Appeared in *IEEE Transactions on Software Engineering*,
April, 1995, Volume 21, Number 4, pp. 356–372.

Correct Architecture Refinement

Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider

Abstract— **A method is presented for the stepwise refinement of an abstract architecture into a relatively correct lower-level architecture that is intended to implement it. A refinement step involves the application of a predefined refinement pattern that provides a routine solution to a standard architectural design problem. A pattern contains an abstract architecture schema and a more detailed schema intended to implement it. The two schemas usually contain very different architectural concepts (from different architectural styles). Once a refinement pattern is proven correct, instances of it can be used without proof in developing specific architectures. Individual refinements are compositional, permitting incremental development and local reasoning. A special correctness criterion is defined for the domain of software architecture, as well as an accompanying proof technique. A useful syntactic form of correct composition is defined. The main points are illustrated by means of familiar architectures for a compiler. A prototype implementation of the method has been used successfully in a real application.**

Keywords— **Software architecture, hierarchy, stepwise refinement, refinement patterns, formal methods, relative correctness, composition**

I. Introduction

DECISIONS about the architecture of a software system can have a major impact on system efficiency, maintainability, and evolvability. Architectural decisions typically are documented in terms of the ubiquitous box-and-arrow diagrams. Practicing engineers interpret the diagrams with respect to common architectural styles, such as dataflow, pipe-and-filter, batch-sequential, blackboard, implicit invocation (event-based), and client-server.

For a large system, its architecture often is described by a hierarchy of related architectures. An architecture hierarchy is a linear sequence of two or more individual architectures that may differ with respect to the number and kind of components and connections among them. 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. In general, an abstract architecture is smaller and easier to understand; a concrete architecture reflects more implementation concerns.

The utility of an architecture hierarchy is severely limited by the current level of informality. Individual architectures may be ambiguous, allowing multiple and perhaps unintended interpretations. The mapping between architectures in the hierarchy is partially specified, if at all, making it impossible to accurately trace the lineage of implementation decisions. The analysis of architecture is limited to syntactic checks. It is not possible to check semantic properties of an architecture, such as the safety and fairness of its connections, or to check the relative correctness of two architectures in the hierarchy. Consequently, a concrete architecture may erroneously be seen as an implementation of a more abstract architecture.

The main contribution in this paper is a methodology for the correct stepwise refinement of software architectures. It is expected to lead to fewer architectural design errors, to extensive and systematic reuse of design knowledge and proofs, and ultimately to an architecture synthesis tool similar to those now used for integrated circuit design. The methodology involves the use of instances of architecture refinement patterns that are correctness preserving and compositional.

A refinement pattern provides a routine solution to a standard architectural design problem. For example, a pattern may show how to implement a single dataflow connection in shared memory, or several patterns may combine to implement dataflow diagrams in terms of some form of client/server architecture. A pattern contains a pair of architecture schemas that are proved to be relatively correct with respect to a given mapping schema between them. The proof is performed only once; every instance of a refinement pattern is guaranteed to be correct. A schema can be homogeneous (consisting of one style) or heterogeneous (consisting of multiple styles). The two schemas in a refinement pattern may, and usually do, contain concepts from different architectural styles.

A useful form of correctness-preserving composition is defined that applies to both individual refinements and existing architectures. The latter is important because we want to be able to assemble existing subsystem architectures into a single system. Two architectures can be composed even if their vocabularies are not disjoint. In general, "horizontal" composition requires a case-by-case proof of correctness. However, we define a simple syntactic criterion that, if satisfied, guarantees compositionality. Because our correctness relation is transitive, the "vertical" composition of levels in an architecture hierarchy preserves correctness, and we are guaranteed that the most concrete architecture in the hierarchy meets the requirements of the most abstract architecture in the hierarchy.

The correctness of architecture refinement and composition involves a special correctness criterion, which is stronger than the usual one for functional refinement, and a special mapping between architectures, that is more complex than the usual mapping between data structures. A mapping between architectures involves an extensive translation in which the representation of components, interfaces, and connections may change and, moreover, these abstract objects may be aggregated, decomposed, or elim-

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

The authors are with the Computer Science Laboratory, SRI International, Menlo Park, California 94025. Email: {moriconi, qian, rar}@csl.sri.com.

inated in the concrete architecture.

A stronger correctness criterion is needed because of the potential uses of architectures. Consider the role an architecture can play in reducing the time to provide fixes, optimizations, and upgrades to systems in deployment. If the architecture accurately models the implementation, it can be used to focus and explore the consequences of changes to the implementation. But if the implementation contains connections that do not appear in the architecture, a developer could easily be misled into making changes that appear to be minor and localized but that, in fact, have widespread consequences. For example, we may specify a pipeline architecture, restricting the system topology to a linear sequence of filters, to facilitate component reusability. If the concrete architecture implements the pipeline, but additionally introduces feedback loops, the *raison d'être* behind the original pipeline architecture is no longer valid. In general, the preservation of "communication integrity" is integral to the utility of an architecture.

Therefore, an architecture should describe explicitly the components, interfaces, and connections that are required of the target system, and perhaps more importantly, those that are *not* intended to appear in the target system. This observation leads to a *completeness assumption* about a given architecture, namely that an architecture contains *all* components, interfaces, and connections intended to be true of the architecture at its level of detail. If a fact is not explicit in the architecture, or deducible from it, we assume that it is not intended to be true of the architecture. In the pipeline example, we couple the linearity property with the completeness assumption to infer 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.

The completeness assumption requires that we 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. The standard method for reasoning about the relative correctness of two specifications is to show that the concrete specification logically implies the abstract specification under a given mapping between them. This allows an implementation to exhibit additional, unspecified behaviors, as long as the specified behavior is implemented. If the standard proof method is applied to architectures, there would be no guarantee that negative properties are preserved under refinement.

Fortunately, there is a well-understood mathematical property, called *faithful interpretation*, that can be adapted for our purposes. If a certain mapping between the two architectures is faithful, both the positive and the implicit negative facts in the abstract architecture are preserved in the concrete architecture. However, a proof of faithfulness is inherently hard, and we are not aware of any general proof technique in the literature. We introduce a systematic technique for proving faithfulness. The inherent complexity of such proofs is one reason why we advocate a methodology that makes use of preproved refinement patterns.

It is worth mentioning that an important consequence of the completeness assumption is that the standard stepwise refinement paradigm is unsound with respect to the correctness relation. Certain refinements of an architecture must be composed horizontally. Completed levels in an architecture hierarchy can be composed vertically.

This paper is organized as follows. The next section illustrates the refinement problem and our approach to a solution. Section III makes useful distinctions among architectural styles, architecture schemas, and instance architectures, and shows how they can be represented as logical theories. We use first-order theories, but our basic framework does not depend on a particular logic. By formalizing architectures and their properties in logic, our results can be applied to a large class of architecture definition languages. Sections IV, V, and VI discuss mappings, correctness, and composition, respectively.

Section VII presents several different refinement patterns that are used in Section VIII in the development of standard architectures for a compiler. The development includes both refinement and composition. Sections IX reports on a larger experiment involving an operational power-control system. Section X describes related work, and the last section summarizes our results, their implications, and makes suggestions for future work.

