Correctness and Composition of Software Architectures*

Mark Moriconi and Xiaolei Qian

Computer Science Laboratory
SRI International
Menlo Park, California 94025

ABSTRACT

The design of a large system typically involves the development of a hierarchy of different but related architectures. A criterion for the relative correctness of an architecture is presented, and conditions for architecture composition are defined which ensure that the correctness of a composite architecture follows from the correctness of its parts. Both the criterion and the composition requirements reflect special considerations from the domain of software architecture.

The main points are illustrated by means of familiar architectures for a compiler. A proof of the relative correctness of two different compiler architectures shows how to decompose a proof into generic properties, which are proved once for every pair of architectural styles, and instance-level properties, which must be proved for every architecture.

1 Introduction

The development of an architecture for a large system is a complicated task that can be made simpler by means of a stepwise development methodology. Ideally, an architect would use a hierarchical approach in which the composition of lower-level architectures is guaranteed to implement a higher-level architecture. The foundations for such an approach must include a method for proving that one architecture implements another architecture and a means of composing architectures so that the composite architecture is correct if all of its components are correct. We examine both problems in this paper. We work at the logic level, independent of a particular architecture definition language. Thus, our results can be applied to a large class of such languages.

An architecture hierarchy is a sequence of two or more individual architectures that may differ with respect to the number and kind of objects and connections. For example, an abstract architecture containing functional components related by dataflow connections may be implemented in a concrete architecture in terms of procedures, control connections, and shared variables. An abstract architecture usually is smaller and easier to understand; a concrete architecture usually reflects more implementation concerns. A given architecture can be homogeneous (consisting of one style) or heterogeneous (consisting of multiple styles). Garlan and Shaw [7] provide a taxonomy of some common styles, including dataflow, pipe-and-filter, client-server, and event-based systems.

Before we can consider the relative correctness of two architectures, we first must decide on the meaning of the architectures. Suppose that, to facilitate system upgrades and maintenance on a particular system, we design a pipeline architecture that restricts the system topology to a linear sequence of filters. If a concrete architecture implements the pipeline, but additionally introduces feedback loops, the *raison d'être* behind the original pipeline architecture is no longer valid. In effect, there is no reason to specify a pipeline in the first place if all possible feedback loops are allowed in its implementation.

Therefore, we make a *completeness assumption* about a given architecture. Informally, the assumption is that, if an architectural fact is not explicit in the architecture, or deducible from the architecture, then the fact is not intended to be true of the architecture. In the pipeline example, it is not possible to infer the existence of a feedback loop from the linearity constraint, so we assume that no feedback loop is allowed in an implementation of the architecture. In general, an architecture (whether static or dynamic) can contain an unbounded number of facts.

*This research was supported by the Advanced Research Projects Agency under Rome Laboratory contract F30602–93–C–0245.

The completeness assumption requires a correctness criterion that differs from the standard one (that is based on theory extension). In our application of the correctness criterion, we make a clear distinction between type-level properties that must be proved only once for every pair of architectural styles and instance-level properties that must be proved for every pair of architectures. This decomposition greatly simplifies correctness proofs and the statement of the mapping between two architectures. Composition is possible under the completeness assumption provided that certain syntactic constraints are satisfied.

This paper is organized as follows. The next two sections introduce basic architectural concepts and illustrate the correctness problem for architectures. Section 4 defines the correctness criterion in terms of logical theories, independent of any particular architectural definition language. Sections 5–7 explain how to use the criterion. Of particular interest is the construction and validation of the mapping between architectures. Section 8 defines necessary and sufficient conditions for architecture composition and defines two specific composition operators. Section 9 discusses related work, and the conclusion summarizes our results and discusses their possible implications for future research in software architecture.

2 Basic Architectural Concepts and Notation

A software architecture is represented using the following concepts.

1. **Component**: An object with independent existence, e.g., a module, process, procedure, or variable.
2. **Interface**: A typed object that is a logical point of interaction between a component and its environment.
3. **Connector**: A typed object relating interface points, components, or both.
4. **Configuration**: A collection of constraints that wire objects into a specific architecture.
5. **Mapping**: An relation between the vocabularies and the formulas of an abstract and a concrete architecture. The formula mapping is required because the two architectures can be written in different styles.
6. **Architectural style**: A style consists of a vocabulary of design elements, a set of well-formedness constraints that must be satisfied by any architecture written in the style, and a semantic interpretation of the connectors.

Components, interfaces, and connectors are treated as *first-class objects* — i.e., they have a name and they are refineable. Abstract architectural objects can be decomposed, aggregated, or eliminated in a concrete architecture. The semantics of components is not considered part of an architecture, but the semantics of connectors is.

We will use a simple notation for describing an architecture. Suppose that we want to describe the interaction between the parser and the semantic analyzer in a standard compiler. A dataflow architecture for this interaction is contained in Figure 1.[1]

```
parse_analyze: MODULE
   IMPORT ...
   EXPORT ...
   COMPONENTS
      parser    :  Function
      analyzer  :  Function
   INTERFACES
      oast      :  OPORT [ast] OF parser
      iast      :  IPORT [ast] OF analyzer
   CONNECTORS
      ast_channel  :  Dataflow_Channel[ast]
   CONFIGURATION
      Connects(ast_channel, oast, iast)
END parse_analyze
```

Figure 1: Example Dataflow Architecture

The parser and analyzer are modeled as functional components. The parser (which accepts a sequence of tokens) has an output port `oast` that supplies an abstract syntax tree. The analyzer accepts a values of type `ast` (producing values of the same type). The dataflow connection is wired to the right ports by the assertion

`Connects(ast_channel, oast, iast)`

where `Connects(c,o,i)` means that connection `c` links output port `o` to input port `i`. All of the objects that make up the architecture are wrapped by a module, which can selectively import and export objects. In this example, we import some useful compiler types and the predefined functional and dataflow styles.

