A Language–based Approach to Specification and Enforcement of Architectural Protocols

Kevin Bierhoff[‡] **Darpan Saini**[*]
Matthew Kehrt[†] **Majid Al-Meshari**[†]
Sangjin Han[†] **Jonathan Aldrich**[*]

March 2010
CMU-ISR-10-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[‡] Two Sigma Investments LLC, New York, NY, kevin.bierhoff@cs.cmu.edu
[*] Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA.
{darpan.saini, jonathan.aldrich} @ cs.cmu.edu
[†] Formerly: Carnegie Mellon University, Pittsburgh, PA, USA.
mkehrt@cs.washington.edu, meshari@stanford.edu

This technical report supercedes CMU-ISRI-07-121.

This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grants CCF-0546550 and CCF-0811592, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

Keywords: Protocol, typestate, software architecture, ArchJava

Abstract

Software architecture research has proposed using protocols for specifying the interactions between components through ports. Enforcing these protocols in an implementation is difficult. This paper proposes an approach to statically reason about protocol conformance of an implementation. It leverages the architectural guarantees of the ArchJava programming language. The approach allows modular reasoning about implementations with callbacks, recursive calls, and multiple instances of component types. It uses a dataflow analysis to check method implementations and a summary-based interprocedural analysis to reason modularly about component composition. The approach is limited to static architectures but can handle multiple instances for component types and arbitrary nesting of components. We tested the implementation on a case study, and the results suggest that the approach can be scaled to large software applications.

1 Introduction

Sound static reasoning about program behavior is notoriously hard. A variety of approaches have been investigated, each with its own tradeoffs. Static analyses and model checkers typically operate on a global scale (e.g. [15, 19]), suffering from state explosion problems that limit their scaleability to large programs. Furthermore, global analysis inhibits software evolution because replacing a component with a newer version may cause the analysis to fail.

Modular approaches avoid both scalability and evolution problems, but have not yet reached practicality for many classes of software. One approach, based on types, operates at a low level of abstraction and often restricts the programming model (e.g. by prohibiting aliasing) [13]. Modular model checking approaches have been explored, but have practical limitations such as the inability to handle recursive calls between modules (e.g. [9]). All of these modular techniques rely on significant programmer intervention.

A potential way forward builds on software architecture [29], which describes the high-level design of a system as a set of run-time components, their ports (interfaces), and connections between ports. A number of formalisms have been used to describe behavior in architecture description languages (ADLs), including partially ordered event sets in Rapide [25], and process calculi in Wright [3] and Darwin [26]. Rapide included a dynamic analysis that verifies behavior at run time, and model checking in Wright can verify the compatibility of components in a design, but no system exists to verify the behavior of implementations of these architectures statically (at compile time).

In order to verify architectural behavior in an implementation, it is necessary to relate the two. Rapide identified the key conformance property, *communication integrity*: a component can (directly) communicate with another component if *and only if* they are explicitly connected through ports. Rapide pioneered the approach of integrating an architectural specification with code, enabling dynamic verification of communication integrity. Later, the ArchJava language built on the same approach and showed how to verify communication integrity statically [2].

This paper evaluates the hypothesis: *Leveraging an architectural description and the communication integrity property makes it possible to modularly and statically verify architectural behavioral properties in practical implementations.*

We validate the hypothesis above in an extension to the ArchJava language, although we believe our approach is applicable to any system that enforces communication integrity. This paper makes the following contributions:

- A rich specification mechanism in which state transitions occur on method calls and returns (section 2). Unlike prior work, this approach allows developers to properly specify recursive code and callbacks between components. Our specifications support many idioms that are important in practice, such as nondeterministic state transitions, and method constraints that refer to the state of other ports.

- An extension of the ArchJava language to describe architectural behavior using a variant of typestate [30] (section 3). Each port defines a fixed set of states and constraints method calls to occur in appropriate states.

- A two-part approach for statically verifying behavior: a static analysis verifies that each component's implementation conforms to its behavioral interface (section 4), and then a compatibility check verifies that a composition of multiple components will not violate any of the components' constraints (section 5). Our approach has three key properties:

  - Modularity: each component is analyzed independently, using only the interfaces of other components. Modifications to a component that do not change its interface cannot cause system verification to fail.
  - Compositionality: our analysis verifies that a component uses its subcomponents correctly, assuming that it is used correctly by the environment. This style of analysis allows us to verify hierarchical architectures of arbitrary depth.
  - Scaleability: the combination of modularity and compositionality means that the overall approach can verify the behavior of architectures of arbitrary size at linear cost. We can do this assuming only that the code be split into components of size no greater than static analysis can handle, and that architectural hierarchy is used to limit the number and complexity of components at any level of abstraction. Our case study suggests that these limits are reasonable.

  Like prior work, our system is limited to static architectures, but unlike prior work our approach supports multiple instances of the same component type in the same architecture.

- An evaluation of the approach which specifies and verifies the architectural behavior of Hillclimber, a moderate sized ArchJava application (section 6). Our evaluation demonstrates that the technique is feasible and can find inconsistencies between the specification and code.

The remainder of this paper is organized as follows. We introduce our approach to the specification of port protocols in section 2. Section 3 lays out a core language that includes these protocols. Static protocol checking of method implementations is formalized in section 4. Section 5 investigates modular component composition checking. Extensions to support protocols in a realistic language are discussed in section 6. Section 7 summarizes related work and section 8 concludes.

2 Port Protocol Specification

This section gives a high-level introduction to the specification of port protocols. We first motivate our approach with an example ArchJava program. We then discuss expressiveness goals and show how we achieve these in our approach.

2.1 Motivation and Example

Listing 1 shows a legal ArchJava component class that implements the front-end of a simple Web server. It has three ports, `Http`, `Control`, and `Handle`. The `Http` port encapsulates the client

interface of the Web server while the other two ports can be hooked up to other components that help servicing incoming requests. The method `Http.get` implements the actual service. It takes an HTTP request, prepares the `Control` port, forwards the request to the `Handle` port and finally tears down `Control` after the request is serviced.

Compared to a standard Web server implementation in C or Java, our implementation in ArchJava has the advantage that it makes its *ports* explicit. This Web server component has three points of interaction with other components, and it lists (exhaustively) all methods that it can call (`requires`) and that can be called (`provides`). Software architecture models were designed to capture this kind of information [29]. ArchJava includes these concepts in a programming language [2].

A number of protocols are implicit in our Web server implementation. Firstly, the Web server is not reentrant. It assumes that only one request is serviced at a time. Secondly, the `Control` port has its own small protocol that requires `prepare` and `teardown` to be called in alternating order. Thirdly, it is required that `Handle.request` is only called after `Control` was prepared (and before tear down). All these make *assumptions* about components connected to the one shown in listing 1. For instance, the component assumes that clients wait with a new request until the last one was answered. Moreover, the implementation *guarantees* that it will indeed follow these protocols and for example only ask the `Handle` port to service the request after some preparation.

Notice that these protocols cannot be extracted from the source code. They are followed by the Web server implementation, but this could be mere coincidence. Maybe it is really no problem to forget to call `teardown`, or no preparation is necessary for servicing a request. In general, protocols have to be documented informally [8] and it is by no means guaranteed that these protocols are observed or even correct [24]. Moreover, the users of a component usually will not explicitly document the assumptions they make about that component. This makes it hard or impossible to decide whether the system will still work if the component is replaced by a different one.

2.2 Typestate Protocols

Our goal is to document and enforce these protocol assumptions and guarantees in a way that does not overburden developers. In contrast to existing work on reasoning about architectural protocols [3, 26] we tie protocols to the implementation in a programming language. Our approach can therefore *statically guarantee* the protocol conformance of that implementation. This section describes our specification approach and its expressiveness. The following sections deal with enforcing these protocols in an implementation.

We build on earlier work on protocol definition for programming languages [12, 13, 8]. We leverage the concept of typestate [30] to specify a state machine that defines a protocol for each port (listing 1). By contrast, research on architectural protocols [3, 26] usually defined protocols in a form of process calculus such as CSP [21]. These are then translated into finite state models to apply model checking techniques. We avoid this extra translation step by using typestates.

Typestates give the developer the opportunity to explicitly name states. In our experience states often have a semantic meaning such as, “the Web server is ready to service a request”, that can be conveyed with the state name [8]. Moreover, typestates are an abstraction that lets the developer think about pre- and post-conditions for each operation separately. With a process

Method spec	S	::=	T	*single case*
		|	T , S	*multiple cases*
Method case	T	::=	B => U	*state transition*
Method boundary	B	::=	t	*no side condition*
		|	$t \wedge c$	*with side condition*
Postcondition	U	::=	B	*single case*
		|	$B \vee U$	*disjunction*
Transition	t	::=	s_1 -> s_2	*boundry transition*
Conditions	c	::=	$z.s$	*state on port*
		|	$z.s \wedge c$	*condition conjunct*
states	s			
ports	z			

Figure 1: Core method specifications

model, operations are interdependent in that protocols are defined as possible event sequences. In our approach, possible event sequences are implied by states shared between post-conditions and pre-conditions of operations.

Listing 1 includes a typestate-based specification of the protocols that we described in the preceding section. Notice that protocols are enclosed with `/*: ... */` and are therefore technically comments that can be ignored by the compiler. As can be seen from the example, we use two kinds of protocol annotations. `/*:states */` annotations define a list of states for a port. `/*:spec */` annotations can be added to a provided or required method to define its protocol with *state transitions*. A state transition defines the behavior of a method with a pre-condition and a post-condition expressed as states [8]. The following paragraphs describe our specification approach in detail. The exact language for method specifications is shown in Figure 1.[1]

States for each port. We associate with each port a set of mutually exclusive states. They are defined as a simple list. For example, the three Web server ports define two states each.

States as abstract tokens. We track the current state of each port as an abstract token (as in Vault [12]). The state does not have a representation in form of a predicate over component fields (as in Fugue [13]). In fact, our states do not have a runtime representation at all.

[1] We write `&` for $\wedge$ and `|` for $\vee$ in example code.

State transition during method execution. During method execution the port can potentially change state. We denote the expected state transition during method execution with a *big arrow* (`=>`). We do not specify how this transition is accomplished. The port can go through an arbitrary number of states during the method execution. For example, the `Control` port defines that `prepare` will ultimately transition from `raw` to `initialized` (the full meaning of this specification will become clear soon).