II. Illustration of Approach to Refinement

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 denotes 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**: A relation that defines a syntactical translation from the language of an abstract architecture to the language of a concrete architecture.
6. **Architectural style**: For the purposes of this paper, a style consists of a vocabulary of design elements, well-formedness constraints that determine how they can be used, and a semantic definition of the connectors associated with the style.

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.

Consider the standard dataflow architecture for a compiler that is depicted at the top of Figure 1. The diagram is intended to convey an intuitive feel for the architecture; it is not a formal description of the architecture. Boxes denote functional components and arrows denote direc-

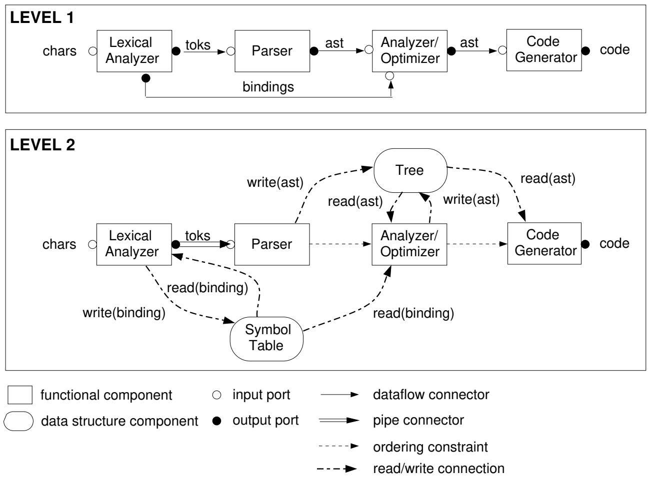

Fig. 1. Two architectures for a compiler

tional dataflow between ports. The labels on arrows denote types or value domains. A value cannot be transmitted between ports unless its type is compatible with the types of the ports. By the completeness assumption, this dataflow model of the compiler fixes its functional units, their interfaces, and the direction, source, and destination of all of its flows.

A textual specification of the dataflow architecture is contained in Figure 2. A dataflow component is a function with a signature describing its interface. Four dataflow connectors are declared to carry values of various types. The configuration assertions wire the connectors and interfaces together into a specific type-consistent architecture. The module imports various types and the functional and dataflow styles for use in the specification of the architecture.

A concrete architecture intended to implement the dataflow model of the compiler is depicted at the bottom of Figure 1. The concrete architecture is a hybrid that implements the dataflow style in terms of pipe-filter, batch-sequential, and shared-memory styles. Abstract signatures have been changed, dataflow connectors have been implemented in several ways, new components (data objects) are introduced, and precedence relations are added to preserve the original flows in the presence of shared-memory communication.[1] A textual specification of the level-2 architecture of the compiler can be found in the appendix.

We do not want to construct the level-1 and the level-2 architectures and then perform an after-the-fact correctness proof. Instead, we want to systematically and incrementally transform the level-1 architecture into the level-2 architecture. The level-2 architecture should be correct by construction, requiring no explicit proofs in its derivation. This can be accomplished through a series of small, local refinements, each of which involves the application of a correct refinement pattern. Then, the local refinements are combined to form the larger composite level-2 architecture, which is guaranteed to correctly implement the level-1 architecture.

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

```
compiler_L1: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
      IMPORT character, code, token, binding, ast
          FROM compiler_types
      IMPORT Function FROM Functional_Style
      IMPORT Dataflow_Channel, Connects
          FROM Dataflow_Style
   COMPONENTS
      lexical_analyzer: Function
         [char_iport: SEQ(character)
             -> token_oport: SEQ(token),
                bind_oport: SEQ(binding)]
      parser: Function
         [token_iport: SEQ(token)
             -> base_ast_oport: ast]
      analyzer_optimizer: Function
         [base_ast_iport: ast, bind_iport: SEQ(binding)
             -> full_ast_oport: ast]
      code_generator: Function
         [full_ast_iport: ast -> code_oport: code]
   CONNECTORS
      token_channel:    Dataflow_Channel[SEQ(token)]
      bind_channel:     Dataflow_Channel[SEQ(binding)]
      base_ast_channel: Dataflow_Channel[ast]
      full_ast_channel: Dataflow_Channel[ast]
   CONFIGURATION
      token_flow:
        Connects(token_channel, token_oport, token_iport)
      bind_flow:
        Connects(bind_channel, bind_oport, bind_iport)
      base_ast_flow:
        Connects(base_ast_channel,
                 base_ast_oport, base_ast_iport)
      full_ast_flow:
        Connects(full_ast_channel,
                 full_ast_oport, full_ast_iport)
END compiler_L1
```

Fig. 2. Specification of dataflow architecture for the compiler

As an illustration of our approach, consider the implementation of the dataflow channel between the parser and analyzer in terms of the reading and writing of a shared abstract syntax tree. More specifically, we propose to refine abstract subarchitecture

```
parser:             Function [ -> base_ast_oport: ast]
analyzer_optimizer: Function [base_ast_iport: ast -> ]
base_ast_channel:   Dataflow_Channel[ast]
base_ast_flow:
  Connects(base_ast_channel,
           base_ast_oport,  base_ast_iport)
```

into concrete subarchitecture

```
parser:               Function[ -> ]
analyzer_optimizer:   Function[ -> ]
abstract_syntax_tree: Variable[ast]
write_base_ast: Writes(parser, abstract_syntax_tree)
read_base_ast:
  Reads(analyzer_optimizer, abstract_syntax_tree)
```

For simplicity, the component signatures contain only the ports that are relevant to this refinement. The dataflow connection is implemented by a component (a shared variable containing the tree) and two connections (the read

```
PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[ -> ]
    COMPONENTS
        f1:  Functional_Style!Function[ -> op: t]
        f2:  Functional_Style!Function[ip: t -> ]
    CONNECTORS
        c:  Dataflow_Style!Dataflow_Channel[t]
    CONFIGURATION
        a:  Dataflow_Style!Connects(c, op, ip)
PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[ -> ]
    COMPONENTS
        f1:  Functional_Style!Function[ -> ]
        f2:  Functional_Style!Function[ -> ]
        m:   Shared_Memory_Style!Variable[t]
    CONFIGURATION
        a1:  Shared_Memory_Style!Writes(f1, m)
        a2:  Shared_Memory_Style!Reads(f2, m)
ABSTRACT TO CONCRETE ASSOCIATIONS:
  op  -->          c  -->  m          a  -->  (a1, a2)
  ip  -->
```

Fig. 3. Simple refinement pattern

and write relations).[2] The new concrete signature for the parser and the analyzer reflects the difference between port-to-port communication and direct shared-memory communication through a variable. As an analogous example, consider an architecture consisting of two procedures that communicate solely by means of procedure calls. If we optimize this architecture so that large objects are no longer transmitted by value, but instead are accessed directly as shared objects, the signatures of the two procedures would change.