The dataflow architecture separates and names all components, ports, and connections. Observe that the signature of a component is not hard-wired to the component. A signature consists of individual ports that can be referenced and refined independently of the associated component. Interface separation will be useful later for architecture composition.

[1]The precise syntax is not important for the purposes of this paper. Later, we formalize this architecture in logic, and that is the representation that is intended to express the intentions of the desigrer.

3 Illustration of the Problem

Suppose that we want to design the architecture for a compiler. A standard dataflow model of a compiler is depicted at the top of Figure 2. The diagram is used only as an informal pedagogical aid; it is not intended to be a formal specification. Boxes denote functional components and arrows denote directional dataflow between ports. The labels on arrows denote types or value domains. An object cannot be transmitted between ports unless its type is compatible with the types of the ports. The diagram is assumed to be complete in that there can be no other functional components, ports, or data flows.[2]

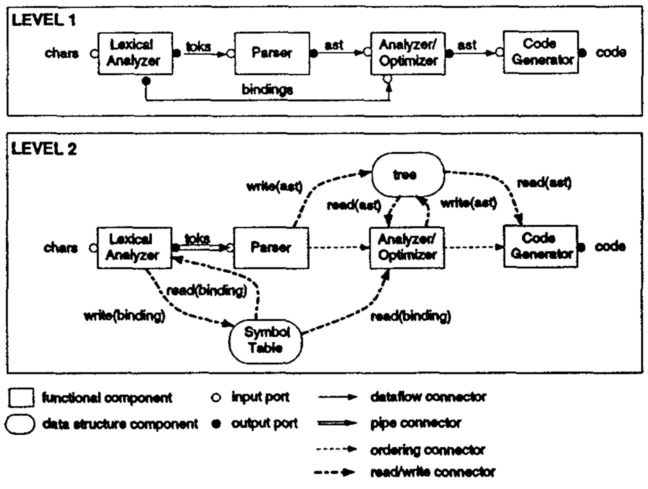

Figure 2: Two architectures for a compiler

Figure 2 also contains a concrete, hybrid architecture for the compiler that implements the dataflow style in terms of pipe-filter, batch-sequential, and shared-memory styles. Abstract signatures are changed in the concrete architecture, dataflow connections are implemented in several ways, through a pipe and shared data objects, and precedence relations are used to prevent direct flow of data from the parser to the code generator.

To illustrate the correctness problem, we focus on the implementation of the dataflow channel between the parser and analyzer in terms of the reading and writing of a shared abstract syntax tree. The implementation architecture is described textually in Figure 3. The shared abstract syntax tree is represented as a variable.[3] The read and write relations are not named; they they are primitives that cannot be refined.

[2]A dataflow connection is treated here as an intransitive relation.

[3]The shared abstract syntax tree might be represented as an encapsulated data type in a real compiler. If we had chosen that representation, the architecture would involve calls to access functions that read and write the internal variable used to represent the tree.

```
concrete_parse_analyze: MODULE
   IMPORT ...
   EXPORT ...
   COMPONENTS
      parser     :  Function
      analyzer   :  Function
      tree       :  Variable[ast]
   CONFIGURATION
      Writes(parser, tree)
      Reads(analyzer, tree)
END concrete_parse_analyze
```

Figure 3: Concrete Shared-Memory Architecture

The intended associations between the two architectures are

```
oast -->
iast -->
ast_channel --> tree
```

The first two associations indicate that the abstract ports do not appear in the concrete architecture, resulting in a new concrete signature for the parser and the analyzer. This change in signature reflects the difference between port-to-port communication and shared-memory communication by direct reading and writing of a shared tree. As an analagous example, consider two procedures that communicate through direct calls. If we reimplement this architecture so that the procedures communicate only indirectly through a shared variable, the signature of both procedures would change. The third association says that dataflow connection is implemented by the abstract syntax tree.[4]

We are interested in three specific questions:

- Does the concrete shared-memory architecture implement the abstract dataflow architecture under the completeness assumption and with respect to a given mapping between architectures?
- Is the mapping between the two architectures meaningful? A relative correctness proof is only as meaningful as the mapping between architectures.
- Assuming that the shared-memory implementation of dataflow is correct, under what conditions can it be composed with correct implementations of other parts of the compiler to form a correct and complete compiler architecture?

The running examples in the paper provide a detailed answer to each of these questions.

[4]The tree is a component. A component can be used to implement other components, or it can be used in conjunction with connectors to implement a connection.

4 Formal Criterion of Correctness

Because of the completeness assumption, we must prove not only that a concrete architecture does not lose properties of the abstract architecture, but also that no new properties about the abstract architecture can be inferred from the concrete architecture. There are standard mathematical concepts that can be used for this purpose.

An *interpretation mapping* is an association between the constants, functions, and predicates of an abstract and a concrete theory. An interpretation mapping is called a *theory interpretation* if the mapped axioms of the abstract theory become theorems of the concrete theory. Note that theory interpretation is just Hoare's approach to reasoning about the correctness of implementations [9]. We additionally require that, if a sentence is not in the abstract theory, its image is not in the concrete theory.

Let Θ and Θ' be theories associated with an abstract and a concrete architecture, respectively. Let I be an interpretation mapping from Θ to Θ'. Then, we must have, for every sentence F,

$$\text{if } F \in \Theta \text{ then } I(F) \in \Theta'$$

for I to be a theory interpretation.

Since we require that an architecture be complete with respect to a given level of detail, we additionally must know that the concrete architecture adds no new facts about the abstract architecture. Therefore, we require that

$$\text{if } F \notin \Theta \text{ then } I(F) \notin \Theta'$$

This says that, if a sentence is not in the abstract theory, its image cannot be in the concrete theory. A theory interpretation I having this property is said to be a *faithful interpretation*. Observe that Θ' is a conservative extension of Θ provided the identity map faithfully interprets Θ in Θ'.

