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

Kevin Bierhoff[*] **Jonathan Aldrich**[*]
Sangjin Han[†]

April 2006
CMU-ISRI-07-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*] Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA.
{kevin.bierhoff,jonathan.aldrich} @ cs.cmu.edu
[†] Formerly: Carnegie Mellon University, Pittsburgh, PA, USA.

This technical report was drafted under the number CMU-CS-06-119 and has been available on the first author's website since April 2006. It was officially published without content changes in December 2007 as CMU-ISRI-07-121.

This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, 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 uses model checking techniques 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.

1 Introduction

Our models and our understanding of software architecture [30] have come a long way in the last decade. Numerous architecture description languages (ADLs) have been proposed to capture the runtime structure of a software system. What used to be drawn with informal "box–and–line diagrams" can now be described in more formal terms of components and connectors.

Component diagrams as they are part of UML 2.0 [28] capture runtime components, their ports (interfaces), and the connections between ports of different components. Intuitively, a component can (directly) communicate with another component if *and only if* they are explicitly connected through ports. This structural property of *communication integrity* [25] is very helpful in reasoning about systems.

But what if we want to constrain the communication patterns across ports? Some ADLs include features to describe architectural protocols. Darwin [26] for example can specify possible event sequences of components and Wright [3] uses CSP [22] to define protocols. The benefit of such an approach is that usage rules for components are made explicit. When composing components we can check if their protocols are compatible.

Model checking techniques have been applied to modular reasoning about temporal properties of implementations [9, 18]. Unfortunately, these approaches do not handle architectures with multiple component instances and cannot reason about callbacks. The ADL Rapide included a runtime technique for detecting protocol violations but could not prevent them statically [25].

This paper proposes a modular approach to enforcing architectural protocols at compile time in implementations with multiple component instances that can be arbitrarily nested. It supports idioms such as callbacks and recursive calls that are difficult to reason about. Thus even though it is restricted to static architectures (with a fixed number of components) the approach supports an important class of systems. Our behavioral specifications are based on typestates [31], a programming language concept to track the possibly changing state of objects in addition to their fixed type. Typestates let us specify port behavior directly with abstract state machines.

To our knowledge this is the first approach that uses communication integrity to reason about implementation behavior. Communication integrity lets us devise a sound modular dataflow analysis that can verify protocol conformance of method implementations. We rely on the ArchJava programming language to capture and enforce architectural structure [1, 2]. We build on model checking techniques developed for assume–guarantee reasoning [17] to verify protocol compliance in component compositions. We leverage the hierarchical structure of software architectures to reason about composition for each component type separately. In summary, the contributions of this paper include the following.

- We propose the first system (we know of) that can statically enforce architectural protocols in software systems with multiple instances per component type. Even though the number of instances is fixed, components can be nested arbitrarily deep. Unlike existing work, we can reason about notoriously tricky programming idioms such as a callbacks and recursive calls. The approach is modular, hierarchical and compositional because it considers only one component type with its immediate subcomponents at a time.

- In contrast to most approaches to defining architectural protocols we use typestates to specify protocols. This gives the programmer the ability to think of the protocol in terms of states rather than event sequences. We support non–determinism and enrich protocols to include dependencies between ports.

- We introduce the novel concept of "boundary transitions" from one state to another at the entrance and exit of methods. This lets us treat typestates as purely static tokens with no runtime representation. In contrast to existing work in this area [12, 16], we can reason not only about methods that call state–changing methods, but also about methods that change states themselves.

- We separate the verification into two independent steps, one that looks at method implementations and one that reasons about component composition. We employ different techniques, dataflow analysis and model checking, in the two steps, and show how they work together. The dataflow analysis extends existing techniques [12] to handle our more complex specifications including non–determinism. Component composition is *flexible* in that we just check that none of the protocols of connected components can ever be violated; the protocols need not be identical.

- We define the first programming language that captures and enforces protocols between architectural components. Our method implementation check is similar to typechecking and therefore does not suffer from state explosion problems typical for software model checking techniques [9].

- This is the first system (that we know of) that can reason about typestates in the presence of limited aliasing of components.