The refinement pattern in Figure 3 specifies a way to implement dataflow in terms of the reading and writing of a single variable. The read and write relations in the concrete schema are primitives that cannot be refined. The italic letters denote schema variables that can be instantiated with object names, and the symbol "!" is used to qualify names. The pattern can be proven correct with respect to the four associations shown at the bottom of the pattern.[3]

The abstract schema in the pattern matches the level-1 subarchitecture. However, if the same substitutions are made in the concrete schema, three schema variables are left uninstantiated — namely, m, a_1, and a_2. Of course, any unused names could be substituted. Let us assume that the architect selects mnemonic names that give the following associations.

```
base_ast_oport   -->
base_ast_iport   -->
base_ast_channel --> abstract_syntax_tree
base_ast_flow    --> (write_base_ast, read_base_ast)
```

[2]The shared abstract syntax tree could have been represented as an encapsulated data type. 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.

[3]In a correctness proof, the associations in the pattern are incorporated into a more complex mapping between the first-order theories that represent the abstract and concrete architectures.

Since this instance of the pattern matches the abstract subarchitecture of the compiler and since all instances of the pattern are guaranteed to preserve correctness, we can conclude that the proposed refinement is correct.

In a later section, we define enough patterns to transform the full level-1 compiler architecture into the full level-2 architecture. Additional patterns are defined that can be used to transform the level-2 architecture into a more efficient batch-sequential architecture. The final batch-sequential architecture can be found in the appendix. The completed compiler architecture can be connected to other subsystem architectures, such as the file system architecture, to form a correct composite system.

III. Architectures as Theories

We want to leave open the choice of language for specifying an architecture. Therefore, we will represent architectures as logical theories. We find it convenient to use first-order theories; however, our results do not depend on this choice.

It is useful to distinguish among three related architectural theories:

- An *architectural style* is a theory consisting of a vocabulary of the relevant architectural concepts and well-formedness axioms that determine how they can be used. Also associated with a style are rules for translating textual specifications in the style into their underlying logical representation.
- An *architecture* is a theory consisting of one or more style subtheories and possibly an infinite number of constants that are names of the objects in the particular architecture. The axioms of the theory are the style axioms and possibly additional axioms that relate the constants.
- An *architecture schema* is an architecture containing one or more schema variables. An *instance* of an architecture schema is obtained by substitution of constants for all of its schema variables. An instance of an architecture schema is sometimes called an *instance architecture* or an *instance theory.*

A. Architectural Styles

Consider the dataflow style. Its 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.

$$\begin{aligned}&\text{OutPort: oport} \times \text{function} \rightarrow \text{bool}\\&\text{Supplies: oport} \times \text{val} \rightarrow \text{bool}\\&\text{InPort: iport} \times \text{function} \rightarrow \text{bool}\\&\text{Accepts: iport} \times \text{val} \rightarrow \text{bool}\\&\text{Carries: channel} \times \text{val} \rightarrow \text{bool}\\&\text{Connects: channel} \times \text{oport} \times \text{iport} \rightarrow \text{bool}\end{aligned}$$

These predicates are used to represent a dataflow architecture in ordinary first-order logic. Sorts can be represented as unary predicates but, for simplicity, we omit them in formulas.

An example of a well-formedness axiom is that every function must have at least one port:

$$\forall x \exists y [\mathrm{InPort}(y, x) \vee \mathrm{OutPort}(y, x)]$$

Another requirement is that a channel attached to an output port must be able to carry any value supplied by the port:

$$\begin{aligned}&\forall x \forall y [\exists z\, \mathrm{Connects}(x, y, z)\\ &\qquad \supset \forall v [\mathrm{Supplies}(y, v) \supset \mathrm{Carries}(x, v)]]\end{aligned}$$

B. Translation to Logic

Architectures and refinement patterns are expressed in a readable textual language. To reason about them, they are translated into logic by means of simple "theory generation rules" which are associated with architectural styles. For the dataflow style, if the specification of an architecture contains an instance of function declaration schema

f: `Functional_Style!Function[ -> `op`: `t`]`

the underlying theory contains the same instance of first-order sentences

$$\begin{aligned}&\mathrm{OutPort}(op, f)\\ &\forall v [\mathrm{Supplies}(op, v) \supset t(v)]\end{aligned}$$

Similarly, a function declaration of the form

f: `Functional_Style!Function[`ip`: `t` -> ]`

is translated to axioms

$$\begin{aligned}&\mathrm{InPort}(ip, f)\\ &\forall v [t(v) \supset \mathrm{Accepts}(ip, v)]\end{aligned}$$

Dataflow connector

c: `Dataflow_Style!Dataflow_Channel[`t`]`

translates to

$$\forall v [t(v) \supset \mathrm{Carries}(c, v)]$$

and configuration constraint

a: `Dataflow_Style!Connects(`c, op, ip`)`

to

$$\mathrm{Connects}(c, op, ip)$$

which is not an object and, therefore, is not named in the logic.

C. Architecture Schemas

The two schemas appearing in the pattern of Figure 3 will be referred to throughout the paper. Theory Θ_{D} corresponds to the abstract schema and theory Θ_{M} corresponds to the concrete schema.

Theory Θ_{D} is formed by applying the theory generation rules of the dataflow style to the abstract schema, which gives

$$\begin{aligned}&\mathrm{OutPort}(op, f_1)\\ &\forall v [\mathrm{Supplies}(op, v) \supset t(v)]\\ &\mathrm{In\_Port}(ip, f_2)\\ &\forall v [t(v) \supset \mathrm{Accepts}(ip, v)]\\ &\forall v [t(v) \supset \mathrm{Carries}(c, v)]\\ &\mathrm{Connects}(c, op, ip)\end{aligned}$$

This theory satisfies the two well-formedness axioms stated earlier.

The concrete architecture schema in Figure 3 is written in a shared-memory style, which permits the reading and writing of a shared variable. Shared-variable communication is modeled using a call site as the interface between a function and the shared variable.[4] A call site serves the same purpose as a port in the dataflow style. The name of every different call site must be unique. Θ_{M} 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 Θ_{M} is

$$\begin{aligned}&\text{Holds: variable} \times \text{val} \rightarrow \text{bool}\\ &\text{CallSiteOf: site} \times \text{function} \rightarrow \text{bool}\\ &\text{Writes: wsite} \times \text{variable} \rightarrow \text{bool}\\ &\text{Puts: wsite} \times \text{val} \rightarrow \text{bool}\\ &\text{Reads: rsite} \times \text{variable} \rightarrow \text{bool}\\ &\text{Gets: rsite} \times \text{val} \rightarrow \text{bool}\end{aligned}$$

The axioms of Θ_{M} are

$$\begin{aligned}&\forall v [t(v) \supset \mathrm{Holds}(m, v)]\\ &\mathrm{CallSiteOf}(w, f_1)\\ &\mathrm{Writes}(w, m)\\ &\forall v [\mathrm{Puts}(w, v) \supset t(v)]\\ &\mathrm{CallSiteOf}(r, f_2)\\ &\mathrm{Reads}(r, m)\\ &\forall v [t(v) \supset \mathrm{Gets}(r, v)]\end{aligned}$$

which must satisfy the well-formedness axioms for the shared-memory style. Schema variables r and w denote names of call sites and do not appear in Figure 3.