Note that a concrete architecture can contain facts not related to the abstract architecture. Therefore, a concrete architecture can introduce new styles and new objects. For example, a concrete architecture may introduce a specification for part of the runtime environment, such as a wrapper for remote procedure calls that will replace the standard one provided by the operating system.

5 First-Order Architectures

We want to leave open the choice of language for specifying an architecture. Therefore, we represent architectures as first-order theories, but our correctness and composition results in no way depend on this choice.

The representation of the dataflow and the shared-memory architectures in Figures 1 and 3, respectively, depend on the styles used in their construction. The dataflow-style vocabulary contains predicates for describing functional components, ports, values associated with ports, dataflow channels, values associated with dataflow channels, and connections of channels to ports. More precisely, the following sorts denote the first-class objects in a dataflow theory: *channel*, *function*, *iport*, and *oport*. We also make use of sorts *bool* and *val*, where *val* denotes the set of all possible values. The dataflow style has the following operations.

OutPort: oport $\times$ function $\rightarrow$ bool
Supplies: oport $\times$ val $\rightarrow$ bool
InPort: iport $\times$ function $\rightarrow$ bool
Accepts: iport $\times$ val $\rightarrow$ bool
Carries: channel $\times$ val $\rightarrow$ bool
Connects: channel $\times$ oport $\times$ iport $\rightarrow$ bool

The number of functions, ports, and channels that can appear in a particular architecture is unbounded. We do not bother to state the general well-formedness axioms associated with this style, or with others. An example of a general dataflow axiom is that every function must have at least one port.

The shared-memory style uses the reading and writing of a variable for intercommunication. Shared-variable communication is modeled using a call site as an interface between a function and the shared variable.[5] A call site serves the same purpose as a port in the dataflow style. The name of every different call site must be unique. The shared-memory style has the following style-specific sorts: *variable* denotes the set of all possible variables and *site* denotes the set of all possible call sites of which there are two kinds. The sort *rsite* denotes the sites that read, or input, values; the sort *wsite* denotes the ones the write, or output, values. The signature for the shared-memory style is

Holds: variable $\times$ val $\rightarrow$ bool
CallSite: site $\times$ function $\rightarrow$ bool
Writes: wsite $\times$ variable $\rightarrow$ bool
Puts: wsite $\times$ val $\rightarrow$ bool
Reads: rsite $\times$ variable $\rightarrow$ bool
Gets: rsite $\times$ val $\rightarrow$ bool

Table 1 contains (partial) theories associated with the two architectures in Figures 1 and 3. Θ_D denotes the dataflow theory and Θ_M the shared-memory theory. Dataflow theory Θ_D says that the parser and analyzer are functional components, the parser's output port can supply values of type *ast*, the analyzer's input

[5] We could have chosen not to model call sites or some equivalent interface object. We made the decision in order to simplify the style mapping from dataflow to shared-memory.

port can accept values of type *ast*, the dataflow channel can transmit values of type *ast*, and the channel is wired to the ports. The shared-memory theory Θ_M replaces ports with call sites, introduces a variable that can hold values of type *ast*, and employs read and write operations on the variable.

6 Mappings

It is useful to distinguish between two kinds of mappings.

- An *name mapping* associates the objects declared in an abstract architecture with objects declared in a concrete architecture.
- A *style mapping* says how the constructs of the abstract-level style can be implemented in terms of the constructs of the concrete-level style. More specifically, it maps all atomic formulas of the abstract-level theory to formulas of the concrete-level theory.

The two are combined to form an interpretation mapping.

6.1 Name Mapping

We saw a specification of the intended associations between the objects in the two architectures earlier. The only difference in the formal mapping is that we introduce the implicit call sites. Let I_N be name mapping

$$\begin{aligned} oast &\mapsto site_1 \\ iast &\mapsto site_2 \\ ast\_channel &\mapsto tree \end{aligned}$$

which relates the two architectures. The domain of a name mapping can be extended to include all abstract-level terms by mapping variables to themselves.

6.2 Style Mapping

Let I_S denote the style mapping in Figure 4 from the dataflow style to the shared-memory style. The t_i denote terms, which in our examples are restricted to logical constants and variables.[6] The last association specifies the implementation strategy. It says that any instance of $Connects(t_1, t_2, t_3)$ can be implemented by having call site t_2, corresponding to output port t_2, be the interface point that provides the values used in the writing of variable t_1, corresponding to channel t_1. On the receiving end of a transmission, input port and call site t_3 serve the same function. The other associations say that channels are mapped to variables, that output ports are mapped to calls that supply values, and that input ports are mapped to calls that receive values. The *Puts* and *Gets* predicates ensure that the right kind of site is associated with the each kind of port.

[6]Note that our languages contain no function symbols. A treatment of them can be found in [6].

6.3 Interpretation Mapping

An ***interpretation mapping*** I is determined from a name mapping I_N and a style mapping I_S, as follows: for every predicate P, all terms $t_1, t_2, \ldots, t_n$, every variable x, and all formulas F and G of the abstract language,

$$\begin{aligned} I(P(t_1,t_2,\ldots,t_n)) &= I_S(P(I_N(t_1), I_N(t_2), \ldots, I_N(t_n)) \\ I(\neg F) &= \neg(I(F)) \\ I(F \wedge G) &= I(F) \wedge I(G) \\ I(F \vee G) &= I(F) \vee I(G) \\ I(F \supset G) &= I(F) \supset I(G) \\ I(\forall x F) &= \forall x I(F)^{7} \\ I(\exists x F) &= \exists x I(F) \end{aligned}$$

Let I^D_M denote the interpretation mapping from theory Θ_D to theory Θ_M. Both the ground facts and general axioms in Θ_D must be mapped. For example,

$$\begin{aligned} & I^D_M(Connects(ast\_channel, oast, iast) \\ &= I_S(Connects(I_N(ast\_channel), \\ &\qquad I_N(oast), I_N(iast))) \\ &= I_S(Connects(tree, site_1, site_2)) \\ &= Writes(site_1, tree) \wedge Reads(site_2, tree) \end{aligned}$$

which is the intended implementation.