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

Figure 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 [30]. 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 figure 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 [7] 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, 7]. We leverage the concept of typestate [31] to specify a state machine that defines a protocol for each port (figure 1). By contrast, research on architectural protocols [3, 26] usually defined protocols in a form of process calculus such as CSP [22]. 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 [7]. Moreover, typestates are an abstraction that lets the developer think about pre- and post-conditions for each operation separately. With a process

```
public component class WebServer {

/*:states idle, busy */
public port Http {
    /*:spec idle -> busy
       & Control.raw & Handle.waiting
    => busy -> idle
       & Control.raw & Handle.waiting */
    provides String get(String get) {
        String result;
        Control.prepare(get);
        try {
            result = Handle.request(get);
        }
        catch(IOException e) { ... }
        finally {
            Control.teardown();
        }
        return result;
    }
}

/*:states raw, initialized */
public port Control {
    /*:spec raw & Handle.waiting
    => initialized & Handle.waiting */
    requires void prepare(Object context);

    /*:spec initialized & Handle.waiting
    => raw & Handle.waiting */
    requires void teardown();
}

/*:states waiting, working */
public port Handle {
    /*:spec waiting->working & Control.initialized
    => waiting & Control.initialized */
    requires String request(String doc)
        throws IOException;
}
}
```

Figure 1: Simple web server example

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 2: 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.

Figure 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 [7]. The following paragraphs describe our specification approach in detail. The exact language for method specifications is shown in figure 2.[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 [7]. 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 2 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 [1]. 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 figure 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 [1], a core language for ArchJava.

3.1 Syntax

The syntax is summarized in figure 3. 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 [23]). 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.

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

Figure 3: 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 2) 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 [1]. 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 4. 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 5 and 7.

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 5). 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 4: 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 6 contains rules for typechecking expressions and declarations. We discuss component subclassing separately in section 6.3. 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 [23]. 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 a new expression of their own. This is equivalent to field initializers in full ArchJava and

[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 5: Auxiliary judgments for evaluation

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

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

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

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

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

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

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

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

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

Figure 6: Expression and declaration typechecking

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 10. The helper judgments fields and mtype are similar to ArchFJ [1], as is checking of connections with $\overline{X}$ ok in D (figure 7).

Figure 7 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 [1]. The core language defined here preserves communication integrity as proved for ArchJava [1].

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

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

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

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

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

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

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

Figure 7: 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 8: 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 [1].

Figure 6 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 8. Protocol conformance checking proceeds for each specification case separately (rule P-CASES). Notice that a method implementation has to conform to *all* cases

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

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

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

Figure 9: Deterministic method call algorithm

defined within a method specification S. Conformance checking for a method case B => 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

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

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

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

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

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

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

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

Figure 10: Auxiliary functions

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 2). 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 9. 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 [7] to type function application for union and intersection types [14].

The rules in figures 8 and 9 rely on auxiliary judgments that are presented in figure 10.

5 Component Composition

So far, we can check that a component implementation respects the protocol defined for that component. We now turn to reasoning about component assemblies. Our goal is to verify that the protocols of connected components are compatible. Our approach proceeds in three steps. First we derive finite-state models for ports and components from the given port specifications. Then we define parallel composition of state machines. Finally, we devise a modular composition check. This section builds on work by Giannakopoulou et al. [17, 10] on assume–guarantee reasoning.

5.1 Modeling Ports and Components

A component $C\langle P^1, \ldots, P^n\rangle$ is built from ports $P^1, \ldots, P^n$. Each P^i is an orthogonal fragment of C and is defined as a structure $P^i = (S^i, \alpha P^i, O^i, F^i, s_0^i)$. 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. αP^i is the set of actions, i.e. the alphabet of P^i. $O^i \subseteq \alpha P^i$ is the subset of actions that are under outside (external) control. The port P^i has no control when these actions occur. $F^i \subseteq \{((s^1, \ldots, s^i, \ldots, s^n), a, (s^1, \ldots, t^i, \ldots, s^n))\} \subseteq S \times \alpha P^i \times S$ defines the transition relation for port P^i that can depend on other ports of the component. Notice that a port can only change its own state. Finally, $s_0^i \in S^i$ is the distinguished start state.

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 subset of actions under external control consists of calls to provided and returns from required methods. 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.