IV. Mappings

To prove the relative correctness of two architectures, we must specify a mapping between them. An *interpretation mapping* is an association between formulas of the language of the abstract theory and formulas of the language of the concrete theory. An interpretation mapping is determined using two different mappings.

- A *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 an abstract-level style can be implemented in terms of the constructs of a concrete-level style. More specifically,

[4]We could have chosen not to model call sites or some equivalent interface object, but this would require a more liberal definition of interpretation than the one given in this paper. The present model simplifies the mapping from Θ_{D} to Θ_{M}.

it maps uninstantiated predicates of the abstract-level language to uninstantiated formulas of the concrete-level language.

Style mappings can be complicated, but need to be defined and proved only once. Name mappings are much simpler and are specific to a given pair of architectures.

A name mapping is determined by the identifier associations in a given refinement pattern. For example, association c-->m in Figure 3 says that channel c of the abstract schema is mapped to variable m of the concrete schema. Association op--> says that the concrete object that corresponds to abstract port op is not explicitly named in the concrete schema. Since we have chosen a shared-memory model that has call sites corresponding to ports, we are free to introduce any unused name for the sites.

Let $N_{\mathrm{M}}^{\mathrm{D}}$ be name mapping

$$\begin{array}{rcl} c & \mapsto & m \\ op & \mapsto & w \\ ip & \mapsto & r \end{array}$$

which relates objects in Θ_{D} to their refinements in Θ_{M}. Observe that not every association in the refinement pattern appears in the name mapping. Identifiers a, a_1, and a_2 refer to part of the specification but do not name objects. Hence, they do not appear in the logical representation. The domain of a name mapping can be extended to include all abstract-level terms by mapping variables to themselves.[5]

Let $S_{\mathrm{M}}^{\mathrm{D}}$ denote the general mapping from the dataflow style to the shared-memory style:

$$\begin{array}{l}
\mathrm{Function}(___1) \mapsto \mathrm{Function}(___1) \\
\mathrm{OutPort}(___1, ___2) \\
\quad \mapsto \mathrm{CallSiteOf}(___1, ___2) \wedge \exists \mathrm{v}\ \mathrm{Puts}(___1, \mathrm{v}) \\
\mathrm{Supplies}(___1, ___2) \mapsto \mathrm{Puts}(___1, ___2) \\
\mathrm{InPort}(___1, ___2) \\
\quad \mapsto \mathrm{CallSiteOf}(___1, ___2) \wedge \exists \mathrm{v}\ \mathrm{Gets}(___1, \mathrm{v}) \\
\mathrm{Accepts}(___1, ___2) \mapsto \mathrm{Gets}(___1, ___2) \\
\mathrm{Channel}(___1) \mapsto \mathrm{Variable}(___1) \\
\mathrm{Carries}(___1, ___2) \mapsto \mathrm{Holds}(___1, ___2) \\
\mathrm{Connects}(___1, ___2, ___3) \\
\quad \mapsto \mathrm{Writes}(___2, ___1) \wedge \mathrm{Reads}(___3, ___1)
\end{array}$$

The Puts and Gets predicates ensure that the right kind of site is associated with each port.

The last association specifies the implementation strategy. In Θ_{D} we have Connects(c, op, ip), which can be implemented by having the call that corresponds to op perform a write operation on the variable that corresponds to channel c, and the one that corresponds to ip read the variable that corresponds to c. 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.

An *interpretation mapping* I is determined from a name mapping N and a style mapping 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,

$$\begin{array}{rcl}
I(P(t_1, t_2, \ldots, t_n)) & = & S(P)(N(t_1), N(t_2), \ldots, N(t_n)) \\
I(\neg F) & = & \neg(I(F)) \\
I(F \wedge G) & = & I(F) \wedge I(G) \\
I(F \vee G) & = & I(F) \vee I(G) \\
I(F \supset G) & = & I(F) \supset I(G) \\
I(\forall x F) & = & \forall x\, I(F)\ ^{6} \\
I(\exists x F) & = & \exists x\, I(F)
\end{array}$$

Let $I_{\mathrm{M}}^{\mathrm{D}}$ denote the interpretation mapping from theory Θ_{D} to theory Θ_{M}. Both the basic facts and the general well-formedness axioms in Θ_{D} must be mapped. For example,

$$\begin{array}{l}
I_{\mathrm{M}}^{\mathrm{D}}(\mathrm{Connects}(c,\ op,\ ip)) \\
\quad = S_{\mathrm{M}}^{\mathrm{D}}(\mathrm{Connects})(N_{\mathrm{M}}^{\mathrm{D}}(c), N_{\mathrm{M}}^{\mathrm{D}}(op), N_{\mathrm{M}}^{\mathrm{D}}(ip)) \\
\quad = S_{\mathrm{M}}^{\mathrm{D}}(\mathrm{Connects})(m,\ w,\ r) \\
\quad = \mathrm{Writes}(w, m) \wedge \mathrm{Reads}(r, m)
\end{array}$$

which is the intended implementation. Similarly, the general dataflow-style requirement that each function have at least one input or output port maps to the shared-memory requirement that each function have a call site that can input or output values. That is,

$$\begin{array}{l}
I_{\mathrm{M}}^{\mathrm{D}}(\forall \mathrm{x} \exists \mathrm{y}[\mathrm{InPort}(\mathrm{y}, \mathrm{x}) \vee \mathrm{OutPort}(\mathrm{y}, \mathrm{x})]) \\
\quad = \forall \mathrm{x} \exists \mathrm{y}[I_{\mathrm{M}}^{\mathrm{D}}(\mathrm{InPort}(\mathrm{y}, \mathrm{x})) \vee I_{\mathrm{M}}^{\mathrm{D}}(\mathrm{OutPort}(\mathrm{y}, \mathrm{x}))] \\
\quad = \forall \mathrm{x} \exists \mathrm{y}[S_{\mathrm{M}}^{\mathrm{D}}(\mathrm{InPort})(N_{\mathrm{M}}^{\mathrm{D}}(\mathrm{y}), N_{\mathrm{M}}^{\mathrm{D}}(\mathrm{x})) \\
\qquad\qquad \vee S_{\mathrm{M}}^{\mathrm{D}}(\mathrm{OutPort})(N_{\mathrm{M}}^{\mathrm{D}}(\mathrm{y}), N_{\mathrm{M}}^{\mathrm{D}}(\mathrm{x}))] \\
\quad = \forall \mathrm{x} \exists \mathrm{y}[(\mathrm{CallSiteOf}(\mathrm{y}, \mathrm{x}) \wedge \exists \mathrm{v}\ \mathrm{Gets}(\mathrm{y}, \mathrm{v})) \\
\qquad\qquad \vee (\mathrm{CallSiteOf}(\mathrm{y}, \mathrm{x}) \wedge \exists \mathrm{v}\ \mathrm{Puts}(\mathrm{y}, \mathrm{v}))]
\end{array}$$

V. Correctness

Two instance architectures, represented as theories, are proven correct with respect to an interpretation mapping between them and the completeness assumption. An interpretation mapping contains a style mapping whose semantic correctness should be established as a proof obligation. Proof of style mappings is discussed in a companion paper [18], which gives a proof of mapping $S_{\mathrm{M}}^{\mathrm{D}}$ from the dataflow to the shared-memory style. The connectors in the styles are defined in a temporal logic, and both safety and fairness conditions are shown to be satisfied by the shared-memory implementation. The safety condition is that the shared-memory implementation preserves order and does not lose values; the fairness condition is that all values written into shared memory will eventually be read. The proof of a style mapping is performed only once; it need not be repeated when the two styles are used.