7 Proof Obligations

A relative correctness proof involves two steps. First, we must prove the correctness of the relevant style mapping. The proof is performed only once; it need not be repeated when the two styles are used. Second, we must demonstrate the relative correctness of the two architectures with respect to the interpretatation mapping formed using the two styles.

7.1 Proof of a Style Mapping

The crucial part of the proof is concerned with the validity of the connector mapping. We would like to know that a dataflow connection can be implemented by the reading and writing of a shared memory location, which is modeled as a variable. This requires a definition of the semantics of both forms of connection. We choose an axiomatic style of semantic definition suitable for describing both safety and fairness properties.

[7]In general, the range of quantifiers must be restricted to a subset of the concrete domain, see [6]. But no restriction is required for our example, because every concrete-level object implements an abstract-level object.

Θ_D	Θ_M
$Function(parser)$	$Function(parser)$
$Function(analyzer)$	$Function(analyzer)$
$OutPort(oast, parser)$	$Variable(tree)$
$\forall v[Supplies(oast, v) \supset ast(v)]$	$\forall v[ast(v) \supset Holds(tree, v)]$
$InPort(iast, analyzer)$	$CallSite(site_1, parser)$
$\forall v[ast(v) \supset Accepts(iast, v)]$	$\forall v[Puts(site_1, v) \supset ast(v)]$
$Channel(ast\_channel)$	$Writes(parser, tree)$
$\forall v[ast(v) \supset Carries(ast\_channel, v)]$	$CallSite(site_2, analyzer)$
$Connects(ast\_channel, oast, iast)$	$\forall v[ast(v) \supset Gets(site_2, v)]$
	$Reads(analyzer, tree)$

Table 1: Partial Dataflow and Shared-Memory Theories

$$\begin{array}{rcl}
Function(t_1) & \mapsto & Function(t_1) \\
OutPort(t_1, t_2) & \mapsto & CallSite(t_1, t_2) \land \exists v Puts(t_1, v) \\
Supplies(t_1, t_2) & \mapsto & Puts(t_1, t_2) \\
InPort(t_1, t_2) & \mapsto & CallSite(t_1, t_2) \land \exists v Gets(t_1, v) \\
Accepts(t_1, t_2) & \mapsto & Gets(t_1, t_2) \\
Channel(t_1) & \mapsto & Variable(t_1) \\
Carries(t_1, t_2) & \mapsto & Holds(t_1, t_2) \\
Connects(t_1, t_2, t_3) & \mapsto & Writes(t_2, t_1) \land Reads(t_3, t_1)
\end{array}$$

Figure 4: A Style Mapping

In particular, we use a temporal logic, called the Temporal Logic of Actions (TLA) [11], to define dataflow and shared-memory communication:

- The semantics of dataflow places minimal restrictions on communication. It says that a multiset of values is transmitted between components. Values can be "lost" and out of order. The fairness condition is that eventually a send or receive occurs unless both are impossible. One reason for impossibility could be failure of the communications line.
- The semantics of shared memory requires that tranmission preserve ordering and that values cannot be lost. The fairness condition is that all values written into shared memory will eventually be read from the memory if it is possible to read them.

For comparison purposes, the appendix contains an operational definition of the two forms of communication in standard CSP [8], following Allen and Garlan [2]. CSP can be used to model the safety properties, but not the fairness properties.

We formalize the semantics of dataflow and shared-memory connections as TLA theories. We define an interpretation mapping $\mathcal{I}_M^D$ from the dataflow semantics to the shared-memory semantics and show that it is a theory interpretation. This is sufficient to establish that dataflow can be implemented with a single shared memory location and that, if the shared-memory communication is fair, the dataflow communication is fair.

We make use of the following TLA notation.

Notation	Meaning
f	list of variables in the old state
f'	list of variables in the new state
$\mathcal{A}$	action—relation between old and new states
Enabled	possible to perform action
$[\mathcal{A}]_f$	$\mathcal{A} \lor (f' = f)$
$\langle\mathcal{A}\rangle_f$	$\mathcal{A} \land (f' \neq f)$
$\Box F$	always F
$\Diamond F$	$\neg\Box\neg F$ (sometimes F)
$\mathrm{WF}_f(\mathcal{A})$	$\Box\Diamond\langle\mathcal{A}\rangle_f \lor \Box\Diamond\neg Enabled\ \langle\mathcal{A}\rangle_f$

The last line says that eventually action $\mathcal{A}$ must either be taken or become impossible to take. For example, a precondition for execution may not be satisfiable.

In the proof, we make use of two TLA inference rules.

$$\text{STL4.} \quad \frac{F \supset G}{\Box F \supset \Box G}$$

where F and G are temporal formulas, says that, if F implies G, the always F implies always G.

$$\text{TLA2.} \quad \frac{[\mathcal{A}]_f \supset [\mathcal{B}]_g}{\Box[\mathcal{A}]_f \supset \Box[\mathcal{B}]_g}$$

is a simplification of Lamport's TLA2 axiom that suffices for our purposes. It says that, if action $\mathcal{A}$ implies $\mathcal{B}$, then always $\mathcal{A}$ implies always $\mathcal{B}$.