Boundary transitions. Most previous typestate specification mechanisms describe only how a method changes the state of a component with respect to clients [12]. However, if the implementation of a method called back to a client method, then the client could make another call into the component, raising the question: what state is the component in as its method executes?

In our model, state transitions occur atomically at method call and return points. These *boundary transitions* are declared with a *small arrow* (`->`) on each side of the big arrow (`=>`). For example, `Http.get` declares boundary transitions from `idle` to `busy` and back, expressing that `get` is not re-entrant. Not only does this allow us to soundly handle callbacks, it allows us to reason about them more abstractly than solutions such as packing and unpacking objects [13].

Method cases and non–determinism. Methods commonly behave differently in different contexts [8]. We allow specifying multiple *method cases* that describe the method's behavior under different pre-conditions. Formally a specification is then an intersection [14] of cases. To accommodate non–determinism during method execution, the post-condition of a method case is a union (disjunction) [14] of final states. Our Web server example does not exhibit non–determinism, but other examples of architectural protocols do so [4].

Port dependencies. Methods of one port will frequently depend on particular states of other ports so that they can call methods on those. This is not always supported by architectural protocols. For instance, a Wright connector specifies its roles completely separately [3]. Our protocols include the definition of dependencies. The specification for the current port is combined (intersected [14]) with state assumptions (in pre-conditions) and guarantees (in post-conditions) on other ports. This has happened in every single method specification for our Web server. For example, the specification for `Handle.request` makes explicit the expectation that the `Control` port is prepared first. Notice that only the port the method belongs to can perform boundary transitions.

Syntactic sugar. To alleviate the developer from some of the protocol specification burden we introduce several shorthand notations. Sometimes we do not care about the state (or states) that are visited during method execution, as in the `Control` port. As far as we are concerned, we cannot do anything with that port while one of its methods is running (and its methods cannot call back).

We therefore support syntactic sugar to omit small arrows. If there is no explicit small arrow in the pre-condition then a boundary transition to a fresh *internal state* will be inserted. In this case the post-condition should not contain a small arrow, either, so that a switch back from the internal state can be added. For example, the `raw => initialized` specification in `Control.prepare` is translated into `raw -> t => t -> initialized`, where t is a fresh state. Notice how

the specifications in the `Control` port formalize that the two methods `prepare` and `teardown` have to be called in alternating order.

If only the small arrow in the post-condition is omitted then the state after executing the method is assumed to be the same as before. A state switch from the right-hand side of the pre-condition to the state given in the post-condition is added in this case. Thus the specification for `Handle.request` expresses that a call to that method switches the state to `working` and the method return switches it back to `waiting`.

Method cases without any arrows are assumed to preserve the given state with a transition to an internal state during method execution. Ports that are not mentioned in a method case are assumed to preserve state. The latter is exemplified by the `Control` and `Handle` ports that do not mention the `Http` port. The exact rules for desugaring surface protocol specifications into the syntax of figure 1 are given in appendix A.

2.3 Implementation

We implemented a prototype that can read and check specifications for consistency with the implementation. Our implementation can handle the Web server example discussed above. It is an add–on to the regular ArchJava compiler. This extension is *optional*: protocols have no run time impact, the protocol checks can be switched off (or ignored), and protocols do not interfere with ArchJava's structural type system [2]. However, a successful protocol check gives a positive assurance of consistency between implementation and behavioral specification. The following sections build up the technical facilities for statically checking specifications and in particular the example given in listing 1.

3 A Core Language

In order to facilitate our reasoning about the correctness of ArchJava programs with respect to protocols we formalize a core fragment of static ArchJava with protocols. The following subsections discuss syntax, dynamic semantics, and typechecking of this core language. The design follows ArchFJ [2], a core language for ArchJava.

3.1 Syntax

The syntax is summarized in figure 2. We distinguish component classes from normal classes with the keyword `component`. C ranges over normal classes, D over component classes, and E over both kinds of classes. Normal classes are defined just as in ArchJava (and Featherweight Java [22]). Component classes are defined with a list of fields (which can be subcomponents or normal objects), a constructor, a list of ports, and a list of (static) connections. Connections hook up matching ports of two components. The ports that are connected have to be part of the current component (`this`) or a direct subcomponent referenced by a field. Notice that we use overbars to denote lists; for instance, $\overline{E\ f} = \mathtt{E_1\ f_1; E_2\ f_2; \ldots; E_n\ f_n}$ defines the list of fields in a component.

$$
\begin{array}{rrcl}
 & CP & ::= & \texttt{component class } \mathtt{D}_1 \texttt{ extends } \mathtt{D}_2 \texttt{ \{ } \overline{\mathtt{E}\ \mathtt{f}}\texttt{; } \mathtt{K}\ \overline{\mathtt{P}}\ \overline{\mathtt{X}}\texttt{ \}} \\
 & CL & ::= & \texttt{class } \mathtt{C}_1 \texttt{ extends } \mathtt{C}_2 \texttt{ \{ } \overline{\mathtt{C}\ \mathtt{f}}\texttt{; } \mathtt{K}\ \overline{\mathtt{M}}\texttt{ \}} \\
\textit{constructor} & K & ::= & \mathtt{E}(\overline{\mathtt{E}\ \mathtt{f}})\texttt{ \{ super}(\overline{\mathtt{f}})\texttt{; } \overline{\texttt{this.f} = \mathtt{f}}\texttt{; \}} \\
\textit{method} & M & ::= & \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\texttt{ \{ return e; \}} \\
 & P & ::= & \texttt{port z \{ [states } \overline{\mathtt{s}}\texttt{; } \overline{\mathtt{Q}}\texttt{] } \overline{\mathtt{R}}\texttt{\}} \\
 & Q & ::= & \texttt{requires S C m}(\overline{\mathtt{C}\ \mathtt{x}})\texttt{;} \\
 & R & ::= & \texttt{provides S M} \\
 & X & ::= & \texttt{connect}(\mathtt{w}_1.\mathtt{z}_1, \mathtt{w}_2.\mathtt{z}_2)\texttt{;} \\
\textit{expressions} & e & ::= & \texttt{x} \\
 & & | & \texttt{new E}(\overline{\mathtt{e}}) \\
 & & | & \texttt{e.f} \\
 & & | & \texttt{z.m}(\overline{\mathtt{e}}) \\
 & & | & \mathtt{e}_1.\mathtt{f} = \mathtt{e}_2 \\
 & & | & \texttt{e.m}(\overline{\mathtt{e}}) \\
\textit{paths} & w & ::= & \texttt{this} \\
 & & | & \texttt{this.}f \\
\textit{types} & E & ::= & C \\
 & & | & D \\
\textit{variables} & x & & \\
\textit{fields} & f & & \\
\textit{classes} & C & & \\
\textit{components} & D & &
\end{array}
$$

Figure 2: Core language syntax

Ports are ranged over with z and define a list of states, a list of required methods and a list of provided methods. Both required and provided methods are annotated with a specification S (figure 1) of how they change the port's state. Notice that all component methods reside in a port. Method bodies consist of a single return statement with a recursive expression e. Legal expressions are variable access (`this` is a special variable for the receiver), `new` expressions to create new objects or components, field access, assignment, and method invocation. Method invocation is allowed on the component's own ports ($z.m$) and on objects ($e.m$).[2] Explicit casting of objects is omitted to simplify the system; it could be added without complications.

3.2 Dynamic Semantics

The dynamic semantics is largely standard and similar to ArchFJ [2]. We use a store that maps locations to objects. Objects are tagged with their runtime type and contain a list of locations for their fields.

$$\textit{Stores} \quad \mu \quad ::= \quad \bullet \mid \mu, l \mapsto E(\bar{l})$$

We write $\mu[l \mapsto E(\bar{l})]$ for a store that is identical to μ except for location l which now points to the given object.

A small–step evaluation semantics is given in figure 3. It uses the judgment $\theta \vdash \langle \mu, e \rangle \longmapsto \langle \mu', e' \rangle$. This means that in the context of receiver object θ (identified by its location) and a store μ, an expression e evaluates to e' and changes the store to μ' in one step. Auxiliary judgments for evaluation are presented in figures 4 and 6.

We track the receiver during evaluation in order to determine the callee of a port method call in rule E-PORTCALL with the judgment connected (figure 4). In order to track all receivers in a call stack we introduce the following additional syntactic form that only occurs during evaluation and represents a kind of stack frame. Locations are also expressions and represent the only values in our system.

$$\textit{Expressions} \quad e \quad ::= \quad \ldots \mid l \rhd e \mid l$$

The rules E-OBJCALL and E-PORTCALL generate a frame for every method call. Rule E-CFRAME then evaluates expression e in the context of the new receiver l defined in the frame. Finally, rule E-FRAME removes the frame once its expression evaluated to a value. This corresponds to a return from a method call.

Congruence rules are summarized with E-CONGRUENCE. We define evaluation contexts in the obvious way.

$$\begin{array}{lrcl} \textit{Eval. contexts} & \Xi[\bullet] & ::= & \bullet \mid \texttt{new}\ E(\bar{l}, \Xi[\bullet], \bar{e}) \mid \Xi[\bullet].f \\ & & \mid & \Xi[\bullet].m(\bar{e}) \mid l.m(\bar{l}, \Xi[\bullet], \bar{e}) \mid z.m(\bar{l}, \Xi[\bullet], \bar{e}) \end{array}$$

$$\frac{l^* \notin \mathsf{dom}\ \mu \quad \mu' = \mu[l \mapsto E(\bar{l})]}{\theta \vdash \langle \mu, \mathtt{new}\ E(\bar{l}) \rangle \longmapsto \langle \mu', l^* \rangle}\ \text{E-NEW} \qquad \frac{\mu(l_0) = E_0(\bar{l}) \quad \mathsf{fields}(E_0) = \overline{E\ f}}{\theta \vdash \langle \mu, l_0.f_i \rangle \longmapsto \langle \mu, l_i \rangle}\ \text{E-FIELD}$$

$$\frac{\mu(l_0) = E_0(\bar{l}) \quad fields(E_0) = \overline{E\ f} \quad \mu' = \mu[l_0 \mapsto E_0(l_1, \ldots, l_{i-1}, l', l_{i+1}, \ldots, l_n)]}{\theta \vdash \langle \mu, l_0.f_i = l' \rangle \longmapsto \langle \mu', l' \rangle}\ \text{E-ASSIGN}$$

$$\frac{\mu(l) = C(\bar{l}) \quad \mathsf{mbody}(m, C) = \overline{x}.e}{\theta \vdash \langle \mu, l.m(\bar{l}) \rangle \longmapsto \langle \mu, l \rhd [\bar{l}/\overline{x}, l/\mathtt{this}]e \rangle}\ \text{E-OBJCALL}$$

$$\frac{\mathsf{connected}(\mu, \theta, z) = l \quad \mu(l) = D(\bar{l}) \quad \mathsf{mbody}(m, D) = \overline{x}.e}{\theta \vdash \langle \mu, z.m(\overline{v}) \rangle \longmapsto \langle \mu, l \rhd [\bar{l}/\overline{x}, l/\mathtt{this}]e \rangle}\ \text{E-PORTCALL}$$

$$\frac{}{\theta \vdash \langle \mu, l' \rhd l \rangle \longmapsto \langle \mu, l \rangle}\ \text{E-FRAME} \qquad \frac{l \vdash \langle \mu, e \rangle \longmapsto \langle \mu', e' \rangle}{\theta \vdash \langle \mu, l \rhd e \rangle \longmapsto \langle \mu', l \rhd e' \rangle}\ \text{E-CFRAME}$$

$$\frac{\theta \vdash \langle \mu, e \rangle \longmapsto \langle \mu', e' \rangle}{\theta \vdash \langle \mu, \Xi[e] \rangle \longmapsto \langle \mu', \Xi[e'] \rangle}\ \text{E-CONGRUENCE}$$