We derive a model of the component, $C\langle P^1, \ldots, P^n\rangle = (S, \alpha C, O, F, s_0)$, as follows. The distinguished *internal transition* $\tau \notin \alpha C$ will be useful later.

- $S = S^1 \times \ldots \times S^n$ is the *state space* of the component.
- $\alpha C = \alpha P^1 \cup \ldots \cup \alpha P^n$ (assuming the αP^i are pairwise disjoint) is the *alphabet* of the component.
- $O = O^1 \cup \ldots \cup O^n \subseteq \alpha C$ is the subset of actions under *outside control*.
- $F = F^1 \cup \ldots \cup F^n \subseteq S \times \alpha C \cup \{\tau\} \times S$ is the *transition relation* for C.
- $s_0 = (s_0^1, \ldots, s_0^n)$ is the component's *start state*.

Thus we model a component as a state machine whose transitions are labeled with the actions that trigger them.[3]

[3]We do not consider calls from one provided method to another one of the same component in this paper. They can be modeled with internal transitions, preferably only where they actually occur in the implementation.

Our composition check needs the ability to mask parts of a component model. Following existing work we introduce an *interface operator* $C \uparrow A$ that limits C to the actions in A and replaces other actions with internal transitions [17]. Formally, if $C = (S, \alpha C, O, F, s_0)$ then $C \uparrow A = (S, \alpha C \cap A, O \cap A, F \uparrow A, s_0)$, where $F \uparrow A = \{(s, a, s') \mid a \in A, (s, a, s') \in F\} \cup \{(s, \tau, s') \mid a \in \alpha C \setminus A, (s, a, s') \in F\}$. We use the interface operator to limit a component model to actions from a subset of ports. We will in particular distinguish the *public* ports of a component, i.e. the ports that are not connected to subcomponents.

5.2 Composition

In this section we define parallel composition of labeled transition systems (LTS) following existing work [17, 10]. An LTS is a structure $T = (S, \alpha T, F, s_0)$ where S (states), αT (alphabet), and s_0 (start state) are defined as above. We distinguish an *error state* $\pi \notin S$ in addition to the internal transition τ. Then the transition relation is defined as $F \subseteq S \times \alpha T \cup \{\tau\} \times S \cup \{\pi\}$.

We write $T \xrightarrow{a} T'$ to mean that an LTS $T = (S, \alpha T, F, s_0)$ can step to T' in one step $(s_0, a, s_0') \in F$. T' is either identical to T except for the start state or $T' = \Pi = (\{\pi\}, \emptyset, \emptyset, \pi)$ in the error case where $s_0' = \pi$.

When composing two LTS T^1 and T^2 defined as $T^i = (S^i, \alpha T^i, F^i, s_0^i)$ we abstract from *shared actions*. These correspond to methods on connected ports between the two components. We compute an LTS $T = T^1 \mid\mid T^2 = (S, \alpha T, F, s_0)$ that represents the composition of T^1 and T^2 as follows.

- $S = S^1 \times S^2$.
- $\alpha T = (\alpha T^1 \cup \alpha T^2) \setminus (\alpha T^1 \cap \alpha T^2)$.
- The transition relation F is the symmetric closure of the following rules.

$$\frac{T^1 \xrightarrow{a} L^1 \quad (a \notin \alpha T^2)}{T^1 \mid\mid T^2 \xrightarrow{a} L^1 \mid\mid T^2} \qquad \frac{T^1 \xrightarrow{a} L^1 \quad T^2 \xrightarrow{a} L^2 \quad (a \neq \tau)}{T^1 \mid\mid T^2 \xrightarrow{\tau} L^1 \mid\mid L^2}$$

  We treat the error state specially and let $\Pi \mid\mid T = T \mid\mid \Pi = \Pi$.

- $s_0 = (s_0^1, s_0^2)$.

The transition relation F lets both components step for shared actions (second rule) and only one component step for other actions (including τ transitions; first rule). This definition is similar to CSP [22]. Testing an LTS for protocol violations now amounts to finding a path from the start state to π. This path serves as a counterexample, i.e. a sequence of actions that leads to the protocol violation.