Figures 5 and 6 contain the TLA theories of dataflow and shared-memory, respectively. The quoted boldface symbols are logical constants. In Figure 5, the dataflow connector is denoted by the *flow* state function, which is a multiset, with three operators: *with* is the insertion operator, *less* is the deletion operator, and *choose* is used to select an element from a nonempty multiset. Values carried by the connector must be in set **Type**, the set of all possible values. The dataflow semantic theory is defined to be Φ, which says three things: the dataflow has to start in the initial state, it must always be possible to perform a send or a receive operation, and the communication line eventually responds to send and receive requests if it is possible to do so (fairness). The shared-memory semantic theory, called Ψ, is defined in a similar manner.

$$
\begin{array}{lcll}
Init_{\Phi} & \stackrel{\text{def}}{=} & & ev = \text{“ready”} \\
 & & \wedge & flow = \text{“emptybag”} \\
\mathcal{S}_{sender} & \stackrel{\text{def}}{=} & & ev = \text{“ready”} \\
 & & \wedge & ev' = \text{“send”} \\
 & & \wedge & flow' = flow \\
 & & \wedge & val' \in \mathbf{Type} \\
\mathcal{R}_{receiver} & \stackrel{\text{def}}{=} & & ev = \text{“ready”} \\
 & & \wedge & ev' = \text{“receive”} \\
 & & \wedge & flow' = flow \\
 & & \wedge & val' = val \\
\mathcal{S}_{flow} & \stackrel{\text{def}}{=} & & ev = \text{“send”} \\
 & & \wedge & ev' = \text{“ready”} \\
 & & \wedge & flow' = flow \ with \ val' \\
 & & \wedge & val' = val \\
\mathcal{R}_{flow} & \stackrel{\text{def}}{=} & & ev = \text{“receive”} \\
 & & \wedge & ev' = \text{“ready”} \\
 & & \wedge & flow \neq \text{“emptybag”} \\
 & & \wedge & val' = choose(flow) \\
 & & \wedge & flow' = flow \ less \ val' \\
\mathcal{N}_{flow} & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{\mathcal{S}_{flow} \vee \mathcal{R}_{flow}} \\
\mathcal{N} & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{\mathcal{N}_{flow} \vee \mathcal{S}_{sender} \vee \mathcal{R}_{receiver}} \\
w & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{(ev, val, flow)} \\
\Phi & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{(\exists flow)(Init_{\Phi} \wedge \Box[\mathcal{N}]_w \wedge \mathrm{WF}_w(\mathcal{N}_{flow}))}
\end{array}
$$

Figure 5: Semantics of Dataflow

Interpretation mapping $\mathcal{I}_M^D$ maps constants, state functions, and operators of the dataflow semantics to those of the shared-memory semantics. $\mathcal{I}_M^D$ is defined by

$$
\begin{array}{lcll}
Init_{\Psi} & \stackrel{\text{def}}{=} & & op = \text{“ready\_write”} \\
 & & \wedge & mem = \text{“undefined”} \\
\mathcal{W}_{writer} & \stackrel{\text{def}}{=} & & op = \text{“ready\_write”} \\
 & & \wedge & op' = \text{“write”} \\
 & & \wedge & mem' = mem \\
 & & \wedge & val' \in \mathbf{Type} \\
\mathcal{R}_{reader} & \stackrel{\text{def}}{=} & & op = \text{“ready\_read”} \\
 & & \wedge & op' = \text{“read”} \\
 & & \wedge & mem' = mem \\
 & & \wedge & val' = val \\
\mathcal{W}_{mem} & \stackrel{\text{def}}{=} & & op = \text{“write”} \\
 & & \wedge & op' = \text{“ready\_read”} \\
 & & \wedge & mem' = val' \\
 & & \wedge & val' = val \\
\mathcal{R}_{mem} & \stackrel{\text{def}}{=} & & op = \text{“read”} \\
 & & \wedge & op' = \text{“ready\_write”} \\
 & & \wedge & mem \neq \text{“undefined”} \\
 & & \wedge & mem' = val' \\
 & & \wedge & val' = mem \\
\mathcal{M}_{mem} & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{\mathcal{W}_{mem} \vee \mathcal{R}_{mem}} \\
\mathcal{M} & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{\mathcal{M}_{mem} \vee \mathcal{W}_{writer} \vee \mathcal{R}_{reader}} \\
u & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{(op, val, mem)} \\
\Psi & \stackrel{\text{def}}{=} & \multicolumn{2}{l}{Init_{\Psi} \wedge \Box[\mathcal{M}]_u \wedge \mathrm{WF}_u(\mathcal{M}_{mem})}
\end{array}
$$

Figure 6: Semantics of Shared Memory

$$
\begin{array}{lcl}
ev & \mapsto & op \\
flow & \mapsto & mem \\
\text{“emptybag”} & \mapsto & \text{“undefined”} \\
\text{“ready”} & \mapsto & either(\text{“ready\_write”}, \\
 & & \quad \text{“ready\_read”}) \\
\text{“send”} & \mapsto & \text{“write”} \\
\text{“receive”} & \mapsto & \text{“read”} \\
t_1 \ with \ t_2 & \mapsto & t_2 \\
t_1 \ less \ t_2 & \mapsto & t_2 \\
choose(t_1) & \mapsto & t_1
\end{array}
$$

where t_1 and t_2 are terms. The last three associations interpret multiset operations in the context of our specific weak fairness condition on shared memory.

To show that $\mathcal{I}_M^D$ is a theory interpretation, we need to prove that $\Psi \supset \mathcal{I}_M^D(\Phi)$. The first step is to prove that

$$Init_{\Psi} \supset \mathcal{I}_M^D(Init_{\Phi}). \qquad (1)$$

Applying $\mathcal{I}_M^D$ to $Init_{\Phi}$ we get:

$$op = either(\text{“ready\_write”}, \text{“ready\_read”}) \wedge mem = \text{“undefined”}.$$

Hence (1) holds. The second step is to show that

$$\Box[\mathcal{M}]_u \supset \mathcal{I}_M^D(\Box[\mathcal{N}]_w). \qquad (2)$$

We can easily show that

$$\begin{array}{rcll} \mathcal{W}_{writer} & \supset & \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\mathcal{S}_{sender}) & (3)\\ \mathcal{R}_{reader} & \supset & \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\mathcal{R}_{receiver}) & (4)\\ \mathcal{W}_{mem} & \supset & \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\mathcal{S}_{flow}) & (5)\\ \mathcal{R}_{mem} & \supset & \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\mathcal{R}_{flow}) & (6) \end{array}$$

from which we infer that

$$[\mathcal{M}]_u \supset \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}([\mathcal{N}]_w).$$