Figure 3: Small–step evaluation semantics

3.3 Typechecking

This section discusses the static typechecking rules for our core language. A program consists of the class table CT, i.e. the list of all normal and component classes declared, and a main expression. Figure 5 contains rules for typechecking expressions and declarations. We discuss component subclassing separately in section 7.3. The judgment conforms in is the starting point for protocol conformance checking as presented in the following section. Our expression typing judgment $\Gamma \vdash_E e : C$ includes the type E of the receiver and is otherwise similar to Featherweight Java [22]. Variable contexts are defined in the standard way as follows.

$$\textit{Contexts} \quad \Gamma \quad ::= \quad \bullet \mid \Gamma, x : E$$

We explain the expression typing rules in turn.

- T-VAR is the standard rule for variable access that looks up the variable's type in the context.
- T-FIELD types field accesses. The type of the field is looked up in the class's declaration.
- T-NEW creates a new object or component. To simplify our system we assume that any subcomponents that are passed in as parameters to a new component are freshly created with

[2]In full ArchJava, components may invoke methods of subcomponents directly. We simulate this idiom with an explicit port connected to the subcomponent. We similarly simulate internal methods of the component by calling methods provided by own ports.

Method body lookup

$$\frac{[\texttt{component}]\ \texttt{class E extends E}'\ \{\ldots\} \in CT \quad m \textit{ not defined in } E \quad \mathsf{mbody}(m, E') = \overline{x}.e}{\mathsf{mbody}(m, E) = \overline{x}.e}$$

$$\frac{[\texttt{component}]\ \texttt{class E}\ \ldots\{\ldots\ \texttt{C m}(\overline{\texttt{C x}})\ \{\ \texttt{return e;}\ \}\ldots\} \in CT}{\mathsf{mbody}(m, E) = \overline{x}.e}$$

Find connected component

$$\frac{\mu(l) = D(\bar{l}) \quad \mathsf{connects}(D) = \overline{X} \quad \texttt{connect this.z}, \texttt{this.f}_\texttt{i}\texttt{.z}' \in \overline{X}}{\mathsf{connected}(\mu, l, z) = l_i}$$

$$\frac{\mu(l) = D(\bar{l}) \quad \mathsf{connects}(D) = \overline{X} \quad \texttt{connect this.z}, \texttt{this.z}' \in \overline{X}}{\mathsf{connected}(\mu, l, z) = l}$$

$$\frac{\mu(l_0) = D_0(\bar{l}) \quad (l = l_i) \quad \mathsf{connects}(D_0) = \overline{X} \quad \texttt{connect this.f}_\texttt{i}\texttt{.z}, \texttt{this.z}' \in \overline{X}}{\mathsf{connected}(\mu, l, z) = l_0}$$

$$\frac{\mu(l_0) = D_0(\bar{l}) \quad (l = l_i) \quad \mathsf{connects}(D_0) = \overline{X} \quad \texttt{connect this.f}_\texttt{i}\texttt{.z}, \texttt{f}_\texttt{j}\texttt{.z}' \in \overline{X}}{\mathsf{connected}(\mu, l, z) = l_j}$$

Connection lookup

$$\frac{}{\mathsf{connects}(\textit{Object}) = \bullet}$$

$$\frac{\texttt{component class D extends D}'\ \{\ \overline{\texttt{E f}};\ \texttt{K}\ \overline{\texttt{P}}\ \overline{\texttt{X}}\ \} \in CT \quad \mathsf{connects}(D') = \overline{X'}}{\mathsf{connects}(D) = \overline{X'}, \overline{X}}$$

Figure 4: Auxiliary judgments for evaluation

$$\frac{(x \in \mathsf{dom}\ \Gamma)}{\Gamma \vdash_E x : \Gamma(x)}\ \text{T-VAR} \qquad \frac{\Gamma \vdash_E e_0 : E_0 \quad (\mathsf{fields}(E_0) = \overline{E\ f})}{\Gamma \vdash_E e_0.f_i : E_i}\ \text{T-FIELD}$$

$$\frac{\Gamma \vdash_E \overline{e : E'} \quad \Gamma \vdash_E \overline{e : D} \Rightarrow \overline{e = \texttt{new}\ D(\ldots)\ \textit{or}\ e = l} \quad \mathsf{fields}(E_0) = \overline{E\ f} \quad (\overline{E' <: E})}{\Gamma \vdash_E \texttt{new}\ E_0(\overline{e}) : E_0}\ \text{T-NEW}$$

$$\frac{\Gamma \vdash_E e_0 : E_0 \quad \Gamma \vdash_E e : C' \quad \mathsf{fields}(E_0) = \overline{E\ f} \quad (C = E_i) \quad (C' <: C)}{\Gamma \vdash_E e_0.f_i = e : C}\ \text{T-ASSIGN}$$

$$\frac{\Gamma \vdash_D \overline{e : C'} \quad (\mathsf{mtype}(m, D) = \overline{C} \to C) \quad (\overline{C' <: C})}{\Gamma \vdash_D z.m(\overline{e}) : C}\ \text{T-PORTCALL}$$

$$\frac{\Gamma \vdash_E e_0 : C_0 \quad \Gamma \vdash_E \overline{e : C'} \quad (\mathsf{mtype}(m, C_0) = \overline{C} \to C) \quad (\overline{C' <: C})}{\Gamma \vdash_E e_0.m(\overline{e}) : C}\ \text{T-OBJCALL}$$

$$\frac{\begin{array}{c}\overline{P}\ \mathsf{ok\ in}\ D_1\ \mathsf{ext}\ D_2 \quad \overline{X}\ \mathsf{ok\ in}\ D_1 \quad \mathsf{fields}(D_2) = \overline{E'\ g} \\ K = \texttt{D}_1(\overline{\texttt{E}'\ \texttt{g}}; \overline{\texttt{E}\ \texttt{f}})\ \{\ \texttt{super}(\overline{\texttt{g}});\ \overline{\texttt{this.f} = \texttt{f};}\ \}\end{array}}{\texttt{component class D}_1\ \texttt{extends D}_2\ \{\ \overline{E\ f};\ K\ \overline{P}\ \overline{X}\ \}\ \mathsf{ok}}\ \text{T-COMP}$$

$$\frac{\overline{S\ M}\ \mathsf{typechecks\ in}\ D \quad \overline{S\ M}\ \mathsf{conforms\ in}\ D.z}{\texttt{port z}\ \{\ \texttt{states}\ \overline{\texttt{s}};\ \overline{\texttt{Q}}\ \overline{\texttt{provides S M}}\ \}\ \mathsf{ok\ in}\ D\ \mathsf{ext}\ \textit{Object}}\ \text{T-PORT}$$

$$\frac{\begin{array}{c}\overline{M}\ \mathsf{typechecks\ in}\ C_1 \quad \mathsf{fields}(C_2) = \overline{C'\ g} \\ K = \texttt{C}_1(\overline{\texttt{C}'\ \texttt{g}}; \overline{\texttt{C}\ \texttt{f}})\ \{\ \texttt{super}(\overline{\texttt{g}});\ \overline{\texttt{this.f} = \texttt{f};}\ \}\end{array}}{\texttt{class C}_1\ \texttt{extends C}_2\ \{\ \overline{C\ f};\ K\ \overline{M}\ \}\ \mathsf{ok}}\ \text{T-CLASS}$$

$$\frac{\overline{x : C}, \texttt{this}\ : E \vdash_E e : C' \quad (C' <: C) \quad \mathsf{override}(m, E, [S]\ \overline{C} \to C)}{[S]\ C\ m(\overline{C\ x})\ \{\ \texttt{return}\ e;\ \}\ \mathsf{typechecks\ in}\ E}\ \text{T-METH}$$

Figure 5: Expression and declaration typechecking

a `new` expression of their own. This is equivalent to field initializers in full ArchJava and ensures that components are not shared by multiple parents in the architecture. Notice that normal classes cannot have fields of component type (rule T-CLASS).

- T-ASSIGN types assignment expressions in the way Java does. Notice that only fields with normal objects can be assigned new values. This ensures that a component composition is not modified after its creation.

- T-PORTCALL typechecks method invocations on ports. This rule therefore only applies to components. After checking the method arguments, we look up the declared type of the method to be invoked, where $\overline{C}$ are the formal method argument types and C is the declared result type.

- T-OBJCALL typechecks method invocations on regular objects. Notice that we require the receiver to be an object rather than a component. This ensures that components cannot call methods on other components directly but have to go through a port. Otherwise, typechecking proceeds analogously to T-PORTCALL.

The override judgment for T-METH is shown in figure 9. The helper judgments fields and mtype are similar to ArchFJ [2], as is checking of connections with $\overline{X}$ ok in D (figure 6).

Figure 6 contains additional rules to typecheck programs. We check a complete program by checking all normal and component classes as well as the main expression (T-PROGRAM). We allow a call to a provided port method right after construction of a component (T-INITCALL). This is necessary to enter the first component in the main expression. We include a rule to typecheck frames (see preceding section) for completeness (T-FRAME).