A. Criterion

Let Θ and Θ' be instance theories (containing no schema variables) associated with an abstract and a concrete architecture, respectively. Let I be an interpretation mapping

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

[6]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.

from the language of Θ to the language of Θ'. For every sentence F, mapping I is a *theory interpretation* provided

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

This is the usual definition of correctness.

Since a given architecture is assumed to be complete with respect to its level of detail, we additionally require that the concrete architecture add no new facts about the abstract architecture. To prove this, we must additionally show that

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

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

B. Proof Technique

Again, let Θ and Θ' be instance theories and I be the interpretation mapping between them. We present a general model-theoretic proof technique for showing that interpretation mapping I is a faithful interpretation of abstract theory Θ in concrete theory Θ'. First, we prove that I is a theory interpretation of Θ in Θ'. This can be done by means of a standard proof technique: *For every axiom in Θ, establish that the image of the axiom under I is a logical consequence of the axioms of Θ'.*

Second, we must prove that interpretation mapping I is a faithful. The proof method has to take into account that there is no direct method for determining that a formula is not in Θ'. Our proof technique for faithfulness is based on two model-theoretic concepts:

- The interpretation mapping I from Θ to Θ' induces a mapping I' from structures of the concrete language to structures of the abstract language.[7] Given a structure $\mathcal{A}'$ of the concrete language, I' maps $\mathcal{A}'$ to a structure $\mathcal{A}$ of the abstract language as follows. The universe of $\mathcal{A}$ is the same as the universe of $\mathcal{A}'$. If I maps atomic formula $P(x_1, x_2, \ldots, x_n)$ to concrete formula F, then I' assigns to predicate P in the abstract language the set of tuples in $\mathcal{A}'$ that satisfy F.
- The theory that describes structure $\mathcal{A}$ is obtained as follows. First, expand the language of $\mathcal{A}$ to include a name for every member of the universe of $\mathcal{A}$. Next, expand $\mathcal{A}$ by assigning every new name to the appropriate member of $\mathcal{A}$. The theory that describes $\mathcal{A}$ is the set of sentences in the expanded language that are true in the expanded structure.

Our technique for proving the faithfulness of I can now be stated as follows: *For every model $\mathcal{A}$ of Θ, find a model $\mathcal{A}'$ of Θ' such that the image of $\mathcal{A}'$ under the induced mapping I' can be expanded to a model of the theory that describes $\mathcal{A}$.* This model-theoretic characterization of faithfulness is equivalent to our theory-based definition of correctness.

[7] Recall from logic that a structure of a first-order language consists of a universe and the assignment of elements of the universe to the constants and relations over the universe to the function and predicate symbols.

Roughly speaking, this characterization requires that, for every model $\mathcal{A}$ of Θ, there is a model $\mathcal{A}'$ of Θ' such that $\mathcal{A}$ and $I'(\mathcal{A}')$ cannot be distinguished using the resources of first-order logic. If we were to use an architectural specification language based on some other logic, a similar characterization based on the expressive power of that logic would be substituted. For example, if the content of our architectural specifications were expressed in type theory, we would require that $I'(\mathcal{A}')$ can be expanded to model every type-theoretic sentence expressible in the language that contains a name for every object in the domain of $\mathcal{A}$, every relation among those objects, every relation among those relations, and so on, that is true in $\mathcal{A}$. (It is easy to see that this amounts to requiring that $I'(\mathcal{A}')$ and $\mathcal{A}$ be isomorphic.) So our general method for demonstrating faithfulness can be used with any logic-based architectural specification language, as long as the question of whether a structure that represents an architecture satisfies a specification has a well-defined answer.

C. Application to Refinement Patterns

A refinement pattern consists of a triple $\langle \Theta, \Theta', N \rangle$ where Θ and Θ' are theories containing schema variables and N is a name mapping from Θ to Θ'. A pattern is correct provided every instance of Θ and Θ' is relatively correct with respect to the same instance of interpretation mapping $I : \Theta \rightarrow \Theta'$ determined by mapping N and the relevant style mapping(s).

Consider theories Θ_{D} and Θ_{M} related by interpretation mapping $I^{\mathrm{D}}_{\mathrm{M}}$. We must show that, for every instantiation of the schema variables, $I^{\mathrm{D}}_{\mathrm{M}}$ is a theory interpretation of Θ_{D} in Θ_{M} and $I^{\mathrm{D}}_{\mathrm{M}}$ is faithful. The former is straightforward.

To prove faithfulness, consider the induced mapping of $I^{\mathrm{D}}_{\mathrm{M}}$. If $\mathcal{M}$ is a structure for Θ', then the induced mapping applied to $\mathcal{M}$ is a structure $\mathcal{D}$ for the dataflow language. The only interesting assignment is to the predicate Connects, which is the set of tuples

$$\{\langle x, y, z \rangle \in |\mathcal{M}|^3 \colon \mathcal{M} \models \mathrm{Writes}(y, x) \wedge \mathrm{Reads}(z, x)\}$$

because $I^{\mathrm{D}}_{\mathrm{M}}$ maps $\mathrm{Connects}(c, op, ip)$ to the formula

$$\mathrm{Writes}(w, m) \wedge \mathrm{Reads}(r, m)$$

where c, op, ip, w, m, and r are schema variables.

To show that $I^{\mathrm{D}}_{\mathrm{M}}$ is faithful, we use $I^{\mathrm{D}}_{\mathrm{M}}$ to transform a model $\mathcal{D}$ of an instance of Θ_{D} to a model $\mathcal{M}$ of an instance of Θ_{M}. The universe of $\mathcal{M}$ is the same as $\mathcal{D}$ in this example. The predicate Function is assigned to the set of all objects that are functions in $\mathcal{D}$, namely,

$$\{x \in |\mathcal{D}| \colon \mathcal{D} \models \mathrm{Function}(x)\}$$

so that $\mathcal{D}$ and $\mathcal{M}$ agree on functions. The predicate Variable is assigned to

$$\{x \in |\mathcal{D}| \colon \mathcal{D} \models \mathrm{Channel}(x)\},$$

the predicate Reads is assigned to

$$\{\langle x, y \rangle \in |\mathcal{D}|^2 \colon \text{for some } z \text{ in } |\mathcal{D}|,\ \mathcal{D} \models \mathrm{Connects}(y, z, x)\}$$

and similarly for the remaining predicates. The image of $\mathcal{M}$ under the induced mapping is $\mathcal{D}$. Obviously, $\mathcal{D}$ can be expanded to a model of the theory that describes $\mathcal{D}$. Therefore, $I_{\mathrm{M}}^{\mathrm{D}}$ is faithful. Note that, since the image of $\mathcal{M}$ under the induced mapping is identical to $\mathcal{D}$, the interpretation $I_{\mathrm{M}}^{\mathrm{D}}$ would remain faithful if we were to switch from first-order logic to some stronger logic, such as type theory.