Applying rule TLA2, we conclude that (2) holds. The third step is to show that

$$\mathrm{WF}_u(\mathcal{M}_{mem}) \supset \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\mathrm{WF}_w(\mathcal{N}_{flow})). \qquad (7)$$

From (3)–(6), we get

$$\langle \mathcal{M}_{mem} \rangle_u \supset \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\langle \mathcal{N}_{flow} \rangle_w).$$

Applying rule STL4 twice and the definition of $\Diamond$, we get

$$\Box\Diamond\langle \mathcal{M}_{mem} \rangle_u \supset \Box\Diamond\mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\langle \mathcal{N}_{flow} \rangle_w).$$

From the definition of *Enabled*, we have

$$Enabled\ \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\langle \mathcal{N}_{flow} \rangle_w) \supset Enabled\ \langle \mathcal{M}_{mem} \rangle_u.$$

Since

$$\mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(Enabled\ \langle \mathcal{N}_{flow} \rangle_w) \supset Enabled\ \mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(\langle \mathcal{N}_{flow} \rangle_w),$$

we apply rule STL4 to get

$$\Box\Diamond\neg Enabled\ \langle \mathcal{M}_{mem} \rangle_u \supset \Box\Diamond\neg\mathcal{I}_{\mathrm{M}}^{\mathrm{D}}(Enabled\ \langle \mathcal{N}_{flow} \rangle_w),$$

from which we conclude that fairness condition (7) holds.

7.2 Relative Correctness Proof

We must show that $I_{\mathrm{M}}^{\mathrm{D}}$ is a theory interpretation and that it is faithful. A proof of the former is straightforward. For example, under $I_{\mathrm{M}}^{\mathrm{D}}$ the axiom

$$Connects(ast\_channel, oast, iast)$$

is interpreted as

$$\begin{array}{l} Writes(parser, tree)\\ \quad \wedge\ Reads(analyzer, tree) \end{array}$$

which is a theorem that follows directly from Θ_{M}.

To show faithfulness, notice that $I_{\mathrm{M}}^{\mathrm{D}}$ induces a mapping I' from shared-memory structures to dataflow structures as follows. If $I_{\mathrm{M}}^{\mathrm{D}}$ maps atomic dataflow formula $P(\bar{x})$ to shared-memory formula F, then I' assigns to dataflow predicate P the set of shared-memory tuples that satisfy F.

Given a model D of Θ_{D}, we can construct a model M of Θ_{M} as follows. The universe of M is the same as D. The assignment to predicates by M is defined as:

$$\begin{array}{rcl} Function & = & \{a \in |D| : D \models Function(a)\}\\ Variable & = & \{a \in |D| : D \models Channel(a)\}\\ Writes & = & \{(a,b) \in |D|^2 : \exists c, d \in |D|\\ & & \quad [D \models OutPort(c,a) \wedge\\ & & \quad\ Connects(b,c,d)]\}\\ & \vdots & \end{array}$$

By a theorem stated in [15] and proved in [16], the fact that induced mapping I' maps M back to D is enough to conclude that $I_{\mathrm{M}}^{\mathrm{D}}$ is faithful.

8 Composing Architectures

A useful form of architecture composition is illustrated in Figure 7. We want to compose two architectures, called "subsystem A" and "subsystem B", into a single system architecture. We construct a new architecture with components "A" and "B" connected through new interfaces. If two conditions are satisfied, the three architectures can be combined to form a composite system that is correct if the three subsystems are.

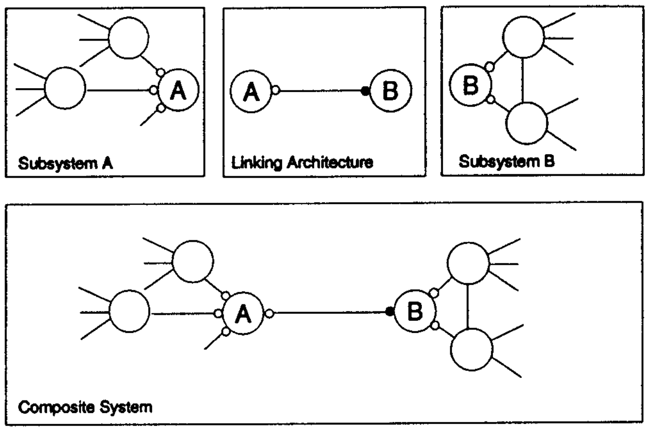

Figure 7: Illustration of Subsystem Composition

Let Θ_1 and Θ_2 be theories that represent two abstract architectures. Let Θ_1' and Θ_2' be concrete theories intended to implement Θ_1 and Θ_2, respectively. Two pairs of architecture theories can be composed only in ways that preserve faithfulness. More precisely, if

$$I_1\colon \Theta_1 \to \Theta_1' \quad \text{and} \quad I_2\colon \Theta_2 \to \Theta_2'$$

are faithful interpretations, then we want

$$I_1 \cup I_2\colon \Theta_1 \cup \Theta_2 \to \Theta_1' \cup \Theta_2'$$

to be a faithful interpretation. (The union of two theories is the deductive closure of the set-theoretic union of the theories.)

This property holds provided two general conditions are satisfied.

1. The composite interpretation mapping must be a function. For a sentence F, we require that

   $$\forall F \in \Theta_1 \cap \Theta_2 \; [I_1(F) = I_2(F)]$$

   which guarantees that interpretation mappings I_1 and I_2 agree on shared objects and shared style constructs.