Technically we need a standard store typing environment for typing locations that we omit here. Subtyping is defined as the reflexive transitive closure of the `extends` relation with root type *Object* and also omitted. For details on these issues see the formalization of ArchJava into ArchFJ [2]. The core language defined here preserves communication integrity as proved for ArchJava [2].

4 Implementation Checking

This section shows how method implementations can be statically checked for protocol conformance. What we would like to ensure beyond normal typechecking concerns is that all communication across a port observes the protocol specified for that port. In other words, the port has to be in an appropriate state when a method is called, and we can then assume that the port is left in the specified state when the method terminates. This allows us to check the validity of the next method call.

Communication integrity makes it possible to reason about protocol conformance locally. Communication integrity guarantees that control flow in a component always starts at a provided port method. When normal object methods are invoked, calls back into the component are impossible. Conversely, if control flow leaves the component through a required port method, callbacks into

Program and additional expression typing

$$\frac{\Gamma \vdash_E e_0 : D \quad e_0 = \texttt{new D}(\ldots) \textit{ or } e_0 = l \quad \Gamma \vdash_E \overline{e : C'} \quad \mathsf{mtype}(m, D) = \overline{C} \to C \quad (\overline{C' <: C}) \quad m \textit{ defined in port } z \quad D.z \vdash \mathsf{spec}(m, D) \cdot \mathsf{start}(D) \hookrightarrow p}{\Gamma \vdash_E e_0.m(\overline{e}) : C} \text{ T-InitCall}$$

$$\frac{\Gamma \vdash_E l : E' \quad \Gamma \vdash_E e : E''}{\Gamma \vdash_E l \rhd e : E''} \text{ T-Frame} \qquad \frac{\overline{CP}\ \mathsf{ok} \quad \overline{CL}\ \mathsf{ok} \quad \bullet \vdash e : E}{(\overline{CP}\ \overline{CL}, e)\ \mathsf{ok}} \text{ T-Program}$$

Connection typechecking

$$\frac{\mathsf{resolve}(D, w_1, D_1) \quad \mathsf{resolve}(D, w_2, D_2) \quad D_1 \textit{ does not otherwise connect } z_1 \quad \textit{all methods required in } D_1.z_1 \textit{ are provided in } D_2.z_2 \quad D_j \textit{ does not otherwise connect } z_2 \quad \textit{all methods required in } D_2.z_2 \textit{ are provided in } D_1.z_1}{\texttt{connect } \mathtt{w_1.z_1}, \mathtt{w_1.z_2}\ \mathsf{ok\ in}\ D} \text{ T-Connect}$$

Auxiliary judgments

$$\overline{\mathsf{fields}(\mathit{Object}) = \bullet}$$

$$\frac{[\texttt{component}]\ \texttt{class E}_1\ \texttt{extends E}_2\ \{\ \overline{\texttt{E f}};\ \texttt{K}\ \overline{\texttt{P}}\ \overline{\texttt{X}}\ \} \in CT \quad \mathsf{fields}(E_2) = \overline{E'\ g}}{\mathsf{fields}(E_1) = \overline{E'\ g}, \overline{E\ f}}$$

$$\frac{[\texttt{component}]\ \texttt{class E extends E}'\ \{\ldots\} \in CT \quad m \textit{ not defined in } E \quad \mathsf{mtype}(m, E') = \overline{C} \to C}{\mathsf{mtype}(m, E) = \overline{C} \to C}$$

$$\frac{[\texttt{component}]\ \texttt{class E}\ \ldots\{\ldots\ \texttt{C m}(\overline{\texttt{C x}})\ldots\} \in CT}{\mathsf{mtype}(m, E) = \overline{C} \to C}$$

$$\frac{}{\mathsf{resolve}(D, \texttt{this}, D)} \qquad \frac{\mathsf{fields}(D) = \overline{E\ f} \quad E_i \textit{ is component class}}{\mathsf{resolve}(D, \texttt{this.f}_\texttt{i}, E_i)}$$

$$\frac{\texttt{component class D extends D}'\ \{\ldots\overline{\texttt{P}}\ldots\} \in CT \quad \overline{P} = \overline{\texttt{port z}\ \{[\texttt{states s}^*, \overline{\texttt{s}}]\ldots\}} \quad c = \textstyle\bigwedge_i z_i.s_i^* \quad [\mathsf{start}(D') = c']}{\mathsf{start}(D) = c[\wedge c']}$$

Figure 6: Additional typechecking rules

$$\frac{}{p \vdash_D x \dashv p}\ \text{P-VAR} \qquad \frac{p \vdash_D e \dashv p'}{p \vdash_D e.f \dashv p'}\ \text{P-FIELD}$$

$$\frac{p \overline{\vdash e \dashv} p'}{p \vdash_D \texttt{new}\ E(\overline{e}) \dashv p'}\ \text{P-NEW} \qquad \frac{p \vdash_D e_1 \dashv p' \vdash_D e_2 \dashv p''}{p \vdash_D e_1.f = e_2 \dashv p''}\ \text{P-ASSIGN}$$

$$\frac{p \vdash_D e_0 \dashv p' \quad p' \overline{\vdash_D e \dashv} p'' \quad (e_0 \neq \texttt{this})}{p \vdash_D e_0.m(\overline{e}) \dashv p''}\ \text{P-OBJCALL}$$

$$\frac{p \overline{\vdash_D e \dashv} p' \quad D.z \vdash S \cdot p' \hookrightarrow p'' \quad (\mathsf{spec}(m, D) = S)}{p \vdash_D z.m(\overline{e}) \dashv p''}\ \text{P-PCALL}$$

$$\frac{T\ M \text{ conforms in } D.z \quad S\ M \text{ conforms in } D.z}{T,\ S\ M \text{ conforms in } D.z}\ \text{P-CASES}$$

$$\frac{\mathsf{right}(B, D.z) \vdash_D e \dashv p' \quad p' \Rightarrow \mathsf{left}(U, D.z)}{B \texttt{ => } U\ C\ m(\overline{C\ x})\ \{\ \texttt{return}\ e;\ \} \text{ conforms in } D.z}\ \text{P-METH}$$

Figure 7: Core protocol checking rules

the component are possible before the call returns, but only through ports. Intuitively, this is why we can check each provided port method separately in a manner very much like typechecking.

One of the benefits of our approach is that we can reason about state dependencies between components even if they are shared with other components. We are limited to static architectures but in exchange we can handle arbitrary callbacks and recursion between shared components. This is in contrast to invariant verification systems like Fugue [13] or Boogie [6] where an object can only depend on objects it owns, making it difficult to handle callbacks.

Because the states of ports are treated as explicit tokens our checking algorithm has to track these tokens through the method implementation. We emphasize that this does *not* happen at run time but rather at compile time. In other words, the compiler maintains symbolic information about the states of ports that it uses for checking method invocations. Because of communication integrity this information is sound, i.e. a conservative approximation of the runtime behavior. In our approach, the ArchJava typechecker guarantees communication integrity [2].

Figure 5 illustrates our approach for checking components (rule T-COMP). We check each port definition and separately reason about component composition with port connections. When checking a port definition (rule T-PORT) we distinguish normal typechecking of provided method bodies from checking protocol conformance. This lets us treat protocol conformance checking as an orthogonal add-on to ArchJava typechecking.

In this section, we are only concerned with checking protocol conformance of expressions. Our approach is shown in figure 7. Protocol conformance checking proceeds for each specification case separately (rule P-CASES). Notice that a method implementation has to conform to *all* cases

$$\frac{c \Rightarrow \mathsf{left}(B, D.z) \quad (p = \mathsf{right}(U, D.z))}{D.z \vdash B \Rightarrow U \cdot c \hookrightarrow p} \qquad \frac{D.z \vdash T \cdot c \hookrightarrow p \quad D.z \vdash S \cdot c \hookrightarrow p'}{D.z \vdash T\texttt{,}\ S \cdot c \hookrightarrow p \vee p'}$$

$$\frac{D.z \vdash T \cdot c \hookrightarrow p \quad D.z \vdash S \cdot c \not\hookrightarrow}{D.z \vdash T\texttt{,}\ S \cdot c \hookrightarrow p} \qquad \frac{D.z \vdash S \cdot c \hookrightarrow p \quad D.z \vdash T \cdot c \not\hookrightarrow}{D.z \vdash T\texttt{,}\ S \cdot c \hookrightarrow p}$$

$$\frac{D.z \vdash S \cdot c \hookrightarrow p' \quad D.z \vdash S \cdot p \hookrightarrow p''}{D.z \vdash S \cdot c \vee p \hookrightarrow p' \vee p''}$$

Figure 8: Deterministic method call algorithm

defined within a method specification S. Conformance checking for a method case $B \Rightarrow U$ then proceeds by assuming the port states immediately following method entry as indicated by B, tracking effects of port method calls within the method body e (as discussed below) and verifying that the states reached immediately prior to method exit imply what is specified in U (rule P-METH).

For reasoning about protocol conformance of (well-typed) expressions we use the judgment $p \vdash_D e \dashv p'$. In this judgment, p is a predicate that describes the states of ports defined for component D *before* considering expression e. The predicate p' indicates the states of D's ports *after* evaluating e. Predicates are disjunctions of port state conjunctions defined as follows.

$$\begin{array}{rrcll} \textit{Predicates} & p & ::= & c & \mid\ c \vee p \\ \textit{Conjuncts} & c & ::= & z.s & \mid\ z.s \wedge c \end{array}$$

The protocol conformance rules track state changes in the order of evaluation. We discuss each rule in turn.

- P-VAR defines that variable access has no effect on states.
- P-FIELD determines the state changes during evaluation of the object expression whose field is accessed. The field access itself does not change states.
- P-NEW tracks state changes during object and component construction. The `new` expression itself does not change any states. The notation $p \overline{\vdash_D e \dashv} p'$ is a shorthand for $p \vdash_D e_1 \dashv p_1 \vdash_D e_2 \dashv p_2 \vdash_D \ldots \dashv p_{n-1} \vdash_D e_n \dashv p'$. This tracks state changes during the evaluation of arguments $e_1, \ldots, e_n$ with initial state p in order and yields the final state p'.
- P-ASSIGN threads state changes through the left-hand side and the right-hand side of an assignment.
- P-OBJCALL tracks state changes through the evaluation of receiver and arguments of a method call on a normal object.