VI. Composition

We define two forms of composition for instance architectures. Horizontal composition is used to compose instances of refinement patterns to form one large composite refinement architecture. It is also used to compose existing architectures into larger architectures. Vertical composition is used to chain together a sequence of correct architectures, allowing us to conclude that the most concrete architecture in a hierarchy is correct with respect to the most abstract architecture in the hierarchy. Vertical composition is justified since faithful interpretation is transitive.

Let Θ_1 and Θ_2 be instance 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

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

are faithful interpretations, then we want

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

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

$$\text{if} \quad F \in \Theta_1 \cap \Theta_2 \quad \text{then} \quad I_1(F) = I_2(F)$$

   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

$$\Theta_1' \cup \Theta_2' \ \vdash\ I_1 \cup I_2(F)$$

   then we must prove that

$$I_1[\Theta_1] \cup I_2[\Theta_2] \ \vdash\ I_1 \cup I_2(F).$$

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).

Of course, we do not want to have to prove that every refinement pattern can be composed with every other refinement pattern. Instead, we would like simple syntactic criterionthat, if satisfied, guarantees compositionality. One such criterion is that the two abstract architectures can share only components and lower-level architectures can share only images of those components under the interpretation mapping. This means that an architecture cannot contain certain global assertions, such as a requirement that there are exactly three connections in any architecture.

An example of the horizontal composition of pattern instances involves the compiler architecture in Figure 1. We have proved that the dataflow connection between the parser and the analyzer is implemented correctly by means of the reading and writing of the tree, using instances of Θ_{D}, Θ_{M}, and $I_{\mathrm{M}}^{\mathrm{D}}$ from Figure 3. Similarly, we can show that the dataflow connection from the lexical analyzer to the parser is correctly implemented by the pipeline connection. The two architectures share only one component, the parser. Therefore, our second condition is satisfied and we can compose them without further proof.

A different kind of example is contained in Figure 4. 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. According to our syntactic constraint, the three architectures can be combined to form a composite system that is correct if the three subsystems are.

VII. Some Refinement Patterns

We present six broadly useful patterns for refining components, connectors, and interfaces.[8] The patterns involve several common architecture styles and each pattern has been proven correct.

A refinement pattern is presented in a table containing two architecture schemas, an association of abstract and concrete objects, and possibly constraints on one or both of the schemas. By convention, a schema variable that occurs in both an abstract and a concrete schema must match the same object, modulo renaming. We prime a concrete schema variable to indicate that it is the name of a new object not associated with any abstract-level object, or that it

[8]Type refinement is not covered because it requires a somewhat different correctness criterion.

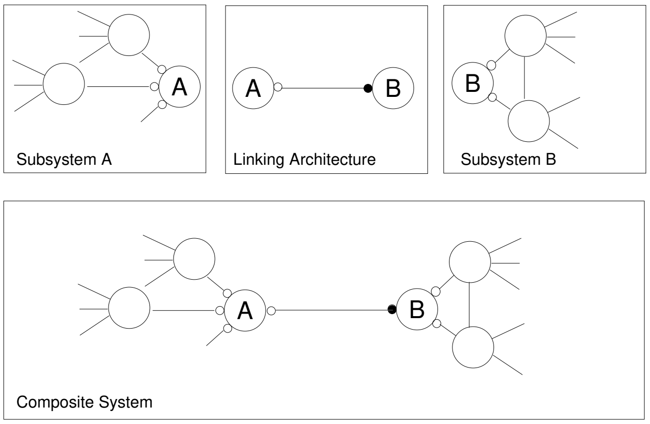

Fig. 4. Illustration of Subsystem Composition

Symbol	Style Name
BS	Batch_Sequential_Style
CT	Control_Transfer_Style
D	Dataflow_Style
F	Functional_Style
PP	Process_Pipeline_Style
SM	Shared_Memory_Style

TABLE I

ABBREVIATIONS FOR STYLE NAMES IN REFINEMENT PATTERNS

PATTERN OF ABSTRACT ARCHITECTURE:

M: MODULE[p_1 -> p_2]
  COMPONENTS
    f: F!Function[p_{11} -> p_{12}]

PATTERN OF CONCRETE ARCHITECTURE:

M: MODULE[p_1 -> p_2]
  COMPONENTS
    f': MODULE[p_{11} -> p_{12}]

ABSTRACT TO CONCRETE ASSOCIATIONS:

f --> f'

Fig. 5. Decomposing a component into subcomponents (Pattern 1)

denotes a required change to the associated abstract-level object. The intended meaning is obvious from context. A reference to a style in a refinement pattern is abbreviated according to the naming conventions summarized in Table I.

We assume that connections in an architecture do not share interface points. Multiple uses of a given interface point are modeled with multiple copies of the same point. This model has the advantage that interfaces and connections can be refined more flexibly. However, this choice of representation can result in an increase in the number of interface points.

A. Component Refinement

Figure 5 contains a refinement pattern for decomposing a functional component into a collection of components wrapped by a module. Component f is refined into module f', hence the association f-->f'. A module signature contains all externally visible interfaces within the module. Since each interface point is an object with a unique name, there is no confusion as to the correspondences between the interface points of f and those of components in f'. By requiring that f and f' have the same signature, we are guaranteed that the original connections involving f are maintained through its subcomponents. The refinement is faithful because the interface requirement on f and f' prevents the addition or deletion of connections.

The next two patterns are for aggregating variables in situations that are common in intermediate stages of a development. This is done for time and space efficiency, especially if the variables hold large objects. Application of the patterns also results in a simpler design.

Figure 6 contains a pattern for merging shared variables when one of them is a private variable. This pattern merges a shared variable m_1, which is written by component f_1 and read by component f_2, with a private variable m_2, which is read and written by component f_1. This is expressed by the association (m_1, m_2)-->m'. There are three basic requirements on this form of refinement:

- The variables denoted by schema variables m_1 and m_2 must have the same type, denoted by schema variable t.
- Only the component denoted by f_1 can write the variable denoted by m_1. This prevents a new flow to f_1, which would violate the faithfulness requirement.
- Only f_1 accesses private variable m_2, otherwise a new flow would be created by the refinement. This requirement is enforced by the constraint on the abstract architecture.

A variant of this pattern combines the shared variable and the private variable into two fields of a record structure. With this variant, the constraint on the abstract architecture is not needed, provided that the components involved access only the proper fields of the record. This kind of refinement would not increase efficiency, but could help simplify the design.