2. It must not be possible to infer new facts about the composite abstract architecture from the composite concrete architecture. That is, for language L_1 of Θ_1 and L_2 of Θ_2, if

   $$F \text{ is a sentence of } L_1 \cup L_2$$

   and

   $$\Theta_1' \cup \Theta_2' \vdash I(F)$$

   then we must prove that

   $$I(\Theta_1) \cup I(\Theta_2) \vdash I(F).$$

The intuition behind the second condition can be illustrated by means of a simple example. Consider an architecture in which there is a dataflow connection from A to B and another architecture that has dataflow connection from B to C. Suppose that both flows are implemented correctly in concrete architectures, but that in one A writes some variable x and in the other C reads a variable x. Each implementation is correct, since neither introduces a new dataflow. However, the composite concrete architecture reads and writes x, from which we can infer an entirely new abstract dataflow connection from A to C. Consequently, the composite abstract architecture is not faithfully interpreted (by the composite mapping) in the composite concrete architecture (under the original assumption that dataflow is intransitive).

Although the second condition is a rather strong logically, it appears to be flexible enough for architecture composition. The form of composition illustrated in Figure 7 can be handled easily by allowing two abstract architectures to share only one component and possibly its interface points. Styles can be shared but no other objects. These constraints guarantee that the two conditions above are satisfied, and the desired composition can be performed in two steps.

Another useful form of composition is the chaining together of a sequence of correct architectures. Since faithful interpretation is transitive, intermediate architectures can be omitted in the development of a concrete architecture. Intermediate architectures arise because we make explicit all important intermediate steps in a development, even if they correspond to small architectural changes. The intermediate architectures need not be explicit as long as there is a sequence of instances of refinement patterns that connect the first (most abstract) and last (most concrete) architectures in the sequence.

We return to the compiler architecture in Figure 2 to give a specific example of composition. We proved that the dataflow connection between the parser and the analyzer is implemented correctly by means of the reading and writing of the tree. That is, we showed that dataflow theory Θ_{D} is implemented correctly by theory Θ_{M} with respect to mapping I^{D}_{M}. Similarly, we can show that the dataflow connection from the lexical analyzer to the parser is correctly implemented by the pipeline connection in the concrete architecture. The two abstract-concrete pairs of architectures share a common component, the parser, but no interface points. Therefore, our second condition is satisfied and we can compose the two pairs directly. (The two mappings are constructed to meet the first condition.) No linking architecture is needed.

9 Related Work

The utility of architecture hierarchies was recognized in the 1970s, but architecture hierarchy was studied only informally at that time. Several notations were developed for describing architectures, including those of Jackson [10], Yourdan and Constantine [17], and DeMarco [5], but little attention was given to understanding the relationship between levels of abstraction.

Moriconi and Hare [14] formalized a relationship between levels in a hierarchy and used the technique of Hoare [9] to prove the relative correctness of two stylistically different architectures. Hoare's technique involves a proof of only theory interpretation, and not of faithfulness. They were the first to introduce a completeness assumption for architectures. An architecture was allowed to contain only finitely many objects (constants), which enabled them to fully mechanize correctness proofs. The completeness assumption, as formalized in this paper, applies equally well to infinite architectures. For example, it is possible to quantify over infinite types (such as integers) and to reason about dynamic architectures with an unbounded number of processes.

The technique of Hoare has been applied more recently to architecture by Broy [4], Brinksma [3], and others. Broy's component refinements turn out to be conservative because interface signatures are preserved, but his connection refinements may not be because additional flows could be added to a channel. Brinksma justifies channel splitting on the basis of behavioral reasoning; application of his rule can violate the completeness assumption.

A Hoare-style representation mapping has been applied to dynamic architectures by Luckham et al [12, 13]. A language called Rapide is used to define executable architectures based on distributed event processing. Mappings relate concrete events to abstract events and are used as the basis for comparative simulation, a technique that complements ours.

The problem of composition of specifications has been studied in a general semantic framework by Abadi and Lamport [1]. Their results are applicable to any domain, whereas our results are syntactic and specialized to the domain of software architecture. The advantage of a syntactic constraint is that it can be checked easily. The disadvantage is that it is more restrictive than semantic composition. Broy [4] gives three operators for composing functional-style architectures, but does not consider the composition of architectures involving multiple styles.

10 Conclusion

An architecture for a large, complex system, and even some simple systems, will involve multiple levels of detail expressed in multiple architectural styles. The novel contributions of the work reported here are:

- A formal criterion for proving that one architecture implements another architecture, even if they are described in different architectural styles. A change in the representation of a component, an interface, or a connector is handled, but a change in the representation of a type requires a slightly different criterion.

- A decomposition of the mapping between architectures into type-level properties that are proved once for every pair of styles and instance-level properties that are proved for every pair of architectures. The importance of this decomposition was underscored by a proof that the connectors of a common concrete style implement the connectors of a common abstract style. The proof was somewhat complicated, establishing both safety and fairness properties, but it does not need to be repeated each time the styles are used.

- Syntactic criteria for composing architectures such that the composition of two correct architectures is correct. One specific composition operator, which is useful for putting together subsystems, allows two architectures to be composed provided they share only components and their interface points. Another composition operator is used to eliminate intermediate levels in an architecture hierarchy.

Our approach applies to any logic used to represent an architecture; it does not depend on a particular architecture definition language or a particular kind of connector semantics. A more comprehensive treatment of the formal techniques in this paper can be found in a companion paper [15].

The work reported here may have implications in several subareas of software-architecture research.

- **Language design.** An architecture definition language (ADL) should treat all refineable objects, including components, interface points, and connectors, as first-class in the sense that they should be named objects with independent meaning. Another implication is that an ADL should make it impossible to subvert the completeness assumption. For example, an ADL type system should not allow components to be values, which would allow interactions to be created indirectly. The last implication is that an ADL should support the specification of two kinds of mappings: style mappings and name mappings between architectures.