- P-PCALL is the core rule of our checking system. We are checking the state changes that result from a call to a port method with $z.m(\overline{e})$. In the spirit of P-NEW, we first consider the method arguments one by one. We then determine the effect of executing $z.m$ given the state predicate p'. The helper function spec looks up the method specification S. $S \cdot p'$ implements a deterministic algorithm to determine the states of the component's ports after the execution of $z.m$ assuming its specification S (see below). If $S \cdot p'$ does not yield a predicate p'', the method call is invalid, and the compiler will issue an error. Otherwise, p'' is the final result of our reasoning about a port method call.

Our judgment to determine the effect of a method call for a given state predicate is $D.z \vdash S \cdot p \hookrightarrow p'$. $D.z$ is the port on which the method call occurs. S is the specification we consider (see figure 1). p is the state predicate we assume. Then p' is the resulting state predicate after executing a method with specification S.

The rules for determining method call effects are given in figure 8. They apply each conjunct c within p to S and require a valid result for *all* conjuncts. For each c we have to find a method case B => U that has a matching pre-condition and can then yield the corresponding post-condition. We developed this algorithm in earlier work [8] to type function application for union and intersection types [14].

The rules in figures 7 and 8 rely on auxiliary judgments that are presented in figure 9.

5 Component Composition

So far, we can check that a component implementation respects the protocol defined for that component. This is not sufficient, however, because even if a component is implemented correctly, it could be connected to a component that has an incompatible interface. For example, consider the composition in listing 2. If A calls m1(), its port P remains in state a (left side of double arrow), but B's port Q transitions from p to q. Now B is ready to return from m1() (right side of double arrow), but A is not ready to receive the return. On the other hand, B can also call m3(), but A is not ready to receive the call because for this it needs to be in b. Neither of these errors are exhibited by the fixed version of B.

Our component composition checking approach finds these kinds of errors, and provides assurance that a system composition which passes the check will not exhibit any such errors at run time. We first derive finite-state models for ports, components and their connections from the given component specifications. Then we model check the resulting system, verifying that no matter how each component is internally implemented, as long as it obeys its local specification than all components in the system will be used in safe ways. Doing this accurately requires matching call and return edges, as in summary-based inter-procedural analysis[28]. The following sub-sections describe our approach in detail.

5.1 Modeling Ports, Components and Connections

A component $C\langle P, c\rangle$ is built from a set of ports P and a set of connections c. The set $P = (P^1, \ldots, P^n)$ where each P^i is an orthogonal fragment of C and is defined as a structure $P^i =$

$$\frac{\texttt{component class D extends D}'\ \{\ldots\} \quad \mathsf{spec}(m, D') = S \quad D \textit{ does not define } m}{\mathsf{spec}(m, D) = S}$$

$$\frac{\texttt{component class D } \ldots \{\ldots \texttt{port z } \{\ldots \texttt{S C m}(\overline{\texttt{C x}}) \ldots\} \ldots\} \in CT}{\mathsf{spec}(m, D) = S}$$

$$\frac{}{\mathsf{left}(s_1 \texttt{ -> } s_2, D.z) = z.s_1} \qquad \frac{}{\mathsf{right}(s_1 \texttt{ -> } s_2, D.z) = z.s_2}$$

$$\frac{\mathsf{left}(t, D.z) = p}{\mathsf{left}(t \wedge c, D.z) = p \wedge c} \qquad \frac{\mathsf{right}(t, D.z) = p}{\mathsf{right}(t \wedge c, D.z) = p \wedge c}$$

$$\frac{\mathsf{left}(B, D.z) = p_1 \quad \mathsf{left}(U, D.z) = p_2}{\mathsf{left}(B \vee U, D.z) = p_1 \vee p_2} \qquad \frac{\mathsf{right}(B, D.z) = p_1 \quad \mathsf{right}(U, D.z) = p_2}{\mathsf{right}(B \vee U, D.z) = p_1 \vee p_2}$$

$$\frac{[\texttt{component}]\ \texttt{class E extends Object}\ \{\ldots\} \in CT}{\mathsf{override}(m, E, [S]\ \overline{C} \to C_0)}$$

$$\frac{\begin{array}{c}[\texttt{component}]\ \texttt{class E extends E}'\ \{\ldots\} \in CT \\ \mathsf{mtype}(m, E') = \overline{C'} \to C_0' \textit{ implies } \overline{C = C'}, C_0 = C_0'\ [, \mathsf{spec}(m, E') = S]\end{array}}{\mathsf{override}(m, E, [S]\ \overline{C} \to C_0)}$$

Figure 9: Auxiliary functions

$(S^i, \alpha^i, F^i, s_0^i, p)$. With $S = S^1 \times \ldots \times S^n$ we define a port as follows:

- S^i is the finite, non-empty set of states of that port.
- α^i is the set of actions, i.e. the alphabet of P^i.
- $F^i \subseteq \{((s^1, \ldots, s^i, \ldots, s^n), a, (s^1, \ldots, t^i, \ldots, s^n))\} \subseteq S \times \alpha^i \times S$ defines the transition relation for port P^i that can possibly depend on other ports of the component. Notice that a port can only change its own state.
- $s_0^i \in S^i$ is the distinguished start state.
- Finally, $p \in \{\mathsf{private}, \mathsf{public}\}$ specifies if the port is private or public.

The set $c = (c^1, \ldots, c^n)$, where each $c^i = (\mathcal{C}_A^i.\mathcal{P}_A^i, \mathcal{C}_B^i.\mathcal{P}_B^i)$ is a connection between two components. Here $\mathcal{C}_{A,B}^i \in \{C, C_{subp}^1, \ldots, C_{subp}^n\}$, where $C_{subp}^1, \ldots, C_{subp}^n$ are subcomponent instances. Similarly $\mathcal{P}_{A,B}^i \in \{P, P_{subp}^1, \ldots, P_{subp}^2\}$, where $P_{subp}^1, \ldots, P_{subp}^m$ are public ports of subcomponent instances. While composing components we only look at the public ports of subcomponents ignoring their private ports and subcomponents, thereby making our analysis modular. We define C_{subp} to represent only the public part of a subcomponent. Formally, for a subcomponent instance $C_{sub} = \langle P, c\rangle$, $P_{subp} = \{x \mid x \in P \wedge x.p = \mathsf{public}\}$ and $C_{subp} = \langle P_{subp}, \emptyset\rangle$. Note that for a subcomponent instance the set c is empty since only connections of subcomponents declared in the top-level component are considered and these are part of the top-level component.

Deriving the P^i from a given component specification is straightforward. The states are taken from the provided list. The set of actions contains two actions for each method m defined in a port: an action $m.c$ for a *call* to m and an action $m.r$ for a *return* from m. The transition relation follows from the method specifications. There is one tuple in the relation for each boundary transition B. Method entry and exit are handled with separate transitions. The start state is the first state in the state list. Deriving the c^i is also straightforward since it is specified in the component definition.

We derive a model of the component, $C\langle P, c\rangle = (S, \alpha^C, F, s_0, c)$, as follows.

- $S = S^1 \times \ldots \times S^n$ is the *state space* of the component.
- $\alpha^C = \alpha^1 \cup \ldots \cup \alpha^n$ (assuming the α^i are pairwise disjoint) is the *alphabet* of the component.
- $F = F^1 \cup \ldots \cup F^n \subseteq S \times \alpha^C \times S$ is the *transition relation* for C.
- $s_0 = (s_0^1, \ldots, s_0^n)$ is the component's *start state*.
- $c = (c^1, \ldots, c^n)$ are the components connections.

Thus we model a component as a finite state machine whose transitions are labeled with the actions that trigger them. For example, the state space of the Web Server component in listing 1 is $S = \{\mathsf{id(le)}, \mathsf{bu(sy)}\} \times \{\mathsf{ra(w)}, \mathsf{in(itialized)}\} \times \{\mathsf{wa(iting)}, \mathsf{wo(rking)}\}$ with start state $s_0 =$

$(\mathsf{id}, \mathsf{ra}, \mathsf{wa})$. The `Http` port understands the alphabet $\alpha^0 = \{\mathsf{get}.c, \mathsf{get}.r\}$, where $O^0 = \{\mathsf{get}.c\}$. The port's transition relation is as follows.

$$F^0 = \{ \quad ((\mathsf{id}, \mathsf{ra}, \mathsf{wa}), \mathsf{get}.c, (\mathsf{bu}, \mathsf{ra}, \mathsf{wa})), \\ ((\mathsf{bu}, \mathsf{ra}, \mathsf{wa}), \mathsf{get}.r, (\mathsf{id}, \mathsf{ra}, \mathsf{wa})) \quad \}$$

The other ports can be similarly modeled, yielding a complete component model. Recall that ports not mentioned in a specification are interpreted to have arbitrary but unchanged state from call to return. This can be expressed with a suitable set of transitions.

The component's connections set is as follows.

$$c = \{ \quad ((\mathsf{WebServer.Handle}, \mathsf{cypher.CypherHandle}), \\ (\mathsf{cypher.Forward}, \mathsf{access.DocumentTree}) \\ (\mathsf{WebServer.Control}, \mathsf{access.FileAccessControl})) \quad \}$$

where CypherHandle, Foward and FileAccessControl, DocumentTree are ports defined in components Cypher and FileAccess respectively.

5.2 Checking component compatibility

To model check the composition of components we do not look at method implementations at all; instead, we use the behavioral specifications on component ports to represent component behavior. We rely on implementation checking (from section 4) to verify that method implementations respect these specifications.

Our model checking algorithm looks for situations where a method currently being executed believes it can make a call based on the current component's state, but the receiving component is not in the right state to receive the call. An analogous situation can happen when returning to a component that is not in the state expected during returns.

In ArchJava, as in most languages, control flow proceeds according to a call stack in which method calls and returns are matched. For precision, we want to model the call stack to ensure that we do not report false errors corresponding to calling one method and returning from another. The call stack makes the state space infinite, so modeling it directly is impractical. However, we borrow techniques from summary-based interprocedural analysis [28] to avoid infeasible interprocedural paths while ensuring that the analysis terminates.

The checker takes as input the component $C\langle P, c\rangle = (S, \alpha^C, F, s_0, c)$ we are currently checking. We check this component by composing it with its subcomponents and an environment component that exercises the possible action sequences on C' s public ports. The environment component C_{env} is the inverse of the public interface of C, and its ports are connected to the public ports of C. It models the least restrictive environment that observes the protocol assumptions made by C.