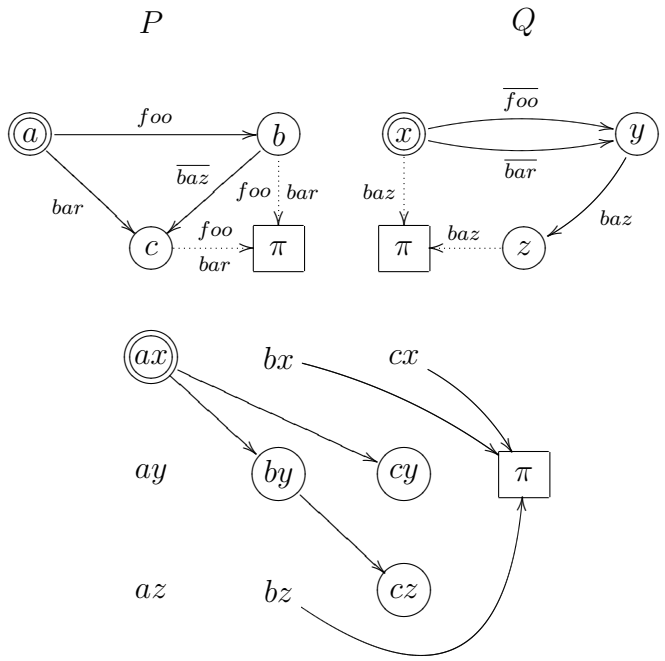

Figure 11: Component composition example

5.3 Modular Composition Checking

Ultimately we are interested in verifying that shared actions will succeed. This means that if one component wants to issue a shared action the other component can perform that action. In other words, when a component calls a required method on one of its ports, the component connected to that port must be ready to run the called method according to its own protocol.

We adapt assume–guarantee reasoning [17] to perform this test and trap unexpected calls in the error state. The usual approach is to check that if one component can perform an action then the other one can do so as well. By contrast, our approach only considers actions that a component can actually initiate in order to avoid false positives from calls that the component cannot make. We can also reason about callbacks, which are not covered by standard assume–guarantee reasoning.

To illustrate the differences, consider the example components in figure 11 (ignoring the dashed lines for now). Start states are marked with a double circle. Following process algebra conventions, we distinguish actions that can be initiated by a component with an overbar. A standard composition analysis of these two components would flag a possible protocol violation for the event sequence $(\mathit{bar}, \mathit{baz})$. However, because P alone decides when *baz* occurs, this sequence is impossible. Implementation checking (section 4) guarantees that components initiate actions only where they are enabled by the protocol. Therefore it suffices to trap unexpected *foo* and *bar* actions in the error state of P (dashed lines). This lets us verify that P and Q can be composed without protocol violations (bottom of figure 11).

Following this idea we construct for a given component model $C = (S, \alpha C, O, F, s_0)$ an "error LTS" $C_e = (S, \alpha C, F_e, s_0)$. F_e is derived from F by adding transitions to the error state for

unexpected calls from outside as follows.

$$F_e = F \cup \{(s, a, \pi) \mid s \in S, a \in O, \neg\exists s' \in S : (s, a, s') \in F\}$$

Based on these techniques we developed a modular composition check that can be applied to architectures with arbitrary nesting. Suppose we want to check a component C with subcomponents $C^1, \ldots, C^k$. The intuition of our approach is that we can perform this check by composing C with its subcomponents and an environment model that exercises the possible action sequences on C's public ports.

Let A be the set of actions on the public ports of C and $A^1, \ldots, A^k$ be the actions on public ports of the C^k. Then we can check C by building the following composition.

$$P = C_e \mid\mid (C^1 \uparrow A^1)_e \mid\mid \cdots \mid\mid (C^k \uparrow A^k)_e \mid\mid (\overline{C} \uparrow A)_e$$

With $\overline{C}$ we mean the *inverse* component of a given component C. If $C = (S, \alpha C, O, F, s_0)$ then $\overline{C} = (S, \alpha C, \alpha C \setminus O, F, s_0)$. It models the least restrictive environment that observes the protocol assumptions made by the component. In the composition we restrict this environment to actions on public ports. Likewise we restrict the subcomponents to their public ports. Notice that order matters: We restrict interfaces and build inverses before we construct error LTS.

The LTS P consists only of τ transitions. If the error state π is reachable from the initial state then there exists a protocol violation in the composition of C with its immediate subcomponents and its environment. The path from the initial to the error state witnesses the violation, serving as a counterexample. We point out that it suffices to perform this test once for each component *type* to verify that protocol violations cannot occur in the system.