- **Refinement methodology.** It seems clear that after-the-fact proof of an architecture hierarchy will be very difficult. This is true primarily because of the need to establish conservativeness (modulo renaming). An incremental development strategy that minimizes the number and difficulty of architecture-specific proofs is needed. One candidate approach involving correctness-preserving architectural transformations is described in [15].

- **Style design.** Styles are an important vehicle for organizing reusable architectural design information. We showed that the specification of style mappings is a key element of style design, and that the semantics of a style can be affected by how the style is intended to be used in relation to other styles.

REFERENCES

[1] M. Abadi and L. Lamport, "Composing Specifications", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 1, January 1993, pp. 73–132.

[2] R. Allen and D. Garlan, "Formalizing Architectural Connection", *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994, pp. 71–80.

[3] E. Brinksma, B. Jonsson, and F. Orava, "Refining Interfaces of Communicating Systems", *TAPSOFT'91: Lecture Notes in Computer Science 494*, S. Abramsky and T.S.E. Maibaum, Eds., Springer-Verlag, 1991, pp. 297–312.

[4] M. Broy, "Compositional Refinement of Interactive Systems", No. 89, Digital Systems Research Center, Palo Alto, California, July 1992.

[5] T. DeMarco, *Structured Analysis and System Specification*, Yourdan Press, 1979.

[6] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.

[7] D. Garlan and M. Shaw, "An Introduction to Software Architecture", In *Advances in Software Engineering and Knowledge Engineering*, Volume 1, V. Ambriola and G. Tortora, Eds., World Scientific Publishing Company, 1993.

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

[9] C.A.R. Hoare, "Proof of correctness of data representations", *Acta Informatica*, Vol. 1, No. 4, 1972, pp. 271–281.

[10] M.A. Jackson, *Principles of Program Design*, Academic Press, 1975.

[11] L. Lamport, "The Temporal Logic of Actions", Technical Report 79, Digital Systems Research Center, Palo Alto, California, December 1991. (to appear in *ACM Transactions on Programming Languages and Systems*)

[12] D.C. Luckham, L.M. Augustin, J.J. Kenney, J.S. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", to appear in *IEEE Transactions on Software Engineering.*

[13] D.C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz", "Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems", *Journal of Systems and Software*, Vol. 21, No. 3, June 1993, pp. 253–265.

[14] M. Moriconi and D.F. Hare, "The PegaSys System: Pictures as Formal Documentation of Large Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, October 1986, pp. 524–546.

[15] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct Architecture Refinement", to appear in *IEEE Transactions on Software Engineering.*

[16] M. Moriconi, X. Qian, and R. Riemenschneider, "A Formal Approach to Correct Refinement of Software Architectures", Technical Report SRI-CSL-94-13, Computer Science Laboratory, SRI International, Menlo Park, California, August 1994.

[17] E. Yourdan and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

A Proof of Connector Mapping in CSP

We can define the semantics of the dataflow and shared memory styles in CSP [8], following Allen and Garlan [2]. We make use of the following CSP notation.

Notation	Meaning
αP	the alphabet of process P
$P \| Q$	P in parallel with Q
$a \rightarrow P$	a then P
$a \rightarrow P \mid b \rightarrow Q$	a then P choice b then Q $(a \neq b)$
$P \backslash C$	P without C (hiding)
$f : A \rightarrow B$	f is a function mapping A to B

We also make use of the count process CT, defined as follows.

$$\begin{aligned} CT_0 &= (up \rightarrow CT_1 \mid around \rightarrow CT_0) \\ CT_{n+1} &= (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n) \end{aligned}$$

The CSP semantics is essentially the same as the TLA semantics. However, a connector is modeled directly in TLA by a state function. It is modeled indirectly in CSP as a process, which essentially computes the state function. Standard CSP cannot be used to express fairness of the kind in our example. Therefore, we prove only safety.

The CSP semantics for the dataflow style is

$$\begin{aligned} \text{DFS} &= \text{Sender} \parallel \text{Receiver} \parallel \text{Flow} \\ \alpha\text{Sender} &= \{\text{oport}\} \\ \text{Sender} &= \text{oport} \rightarrow \text{Sender} \\ \alpha\text{Receiver} &= \{\text{iport}\} \\ \text{Receiver} &= \text{iport} \rightarrow \text{Receiver} \\ \alpha\text{Flow} &= \{\text{oport, iport}\} \\ \text{Flow} &= (\text{CT}_0 \parallel \text{Flow'}) \backslash \{\text{around, down, up}\} \\ \alpha\text{Flow'} &= \{\text{around, down, up, oport, iport}\} \\ \text{Flow'} &= \text{oport} \rightarrow \text{up} \rightarrow \text{Flow'} \\ &\quad \mid \text{around} \rightarrow \text{Flow'} \\ &\quad \mid \text{down} \rightarrow \text{iport} \rightarrow \text{Flow'} \end{aligned}$$

and the CSP semantics for the shared-memory style is

$$\begin{aligned} \text{SMS} &= \text{Writer} \parallel \text{Reader} \parallel \text{Var} \\ \alpha\text{Writer} &= \{\text{write}\} \\ \text{Writer} &= \text{write} \rightarrow \text{Writer} \\ \alpha\text{Reader} &= \{\text{read}\} \\ \text{Reader} &= \text{read} \rightarrow \text{Reader} \\ \alpha\text{Var} &= \{\text{write, read}\} \\ \text{Var} &= \text{write} \rightarrow \text{read} \rightarrow \text{Var} \end{aligned}$$

We must show that the shared-memory style is a correct implementation of the dataflow style. Intuitively, every behavior of the shared-memory style should correspond to an allowable behavior of the dataflow style. Since the alphabets of the two styles are different, this can be done using the CSP change-of-symbol operator f: $f(write) = oport$ and $f(read) = iport$. Hence, the correctness proof amounts to showing that $f(SMS) \sqsubseteq DFS$, which is straightforward.