships.

These techniques are based on the recognition that software architectures need not always be rigid in establishing properties such as consistency and completeness. For example, it is not always the case that two components that share a communication link can actually communicate (e.g., due to mismatched interfaces). At the architectural level, this mismatch can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into the implemented system, implementation-time decisions may still allow the system to perform at least in a degraded mode. Thus, informing the architect of the potential problem and leaving the decision up to the architect is often preferable to automatically rejecting the option.

Architectures also have a great potential for improving the quality of the resulting software by allowing early analysis. Of course, ensuring specific properties of a system at the level of architecture is of little value unless it can also be ensured that those properties will be preserved in the resulting implementation. Attempting to do so manually is much more error prone than by utilizing a systematic and automated approach.

In this paper, we have presented a simple and practical approach for transferring architecture-level decisions to the implementation. We do not attempt to provide a general solution for system generation. That would essentially reduce architecture-based software development to a variant of transformational programming [32], thus inheriting all of the latter's problems and limitations, with the added problem of scale. Instead, the problem is rendered more tractable by focusing on a particular architectural style and implementation infrastructure. We do, however, have the flexibility to incrementally generalize our support for system generation since DRADEL is evolvable and can easily accommodate additional code generation tools.

DRADEL is a culmination of several years of research in software architectures and ADLs, evolution, and OTS reuse. We believe that the concepts embodied in DRADEL make it a promising candidate for extracting a more general, "reference" architecture for environments for architecture-based development and evolution and, in particular, for transferring architectural decisions to the implementation. We intend to investigate this possibility in the future.

A number of additional issues remain items of future work. We are currently considering several existing theorem provers and model checkers as possible complements to DRADEL's *TypeChecker*: NORA/HAMMR [35], Larch proof assistant (LP) [11], VCR [6], and PVS [30].[6] Operation pre- and postconditions are currently only comments in a generated implementation and serve as guidelines to developers. We intend to promote these comments to assertions that can be checked at system execution time, and are beginning to adapt the appropriate techniques for doing so [34]. Finally, we will fully integrate DRADEL with ArchStudio, our toolset for runtime architecture-based evolution [29]. Other items of future work include investigation of the applicability of our type theory for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, and automating the evolution of existing components to populate partial architectures.

ACKNOWLEDGEMENTS

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Approved for public release — distribution unlimited. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, Springer-Verlag, 1991.
3. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.
4. C. Canal, E. Pimentel, J. M. Troya. A pi-calculus Semantics for an Architecture Description Language. Technical Report,

6. Theorem provers and model checkers typically require some assistance from the user. This is entirely consistent with DRADEL's expectation that the user will play an active role in architecture-based system development and evolution.

LCC-ITI-98-07, Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain, April 1998.

5. K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.
6. B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-Based Software Component Retrieval Tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.
7. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, New Orleans, LA, USA, December 1994.
8. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
9. C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.
10. M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, Austin, TX, May 1991.
11. J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.
12. P. Inverardi and A. L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, April 1995.
13. P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages (COORD '97)*, Berlin, 1997.
14. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
15. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.
16. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.
17. N. Medvidovic. Architecture-Based Specification-Time Software Evolution. Ph.D. Dissertation, University of California, Irvine, December 1998.
18. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.
19. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)* and *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.
20. N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.
21. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Type Theory for Software Architectures. Technical Report, UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.
22. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.
23. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.
24. N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium*, Los Angeles, CA, April 1996.
25. B. Meyer. Applying "Design by Contract." *IEEE Computer*, October 1992.
26. R. Monroe. Armani Language Reference Manual, version 0.1. Private communication, March 1998.
27. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, April 1995.
28. O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, Washington, D.C., USA, October 1993.
29. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.
30. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. *PVS:* Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, eds., *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, July/August 1996, Springer-Verlag.
31. J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, vol. 3, num. 2, 1992.
32. H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, September 1983.
33. D. E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, PA, May 1989.
34. D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, January 1995.
35. J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997.
36. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.
37. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
38. S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.
39. D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.
40. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, October 1997.