Each element (state) of the worklist S_w we consider is defined as $(S_{atCall}, S_{curr}, C_{curr}, m_{curr})$, where S_{atCall} is the state of all components when method m_{curr} is called, S_{curr} is the current state of all components, C_{curr} the current component and m_{curr} the current method. The need for storing the *atCall* state is explained later in the section. By state of all components we mean the combined

1. Given F_p^i is the transition relation for port $\mathcal{P}^i$ in component $\mathcal{C}^i$
2. Given F_p^j is the transition relation for port $\mathcal{P}^j$ in component $\mathcal{C}^j$
3. $S'^i = \{s' | (s, a, s') \in F_p^i\}$
4. $S'^j = \{s' | (s, a, s') \in F_p^j\}$
5. $S'_c = S_c \oplus \{(\mathcal{C}^i \mapsto \mathcal{S}'^i), (\mathcal{C}^j \mapsto \mathcal{S}'^j)\}$ ($\oplus$ overwrites the values mapped to by $\mathcal{C}^{i,j}$ with $\mathcal{S}^{i,j}$ in S_c)

Figure 10: Computing state transitions

state of C_{env} and C and all its subcomponents. Both S_{atCall} and $S_{curr} \in \{C_{env} \mapsto S^e, C \mapsto S^C, C^1_{sub} \mapsto S^1_{sub}, \ldots, C^2_{sub} \mapsto S^n_{sub}\}$, where $S^e \in S$ is the state of C_{env}, $S^C \in S$ the state of C and similarly $S^1_{sub}, \ldots, S^n_{sub}$ are the states of each of its subcomponents.

We start by visiting the initial state $s_0^w = (s, s, C_{env}, m_d)$. s is computed in the obvious way from the initial states of C_{env} and C and its subcomponents. Notice that both the S_{atCall} and S_{curr} states are the same as soon as a method is called. Here, m_d is a dummy method that is used to represent the main method of the environment component. For every connection $(\mathcal{C}^i.\mathcal{P}^i, \mathcal{C}^j.\mathcal{P}^j)$, component $\mathcal{C}^i$ can either call a *requires* method (*provided* by $\mathcal{C}^j$) or return from a *provides* method (*required* by $\mathcal{C}^j$) resulting in a state transition. The transition is of the form (S_w, a, S'_w), where a is either $m.c$ or $m.r$ and S'_w is the new state derived using algorithms 5 and 6.

For every connection $(\mathcal{C}^i.\mathcal{P}^i, \mathcal{C}^j.\mathcal{P}^j)$, the checker considers the *requires* methods for $\mathcal{P}^i$ and *provides* methods for $\mathcal{P}^j$. If it finds that $\mathcal{C}^i$ is in the right state to call a *requires* method but $\mathcal{C}^j$ is not in the appropriate state to receive the call, a *call* error is raised. On the other hand, if $\mathcal{C}^j$ is ready to return but $\mathcal{C}^i$ is not ready to receive the return, a *return* error is raised. Similar checks are performed for the *requires* methods of $\mathcal{P}^j$ and *provides* methods of $\mathcal{P}^i$. We use summary-based interprocedural analysis techniques to match call-return labels, making sure that from method m we only return to those methods m' that could possibly have called m. This is the reason why every state contains the controlling component and method. If we didn't store the method as part of the state, the checker would flag the fixed composition in listing 2 as erroneous. This is because after calling A.m1() followed by B.m2() the combined system state is (A $\mapsto b$, B $\mapsto r$) and with knowledge of this state alone, A could return from either m2() or m3(), but B is not in a state to receive the return from m3(). For a full description of the algorithm see Algorithm 1.

We store S_{atCall} with every state for correctly determining candidate caller methods while checking a return (Algorithm 2). For instance consider the following scenario in which we are analyzing method m, and S_1, S_2, S_3 are three possible states which could result on a call to m (*atCall* states). Each of these could further create new states due to call-return transitions made by the body of m as follows:

- $S_1 \rightsquigarrow^* S_4$
- $S_2 \rightsquigarrow^* S_5$

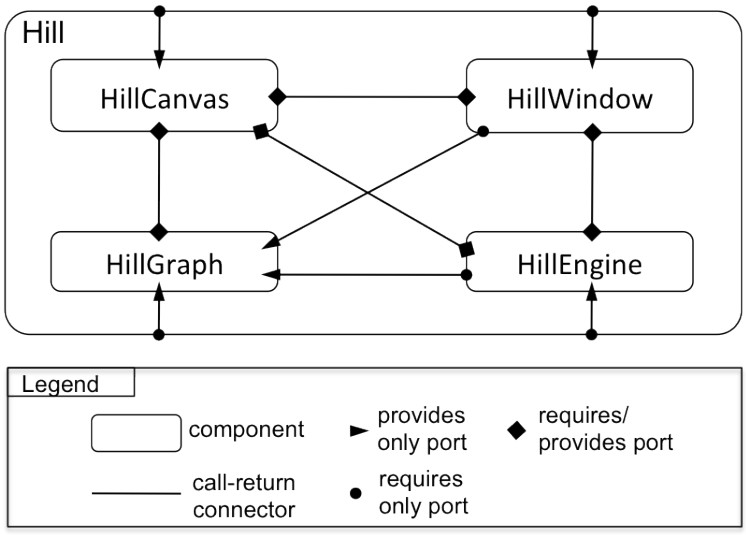

Figure 11: Partial HillClimber Architecture

- $S_3 \rightsquigarrow^* S_6$

By storing the *atCall* state we are able to compute the exact list of methods that could lead to S_4, S_5, S_6 (Algorithm 3).

We visit a new state only if the transition has not been seen previously. Given that our state space is finite and every state combination is visited once we can conclude that our checking algorithm terminates. We point out that it suffices to run the algorithm once for each component *type* to verify that protocol violations cannot occur in the system.

6 Case Study: HillClimber

The HillClimber application is part of AIspace [3], a collection of Java applications used for teaching students artificial intelligence. HillClimber demonstrates stochastic local search algorithms for constraint satisfaction problems (CSP). It was reengineered to ArchJava by Abi-Antoun and Coelho [1] and consists of about 16,000 lines of code including 9 main components and over 75 classes. Figure 11 partially describes the architecture of HillClimber showing the components and connections relevant for this case study. As can be seen from the figure, the subcomponents are fully connected and most of the ports are bi-directional (contain both *requires* and *provides* methods). Although HillClimber is an educational tool, such connections are illustrative of typical application software where components can arbitrarily call back into each other. Hence, through this case study we emphasize the need to reason about callbacks.

The main components in HillClimber interact with each other as follows: the application *window* (HillWindow) uses a *canvas* (HillCanvas) to display *nodes* (HillNode) and *edges* (HillEdge) of a *graph* (HillGraph) in order to demonstrate the algorithms provided by the *engine* (HillEngine)

[3] http://www.aispace.org

Algorithm 1 Check Component Compatibility

1. **input:** Component $C\langle P, c\rangle$
2. Compute initial state $s_0^w$and add to worklist
3. **while** worklist is not empty **do**
4. $S_w = (S_{atCall}, S_{curr}, C_{curr}, m_{curr}) =$ worklist.$remove()$
5. **for each** connection $(C_{curr}.P_1, C_2.P_2) \in C$ **do**
6. **for each** *requires* method $m_r \in P_1$ **do**
7. **for each** spec $T_r \in m_r$ **do**
8. get corresponding *provides* method $m_p \in P_2$
9. **if** $isCallEnabled(S_w, T_r)$ **then**
10. **for each** spec $T_p \in m_p$ **do**
11. **if** $isCallEnabled(S_w, T_p)$ **then**
12. $S'_w = CallTransition(S_w, T_r, T_p, C_2, m_p)$
13. **if** (S'_w) has not previously been seen **then**
14. worklist.$add(S'_w)$
15. **else**
16. *// because it is possible that we missed checking*
17. *// returns from* m_p
18. **for each** $S \in \{S'_w \cup reachableFrom(S'_w)\}$ **do**
19. $checkReturn(C_{curr}, m_p, m_r, S)$
20. **end for each**
21. **end if**
22. **end if**
23. **end for each**
24. **if** $\forall(T_p \bullet T_p \in m_p)\ \neg isCallEnabled(S_w, T_p)$ **then**
25. signal ERROR
26. **end if**
27. **end if**
28. **end for each**
29. **end for each**
30. get corresponding *requires* method $m_r \in P_2$ for m_{curr}
31. $checkReturn(C_{curr}, m_{curr}, m_r, S_w)$
32. **end for each**
33. **end while**

Algorithm 2 checkReturn

1. **input:** C_{curr}, m_p, m_r, S
2. **for each** spec $T_p \in m_p$ **do**
3. **for each** case $B_p \in T_p$ **do**
4. **if** $isReturnEnabled(S, B_p)$ **then**
5. **for each** caller method $m_{caller} \in getCallers(S)$ **do**
6. **for each** $B_r \in m_r$ **do**
7. **if** $isReturnEnabled(S, B_r)$ **then**
8. compute C_{caller} from m_{caller}
9. $S'_w = returnTransition(S, B_r, B_p, C_{caller}, m_{caller})$
10. **if** S'_w has not previously been seen **then**
11. worklist.$add(S'_w)$
12. **end if**
13. **end if**
14. **end for each**
15. **if** $isReturnEnabled(S, B_p) \wedge \forall(B_r \bullet B_r \in m_r)\ \neg isReturnEnabled(S, B_r)$ **then**
16. signal ERROR
17. **end if**
18. **end for each**
19. **end if**
20. **end for each**
21. **end for each**

Algorithm 3 getCallers

1. **input:** $S = (S_{atCall}, S_{curr}, C, m_p)$
2. $callers$ = list of caller methods
3. **for each** state $S' = (S'_{atCall}, S'_{curr}, C', m') \in currentReachableStates$ **do**
4. **for each** corresponding ***requires*** method m_r of m_p that m' can call **do**
5. **for each** spec T_r of m_r and each spec T_p of m **do**
6. **if** $isCallEnabled(S', T_r) \wedge isCallEnabled(S', T_p)$ **then**
7. $S'' = callTransition(S', T_r, T_p, C', m_p)$
8. **if** $S''.S_{curr} = S.S_{atCall}$ **then**
9. add m' to $callers$
10. **end if**
11. **end if**
12. **end for each**
13. **end for each**
14. **end for each**
15. **return** $callers$