Figure 7 contains a pattern for merging shared variables when neither of them are private. The two shared variables are connected by a common functional component. A shared variable denoted by schema variable m_1 is written by functional component f_1 and read by f_2. Shared variable m_2 is written by f_2 and read by f_3. The merge is expressed by the association (m_1, m_2)-->m'.

Our correctness criterion places the following restrictions on the architectures:

- The variables to be merged must be of the same type t.
- Since we treat dataflow as an intransitive relation, we also treat other relations dealing with the flow of data as intransitive relations. Therefore, functional components f_1, f_2, and f_3 have to be executed sequentially in batch mode so that we cannot infer the existence of a new abstract flow from f_1 to f_3. This is prevented by configuration assertions a'_5 and a'_6.
- No other functional components can read m_1 or write

PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1 : F!Function[p_{11} -> p_{12}]
f_2 : F!Function[p_{21} -> p_{22}]
m_1: SM!Variable[t]
m_2: SM!Variable[t]
CONFIGURATION
a_1: SM!Writes(f_1, m_1)
a_2: SM!Reads(f_2, m_1)
a_3: SM!Writes(f_1, m_2)
a_4: SM!Reads(f_1, m_2)

PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1 : F!Function[p_{11} -> p_{12}]
f_2 : F!Function[p_{21} -> p_{22}]
m': SM!Variable[t]
CONFIGURATION
a'_1: SM!Writes(f_1, m')
a'_2: SM!Reads(f_2, m')
a'_3: SM!Reads(f_1, m')

ABSTRACT TO CONCRETE ASSOCIATIONS:

(m_1, m_2)	-->	m'	(a_1, a_3)	-->	a'_1
a_2	-->	a'_2	a_4	-->	a'_3

CONSTRAINTS ON ABSTRACT ARCHITECTURE:
$\neg(\exists f$: F!Function)
[$f \neq f_1$
$\wedge$ [SM!Writes(f, m_1)
$\vee$ SM!Writes(f, m_2)
$\vee$ SM!Reads(f, m_2)]]

Fig. 6. Merging a shared variable with a private variable (Pattern 2)

m_2, which is enforced by a constraint on the abstract architecture.

A variant of this pattern combines the shared variables into two fields of a record structure. With this variant, the sequential ordering assertions in the concrete architecture and the constraint on the abstract architecture are not needed.

B. Connector Refinement

Figure 8 contains a pattern for implementing a dataflow connector by a pipe. Dataflow channel c from f_1 to f_2 is refined into a pipe c' connecting f_1 to f_2. The connector refinement is expressed by the associations c-->c' and a-->a'. This refinement is obviously faithful. Semantically, it can be justified on the basis of the meaning of the dataflow and pipe connectors.

Figure 9 contains a pattern for refining two functional components f_1 and f_2 that are executed in batch-sequential mode into a module with a main functional component f' transferring control first to f_1 and then to f_2. The correctness of refinements of this form depends on the following properties.

- Component f_1 has to complete before f_2 can start, which is enforced by configuration assertion a'.
- Concrete component f' cannot transfer control to f_2 until f_1 completes, and f_1 cannot transfer control to f' after f_2 starts. These ordering relationships are

PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1 : F!Function[p_{11} -> p_{12}]
f_2 : F!Function[p_{21} -> p_{22}]
f_3 : F!Function[p_{31} -> p_{32}]
m_1: SM!Variable[t]
m_2: SM!Variable[t]
CONFIGURATION
a_1: SM!Writes(f_1, m_1)
a_2: SM!Reads(f_2, m_1)
a_3: SM!Writes(f_2, m_2)
a_4: SM!Reads(f_3, m_2)

PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1 : F!Function[p_{11} -> p_{12}]
f_2 : F!Function[p_{21} -> p_{22}]
f_3 : F!Function[p_{31} -> p_{32}]
m': SM!Variable[t]
CONFIGURATION
a'_1: SM!Writes(f_1, m')
a'_2: SM!Reads(f_2, m')
a'_3: SM!Writes(f_2, m')
a'_4: SM!Reads(f_3, m')
a'_5: BS!Starts_After_Finish_Of(f_2, f_1)
a'_6: BS!Starts_After_Finish_Of(f_3, f_2)

ABSTRACT TO CONCRETE ASSOCIATIONS:

(m_1, m_2)	-->	m'	a_1	-->	a'_1
a_2	-->	a'_2	a_3	-->	a'_3
a_4	-->	a'_4			

CONSTRAINTS ON ABSTRACT ARCHITECTURE:
$\neg(\exists f$: F!Function)
[$f \neq f_2$
$\wedge$ [SM!Reads(f, m_1)
$\vee$ SM!Writes(f, m_2)]]

Fig. 7. Merging shared variables (Pattern 3)

PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1: F!Function[p_{11} -> op:t, p_{12}]
f_2: F!Function[ip:t, p_{21} -> p_{22}]
CONNECTORS
c: D!Dataflow_Channel[t]
CONFIGURATION
a: D!Connects(c, op, ip)

PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[p_1 -> p_2]
COMPONENTS
f_1: F!Function[p_{11} -> op:t, p_{12}]
f_2: F!Function[ip:t, p_{21} -> p_{22}]
CONNECTORS
c': PP!Pipe[t]
CONFIGURATION
a': PP!Connects(c', op, ip)

ABSTRACT TO CONCRETE ASSOCIATIONS:

c	-->	c'	a	-->	a'

Fig. 8. Implementing a dataflow connector by a pipe (Pattern 4)

```
PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[p1 -> p2]
    COMPONENTS
        f1:  F!Function[p11 -> p12]
        f2:  F!Function[p21 -> p22]
    CONFIGURATION
        a:  BS!Starts_After_Finish_Of(f1, f2)
PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[p1 -> p2]
    COMPONENTS
        f':  F!Function[ -> ]
        f1:  F!Function[p11 -> p12]
        f2:  F!Function[p21 -> p22]
    CONNECTORS
        s'1:  CT!Enabling_Signal
        s'2:  CT!Enabling_Signal
    CONFIGURATION
        a'11:  CT!Sender(s'1, f1)
        a'12:  CT!Receiver_Signal(s'1, f')
        a'21:  CT!Sender(s'2, f')
        a'22:  CT!Receiver_Signal(s'2, f2)
        a'  :  CT!Before(s'1, s'2)
ABSTRACT TO CONCRETE ASSOCIATIONS:
 a  -->  a'
CONSTRAINTS ON CONCRETE ARCHITECTURE:
    ¬(∃s':  CT!Enabling_Signal)
        [CT!Sender(s', f')
            ∧ CT!Receiver_Signal(s', f2)
            ∧ CT!Before(s', s'1)]
    ¬(∃s':  CT!Enabling_Signal)
        [CT!Sender(s', f1)
            ∧ CT!Receiver_Signal(s', f')
            ∧ CT!Before(s'2, s')]
```

Fig. 9. Implementing ordering constraint using explicit control transfer (Pattern 5)

enforced by the two constraints on the concrete architecture.

- All functional components have to be enabled by f' and every control transfer must be between f' and a functional component. This is enforced by a well-formedness constraint in the control-transfer style, not by a constraint in the pattern.