6 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.

6.1 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 8 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.

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

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

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

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

Figure 12: Additional protocol checking rules

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.

6.2 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 [29]. 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 9.

6.3 Component Subclassing

Component subclasses in full ArchJava can define additional ports and provided methods [1]. They can also override existing methods. A viable approach is to check overriding methods against the inherited protocol [13]. The override judgment (figure 10, used by T-METH in figure 6) 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 [7].

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 [1]. These restrictions are captured by the following rule that complements T-PORT (figure 6) for component subclasses.

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

7 Related Work

Architectural protocols were proposed in the Rapide [25], Wright [3], and Darwin [26] ADLs. Rapide includes pattern–based behavior specifications for interfaces. In Wright, protocols are defined with a process notation based on CSP [22]. Darwin proposes its own specification language FSP that is very similar to a process calculus. The latter two approaches use variants of model checking to reason about protocols and composition. We define protocols as explicit state machines using typestates [31] and enforce them in a programming language. Many other architectural description languages have been proposed over the years, including structured classifiers for the UML 2.0 [28]. Most ADLs focus on structure and do not include behavior specifications.

ArchJava is a programming language that includes architectural primitives as first-class language constructs [1, 2]. ArchJava components can define ports, subcomponents, and connections between them. Modular typechecking guarantees communication integrity [25]: a component communicates with other components only through explicitly declared ports. We leverage this guarantee for our static protocol reasoning. ArchJava includes features for dynamic architectures that we plan to support with protocol enforcement capabilities in the future.

There has been a variety of research on augmenting general-purpose programming languages, in particular C and object-oriented languages, with protocols based on typestates [31, 12, 16, 13, 7]. We adopt ideas for method cases and non–determinism from our earlier work on more expressive protocols [7]. The usage of a separate set of states for each port is a special case of state dimensions as proposed in that work.

Existing typestate systems can statically enforce protocols only for linear objects [12, 16, 13], i.e. objects with only one (active) reference. We can reason about components with an arbitrary number of ports. Ports are technically aliases of the component that can be accessed independently. We can reason about callbacks even in the presence of aliasing.

There are alternative approaches for defining protocols including labeled transition systems [8] and "interface automata" [11]. They are ultimately translated into variants of state machines. None of these approaches includes static checking of implementations. The notion of composition defined for interface automata is roughly similar to ours.

Several lines of research use model checking techniques for modular reasoning about models of software [19, 17, 15]. Assume–guarantee reasoning is one way to apply model checking to

components separately [19] but it usually cannot handle callbacks and recursion. Giannakopoulou et al. proposed a framework that learns environment assumptions of a component [17, 10]. We build on their formalisms and support callbacks and recursive calls. Learning assumptions is not the goal of this work but could be added to our approach to reduce the protocol specification burden. Fisler and Krishnamurthi can reason compositionally about state machines that collaboratively extend a base system [15].

Model checking has also been used for checking temporal properties of implementations [20, 21]. Usually these are designed as whole–program analyses that scale poorly to large code bases. Blast [21] for example inlines function calls. The developer has to provide code stubs for library functions that serve as a form of abstraction. SLAM can verify correct usage of library protocols in, e.g., device drivers [5].

The Magic tool provides a way to modularly apply model checking to C programs [9] based on user–provided state machines for library functions. However, Magic also has problems with scalability because it inlines these state machines for function calls. The assume–guarantee approach taken by Giannakopoulou *et al.* includes modular verification of Java code with Java PathFinder [20], a model checker for Java. It uses assumptions and properties derived in the design phase to check implementations [18] 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.

8 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.

9 Acknowledgments

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

References

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

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *International Conference on Software Engineering*, pages 187–197, May 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] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *ACM Symposium on the Foundations of Software Engineering*, pages 217–226, September 2005.

[8] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *ACM Symposium on the Foundations of Software Engineering*, pages 50–59, 2000.

[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] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, April 2003.

[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] 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.

[16] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[17] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *International Conference on Automated Software Engineering*, September 2002.

[18] 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.

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

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

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

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

[23] 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.

[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] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.

[29] 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.

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

[31] 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