Algorithm 4 reachableFrom

1. **input:** $S_w = (S_{atCall}, S_{curr}, C, m)$
2. $states$ = list of all reachable states
3. **for each** state $S' = (S'_{atCall}, S'_{curr}, C', m') \in currentReachableStates$ **do**
4. **if** $S'.atCall = S.S_{curr}$ **then**
5. add S to $states$
6. **end if**
7. **end for each**
8. **return** $states$

Algorithm 5 callTransition

1. **input:** $S_w = (S_{atCall}, S_{curr}, C_{caller}, m), T_r, T_p, C_{callee}, m_p$
2. compute S'_{atCall} from S_{curr} using T_r and T_p as given in Figure 10
3. $S'_{curr} = S'_{atCall}$
4. $S'_w = (S'_{atCall}, S'_{curr}, C_{callee}, m_p)$
5. **return** S'_w

Algorithm 6 returnTransition

1. **input:** $S_w = (S_{atCall}, S_{curr}, C_{return}, m_p), B_r, B_p, C_{caller}, m_{caller}$
2. $S'_{atCall} = S_{atCall}$
3. compute S'_{curr} from S_{curr} using B_r and B_p as given in Figure 10
4. $S'_w = (S'_{atCall}, S'_{curr}, C_{caller}, m_{caller})$
5. **return** S'_w

Algorithm 7 isCallEnabled

1. **input:** S, T
2. **if** state $S.S_{curr}$ matches T for a valid call (left side of big arrow) **then**
3. **return** true
4. **else**
5. **return** false
6. **end if**

Algorithm 8 isReturnEnabled

1. **input:** S, B
2. **if** state $S.S_{curr}$ matches B for a valid return **then**
3. **return** true
4. **else**
5. **return** false
6. **end if**

[1]. Instances of these components and their connections, except for HillNode and HillEdge, are created and initialized in the top-level component Hill.

For this case study, we annotated the ports of HillEngine, HillWindow, HillGraph, HillCanvas and the top-level component Hill that are shown in figure 11. The average number of methods in a connection between subcomponents was about 30, ranging from 10 in the HillWindow-HilGraph to 34 in the HillCanvas-HillGraph connections.

The HillEngine provides two modes of solving a CSP using an algorithm: *autosolve*, where it continues to run till all constraints are satisfied and *batchrun*, where it performs a set of attempts each with a fixed number of steps. The protocol for performing an autosolve is as follows: first the HillEngine must be *initialized*, then the HillWindow must be *autoEnabled*, at which point the HillEngine is ready to start the algorithm (the protocol for a batch run is similar). The HillEngine *provides* a startAutoSolve method on its engineCanvasPort. The HillCanvas component implements a user interface listener and calls startAutoSolve() on receiving the appropriate event. Keeping this in mind the ports of HillEngine were annotated as described in listing 3.

Our analysis helped debug a subtle problem in our protocol specification due to the bidirectional nature of the connections. The component composition checker flagged an error due to the following possible scenario: after initialization HillEngine calls HillWindow.enableAuto(), which invokes HillCanvas.setMode(), which then invokes
HillEngine.startAutoSolve(). This scenario would violate the HillEngine's specification for startAutoSolve() since the engineWindowPort is not in the autoEnabled state until enableAuto() returns.

The problem is in the specification of setMode() in the port of the HillWindow (see listing 4). We fixed this error by changing the specification to ensure that HillWindow cannot call setMode() until enableAuto() has finished and we are in the autoEnabled state:

/:spec windowButtonsDisabled & hillWindow.initialized*
*& hillWindow.autoEnabled ⇒ windowButtonsDisabled */*

Once this specification problem was fixed, the composition check confirmed that the configuration was safe. We also verified the implementation of the component methods against their specifications; the number of methods that did not conform to their specification was less than 10%. Our experience demonstrates that subtle bugs can arise in specifications due to callbacks, and thus it is important for tools to support reasoning about them.

Our unoptimized prototype analysis checks the code for the components in Figure 6 as well as their composition in about 45 seconds [4]. Although HillClimber is only of moderate size, the modular nature of our algorithms means that verification time will scale linearly to larger applications provided that no one component is substantially more complex than those in HillClimber.

[4] user time measured on a 2.4 Ghz CPU with 2 GB RAM, time does not include ArchJava typechecking

$$\frac{p \vdash_D e_1 \dashv p' \vdash_D e_2 \dashv p''}{p \vdash_D e_1\texttt{;}\ e_2 \dashv p''}\ \text{P-SEQ}$$

$$\frac{p \vdash_D e_1 \dashv p' \vdash_D e_2 \dashv p'' \quad (\circ = +, -, \ldots)}{p \vdash_D e_1 \circ e_2 \dashv p''}\ \text{P-ARITH}$$

$$\frac{p \vdash_D e_1 \dashv p' \vdash_D e_2 \dashv p'' \quad (\otimes = \texttt{\&\&}, \texttt{||}, \ldots)}{p \vdash_D e_1 \otimes e_2 \dashv p' \vee p''}\ \text{P-BOOL}$$

$$\frac{p \overline{\vdash_D e \dashv p'} \quad D.m \cdot p' \hookrightarrow p''}{p \vdash_D \texttt{this}.m(\overline{e}) \dashv p''}\ \text{P-INTERNALCALL}$$

Figure 12: Additional protocol checking rules

7 Extensions

This section discusses how the approach developed so far can be generalized to a more realistic language like ArchJava. We begin by extending implementation checking to support typical statement–based methods. Then we discuss how helper methods within a component can be handled. Finally, we consider support for subclassing of components.

7.1 Intraprocedural Analysis

In order to support typical statement sequences in methods we can extend our protocol checking rules to statement sequences, arithmetic, and boolean operations in the obvious way (figure 12). Notice how we take short–circuiting evaluation of boolean predicates into account (rule P-BOOL).

We can devise a dataflow analysis [27] to track state information through control structures like loops and conditional branches. We use a standard forward may–analysis with our checking rules from figures 7 and 12 as the transfer function for individual statements. That analysis will automatically reason about control structures correctly: for instance, state information from the end of a loop will be fed back into the first loop statement.

The lattice we use is essentially a set of tuples that represents the disjunctions of conjuncts in the predicates p defined in section 4. Each conjunct c can mention a port z at most once (otherwise the predicate would be unsatisfiable). Thus a conjunct can be represented with a tuple containing the state of each port. A predicate can then be represented as a set containing the possible tuples.

7.2 Interprocedural Analysis

ArchJava component types can include methods that are not associated with a port. These "helper" methods can be called from port methods, and they can call port methods themselves. There are

two basic options for handling these methods. They can be (a) explicitly annotated just like port methods or (b) analyzed together with the methods that call them.

In order to reduce the burden for the programmer we propose to do the latter and employ a summary–based interprocedural analysis [28]. This means, roughly speaking, that at every call site to an internal method we take the current state information and use it to analyze the called method. We remember the analysis result in case the method is called again in the same context. We can then continue analyzing the caller. There are standard procedures for handling recursive calls and the like that are similar to handling loops in an intraprocedural analysis.

We can be smarter and remember the analysis result for each conjunct in a state predicate separately. At the next call site we can just look up their state transitions and compute results for new conjuncts (rule P-INTERNALCALL in figure 12). Determining the state predicate after a call based on this *summary* information is analogous to figure 8. This has been implemented as part of our implementation checking.

7.3 Component Subclassing

Component subclasses in full ArchJava can define additional ports and provided methods [2]. They can also override existing methods. A viable approach is to check overriding methods against the inherited protocol [13]. The override judgment (figure 9, used by T-METH in figure 5) enforces this by requiring the specification of an overriding method to be *identical* to the inherited one. Earlier work shows how subclasses can *refine* method specifications instead [8].

Additional provided methods in existing ports can be specified with the states already defined for that port. Additional ports can define their own states and specify provided methods with these states. Component subclasses cannot define additional required methods [2]. These restrictions are captured by the following rule that complements T-PORT (figure 5) for component subclasses.

$$\frac{\overline{S\ M}\ \text{typechecks in}\ D \quad \overline{S\ M}\ \text{conforms in}\ D.z \quad z\ \textit{known in}\ D'\ \textit{iff no states defined for}\ z\ \textit{in}\ D}{\texttt{port z \{ [states } \overline{\texttt{s}}\texttt{;] } \overline{\texttt{provides S M}}\texttt{ \}}\ \text{ok in}\ D\ \text{ext}\ D'}\ \text{T-EXT}$$

8 Related Work

The introduction discussed related work on architectural protocols and enforcing communication integrity; here we describe other protocol verification research.

A number of type systems were proposed that augment general-purpose programming languages, in particular C and object-oriented languages, with protocols based on typestates [30, 13, 8]. We can verify method cases and non–determinism as proposed in recent work on expressive object protocols [8]. Alternative approaches for defining protocols include "interface automata" [11]. Their notion of composition is roughly similar to ours but implementation verification is not considered. Lam *et al.* verify set-based typestate protocols of data structures with reported scalability limits due to usage of theorem provers [23]. Verification is modular but their work tracks

states of data that components operate on while we focus on states of the architectural components themselves.

Existing type systems can statically enforce protocols for linear objects (objects with one reference) [13]. While various mechanisms for reducing that linearity restriction in settings with dynamic object creation where proposed, we can reason about arbitrary connections in a fixed component hierarchy. Connections are technically aliases of components that can be accessed independently. We can reason about callbacks across connections even in the presence of aliasing. In previous work, we have reasoned about aliased objects using access permissions [7].

Several lines of research use model checking techniques for modular reasoning about models of software [18, 17, 16]. Assume–guarantee reasoning is a way to apply model checking to components separately [18] but it usually cannot handle callbacks and recursion. We build on Giannakopoulou's formalisms [17], support callbacks and recursive calls, and increase precision by relying on implementation checking. This and other work has also addressed protocol and environment inference which is orthogonal to our approach. Finally, Fisler and Krishnamurthi can reason compositionally about state machines that collaboratively extend a base system [16].

Model checking has also been used for checking temporal properties of implementations [19, 20]. Whole–program analyses scale poorly to large code bases. Blast [20] for example inlines function calls. The developer has to provide code stubs for library functions that serve as a form of abstraction. The Magic tool provides a way to modularly apply model checking to C programs [9] based on user–provided state machines for program and library functions. However, Magic also has problems with scalability because it inlines these state machines at call sites. The assume–guarantee approach taken by Giannakopoulou *et al.* includes modular verification with the Java PathFinder [19] model checker. It uses assumptions and properties derived in the design phase to check implementations [17] in a scalable way. None of these approaches can handle callbacks or recursive calls which are supported by our approach. Our implementation checking proceeds similarly to a typechecker and therefore does not exhibit the state explosion problems typical for software model checkers.

Finally, dataflow analyses have been used to reason about protocols [5, 15]. Like model checkers, these approaches can handle aliasing in component clients. In order to be conservative they typically use a form of global alias analysis [5, 10, 15]. Like many of the type systems discussed above, dataflow analyses typically focus on verifying clients of components with protocols. SLAM for example has been used to verify correct usage of library protocols in device drivers [5]. Our approach does not need a global alias analyses, making it more scalable. Moreover, we can reason about both sides of an interface and handle callbacks across the interface.

9 Conclusions

This paper presents a novel approach for specifying architectural protocols based on typestates and modular techniques for checking component types for protocol conformance. Checking proceeds in two separate steps. A static dataflow analysis checks component method implementations for compliance with the protocols specified for that component. A test based on model checking of labeled transition systems verifies that a component and its immediate subcomponents can

be composed without the possibility of protocol violations. These checks can hierarchically and modularly check the whole system for protocol conformance.

This is the first approach (that we are aware of) that can statically reason about typestates in the presence of true aliasing. It can handle notoriously complicated programming idioms such as callbacks and recursive dependencies both in specifications and their verification. Our approach is based on ArchJava, a programming language that includes architectural primitives like components and ports as first-class constructs. ArchJava's type system gives structural guarantees that make our protocol checks feasible. Our approach is not limited to ArchJava, though. It can work with any language that guarantees communication integrity, i.e. that components communicate with other ports *only* through their explicitly declared ports.

Our approach currently does not support dynamic architectures, i.e. architectures that change over time. ArchJava supports dynamic architectures with port references that can be passed around and port types that can have an arbitrary number of instances. We believe that a port aliasing control regime together with restrictions on the protocol dependencies between port types can enable sound protocol checking of dynamic architectures.

Acknowledgments. We thank Ciera Jaspan and Nels Beckman for their helpful feedback on this paper.

References

[1] Marwan Abi-Antoun and Wesley Coelho. A case study in incremental architecture-based re-engineering of a legacy application. *Software Architecture, Working IEEE/IFIP Conference on*, 0:159–168, 2005.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *European Conference on Object–Oriented Programming*. Springer, June 2002.

[3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[4] Robert J. Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *ACM Symposium on the Foundations of Software Engineering*, pages 70–79, November 1998.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth SPIN Workshop*, pages 101–122, May 2001.

[6] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[7] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, page 320. ACM, 2007.

[8] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *ACM Symposium on the Foundations of Software Engineering*, pages 217–226, September 2005.

[9] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering*, pages 385–395, May 2003.

[10] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[11] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ACM Symposium on the Foundations of Software Engineering*, pages 109–120, September 2001.

[12] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[13] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*. Springer, 2004.

[14] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *ACM Symposium on Principles of Programming Languages*, pages 281–292, 2004.

[15] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis*, pages 133–144, July 2006.

[16] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ACM Symposium on the Foundations of Software Engineering*, pages 152–163, September 2001.

[17] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*, pages 211–220, May 2004.

[18] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[19] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[21] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[22] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999.

[23] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *International Conference on Verification, Model Checking and Abstract Interpretation*, pages 430–447, 2005.

[24] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *ACM Symposium on the Foundations of Software Engineering*, pages 296–305, 2005.

[25] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[26] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *Working IFIP Conference on Software Architecture*, pages 35–50, February 1999.

[27] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2nd edition, 2005.

[28] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis. Theory and Applications*, pages 189–233. Prentice Hall, 1981.

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

[30] Robert E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

A Method Specification Desugaring

Surface specifications are defined as follows and translated according to figure 13.

Method spec	S	::=	T	*base case*
		|	T `,` S	*intersection*
		|	`Unchanged`	*preserve any state*
Method case	T	::=	B `=>` U	*state transition*
		|	s	*preserve specific state*
Method boundary	B	::=	t	*no side condition*
		|	t `&` c	*with side condition*
Postcondition	U	::=	B	*base case*
		|	B `|` U	*union*
Transition	t	::=	s	*hidden execution*
		|	s_1 `->` s_2	*boundry transition*
Conditions	c	::=	$z.s$	*state on port*
		|	$z.s$ `&` c	*condition conjunct*

surface spec $\rightsquigarrow$ *internal spec*

$$\frac{T \rightsquigarrow T' \qquad S \rightsquigarrow S'}{T\texttt{, } S \rightsquigarrow T'\texttt{, } S'} \qquad \frac{s \texttt{ => } s \rightsquigarrow T'}{s \rightsquigarrow T'}$$

$$\frac{s_1\texttt{, } \ldots\texttt{, } s_n \rightsquigarrow S' \qquad (s_1, \ldots, s_n \textit{ is the set of port states})}{\texttt{Unchanged} \rightsquigarrow S'}$$

$$\frac{B \rightsquigarrow s_1 \texttt{ -> } s_2[\wedge c] \qquad s_2 \rhd U \rightsquigarrow U'}{B \texttt{ => } U \rightsquigarrow s_1 \texttt{ -> } s_2[\wedge c] \texttt{ => } U'}$$

Domain Expansion *surface domain* $\rightsquigarrow$ *internal domain*

$$\frac{}{s_1 \texttt{ -> } s_2 \rightsquigarrow s_1 \texttt{ -> } s_2} \qquad \frac{(s_f \textit{ fresh})}{s \rightsquigarrow s \texttt{ -> } s_f}$$

$$\frac{c \rightsquigarrow c'}{s.z \texttt{ \& } c \rightsquigarrow s.z \wedge c'} \qquad \frac{t \rightsquigarrow t' \qquad c \rightsquigarrow c'}{t \texttt{ \& } c \rightsquigarrow t' \wedge c'}$$

Range Expansion $s_e \rhd$ *surface range* $\rightsquigarrow$ *internal range*

$$\frac{}{s_e \rhd s_1 \texttt{ -> } s_2 \rightsquigarrow s_1 \texttt{ -> } s_2} \qquad \frac{}{s_e \rhd s \rightsquigarrow s_e \texttt{ -> } s}$$

$$\frac{s_e \rhd t \rightsquigarrow t' \qquad c \rightsquigarrow c'}{s_e \rhd t \texttt{ \& } c \rightsquigarrow t' \texttt{ \& } c'} \qquad \frac{s_e \rhd B \rightsquigarrow B' \qquad s_e \rhd U \rightsquigarrow U'}{s_e \rhd B \texttt{ | } U \rightsquigarrow B' \vee U'}$$

Figure 13: Expansion Rules

```
1  public component class WebServer {
2  /*:states idle, busy */
3  public port Http {
4      /*:spec idle -> busy & Control.raw & Handle.waiting
5      => busy -> idle & Control.raw & Handle.waiting */
6      provides String get(String get) {
7          String result;
8          Control.prepare(get);
9          try {
10             result = Handle.request(get);
11         }
12         catch(IOException e) { ... }
13         finally {
14             Control.teardown();
15         }
16         return result;
17     }
18 }
19 /*:states raw, initialized */
20 public port Control {
21     /*:spec raw & Handle.waiting
22     => initialized & Handle.waiting */
23     requires void prepare(Object context);
24
25     /*:spec initialized & Handle.waiting
26     => raw & Handle.waiting */
27     requires void teardown();
28 }
29 /*:states waiting, working */
30 public port Handle {
31     /*:spec waiting->working & Control.initialized
32     => waiting & Control.initialized */
33     requires String request(String doc) throws IOException;
34 }
35 private final Cypher cypher = new Cypher();
36 private final FileAccess access = new FileAccess();
37 connect Handle, cypher.CypherHandle;
38 connect cypher.Forward, access.DocumentTree;
39 connect Control, access.FileAccessControl;
40 }
```

Listing 1: Simple web server example

```
41 public component class A {
42     /*:states a,b*/
43     public port P {
44         /*:spec a->a=>b->a */
45         requires void m1();
46         /*:spec a->b=>b->b */
47         provides void m2();
48         /*:spec b->b=>b->b */
49         provides void m3();
50   }
51   final private B b = new B();
52   connect P, b.Q;
53 }
54
55 public component class B /*buggy */ {
56     /*:states p,q*/
57     public port Q {
58         /*:spec p->q=>q->p */
59         provides void m1();
60         /*:spec q->q=>q->q */
61         requires void m2();
62         /*:spec q->q=>q->q */
63         requires void m3();
64     }
65 }
66
67 public component class B /*fixed */ {
68     /*:states p,q,r,s,t*/
69     public port Q {
70         /*:spec p->q=>t->p */
71         provides void m1();
72         /*:spec q->r=>r->r */
73         requires void m2();
74         /*:spec r->s=>s->t */
75         requires void m3();
76     }
77 }
```

Listing 2: Component Composition example

```
78  /*:states raw, initialized */
79  public port hillEngine {
80    /*:spec raw => initialized */
81    provides void init();
82  }
83
84  /*:states ready, autoEnabled, batchEnabled */
85  public port engineWindowPort {
86    /*:spec ready & hillEngine.initialized
87    => autoEnabled*/
88    requires void enableAuto();
89  }
90
91  /*:states stopped, runningBatchMode,
92    runningAutoSolveMode */
93  public port engineCanvasPort {
94  /*:spec stopped & hillEngine.initialized
95    & engineWindowPort.autoEnabled
96  => runningAutoSolveMode */
97    provides void startAutoSolve();
98  }
```

Listing 3: Partial HillEngine specification (not all methods and ports shown)

```
99  /*:states windowButtonsDisabled, windowButtonsEnabled,
100   canvasButtonsDisabled, canvasButtonsEnabled */
101 public port windowCanvasPort {
102   /*:spec windowButtonsDisabled & hillWindow.initialized
103   => windowButtonsDisabled */
104   requires void setMode(int newMode);
105   ... other methods
106 }
```

Listing 4: setMode specification in HillWindow