C. Interface Refinement

Figure 10 contains the full specification of the pattern introduced earlier in Figure 3. The refinement of the dataflow connection into a shared-memory implementation has the side effect of changing the signature of the two functions, since connections do not share interface points.

VIII. Example Revisited

We now apply the refinement patterns to the compiler architectures illustrated earlier in Figure 1. In particular, we show how the level-1 compiler architecture can be refined into the level-2 compiler architecture using five of the patterns. The textual specification of the architectures are simplified through the use of ellipses for parts of the specification that are not relevant to the refinement under consideration. The full textual specifications for levels 1 and 2 are in Figure 2 and the appendix, respectively.

```
PATTERN OF ABSTRACT ARCHITECTURE:
M: MODULE[ip:t, p1 -> op:t, p2]
    COMPONENTS
        f1:  F!Function[p11 -> op:t, p12]
        f2:  F!Function[ip:t, p21 -> p22]
    CONNECTORS
        c:  D!Dataflow_Channel[t]
    CONFIGURATION
        a:  D!Connects(c, op, ip)
PATTERN OF CONCRETE ARCHITECTURE:
M: MODULE[p1 -> p2]
    COMPONENTS
        f1 :  F!Function[p11 -> p12]
        f2 :  F!Function[p21 -> p22]
        m' :  SM!Variable[t]
    CONFIGURATION
        a'1:  SM!Writes(f1, m')
        a'2:  SM!Reads(f2, m')
ABSTRACT TO CONCRETE ASSOCIATIONS:
 c        -->  m'          a  -->  (a'1, a'2)
 (op, ip) -->
```

Fig. 10. Implementing dataflow with a shared variable (Pattern 6)

The development of the level-2 architecture involves three main steps — the introduction of the pipe between the lexical analyzer and the parser, the development of the shared tree accessed by the parser, analyzer/optimizer, and code generator, and the development of the shared symbol table between the lexical analyzer and the optimizer. All patterns, with the exception of Pattern 5, are used. (Pattern 5 is applied repeatedly to the level-2 compiler architecture to get the level-3 architecture in the appendix.)

A. Introduction of the Pipe

This refinement is a straightforward application of Pattern 4. Consider the following abbreviated subarchitecture of the level-1 compiler.

```
compiler_L1: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    lexical_analyzer: Function
      [... -> token_oport: SEQ(token), ...]
    parser: Function[token_iport: SEQ(token) -> ...]
  CONNECTORS
    token_channel: Dataflow_Channel[SEQ(token)]
  CONFIGURATION
    token_flow:
      Connects(token_channel, token_oport, token_iport)
```

Pattern 4 can be used to refine dataflow channel `token_channel` into pipe `token_pipe`, resulting in the following level-2 architecture.[9]

```
compiler_L2: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    lexical_analyzer_module: MODULE
      [... -> token_oport: Finite_Stream(token)]
    parser: Function
      [token_iport: Finite_Stream(token) -> ]
```

[9] An output and an input port of type `SEQ(token)` were implemented as type `Finite_Stream(token)`. A stream is a function from clock times to values. The correctness of this type refinement is not treated in this paper.

```
  CONNECTORS
    token_pipe: Pipe[Finite_Stream(token)]
  CONFIGURATION
    token_flow:
      Connects(token_pipe, token_oport, token_iport)
```

B. Development of the Shared Abstract Syntax Tree

Consider the following dataflow architecture.

```
compiler_L1: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    parser: Function [... -> base_ast_oport: ast]
    analyzer_optimizer: Function
      [base_ast_iport: ast, ... -> full_ast_oport: ast]
    code_generator: Function
      [full_ast_iport: ast -> ...]
  CONNECTORS
    base_ast_channel: Dataflow_Channel[ast]
    full_ast_channel: Dataflow_Channel[ast]
  CONFIGURATION
    base_ast_flow:
      Connects(base_ast_channel,
                 base_ast_oport, base_ast_iport)
    full_ast_flow:
      Connects(full_ast_channel,
                 full_ast_oport, full_ast_iport)
```

It can be split into two dataflow architectures and Pattern 6 is applied to each to construct two shared memory architectures, which are composed horizontally to form a single architecture. Then, Pattern 3 can be applied to merge the two shared data structures into a single shared tree, called `abstract_syntax_tree`. The three architectures compose vertically, so we know that the final architecture, given below, is correct with respect to the original dataflow architecture.

```
compiler_L2: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    parser:               Function[... -> ]
    analyzer_optimizer:   Function[ -> ]
    code_generator:       Function[ -> ...]
    abstract_syntax_tree: Variable[ast]
  CONFIGURATION
    write_base_ast:
      Writes(parser, abstract_syntax_tree)
    read_base_ast:
      Reads(analyzer_optimizer, abstract_syntax_tree)
    write_full_ast:
      Writes(analyzer_optimizer, abstract_syntax_tree)
    read_full_ast:
      Reads(code_generator, abstract_syntax_tree)
    precedence_1:
      Starts_After_Finish_Of(analyzer_optimizer, parser)
    precedence_2:
      Starts_After_Finish_Of(code_generator,
                             analyzer_optimizer)
```

C. Development of the Shared Symbol Table

This refinment involves three individual refinements, but only vertical composition. Consider the following architecture, which specifies the dataflow from the lexical analyzer to the analyzer/optimizer that is used to transmit binding information.

```
compiler_L1: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    lexical_analyzer: Function
      [char_iport: SEQ(character)
         -> bind_oport: SEQ(binding), ...]
    analyzer_optimizer: Function
      [..., bind_iport: SEQ(binding) -> ...]
  CONNECTORS
    bind_channel: Dataflow_Channel[SEQ(binding)]
  CONFIGURATION
    bind_flow:
      Connects(bind_channel, bind_oport, bind_iport)
```

The three refinement steps are:

1. Pattern 1 is used to refine the lexical analyzer into a new module containing itself and a private symbol table used to store bindings locally before proceeding to the next phases of compilation, which could modify the table.
2. Pattern 6 is used to introduce a shared variable between the lexical analyzer and the optimizer, corresponding to `bind_channel`, that can be used to transmit the completed symbol table. [10]
3. Pattern 2 is used to merge the private symbol table and the shared variable into a single shared repository. This reflects a conscious decision to allow no component other than the lexical analyzer to write the table. As a consequence, any additional information, such as storage requirements, and code restructuring must be represented in the abstract syntax tree.

The resulting architecture is given below.

```
compiler_L2: MODULE
      [char_iport: SEQ(character) -> code_oport: code]
  COMPONENTS
    lexical_analyzer_module: MODULE[... -> ...]
      COMPONENTS
        lexical_analyzer: Function[... -> ...]
        symbol_table: Variable[SEQ(binding)]
      CONFIGURATION
        write_bind:
          Writes(lexical_analyzer, symbol_table)
        read_bind:
          Reads(lexical_analyzer, symbol_table)
      END lexical_analyzer_module
    analyzer_optimizer: Function[ -> ]
  CONFIGURATION
    read_bind:
      Reads(analyzer_optimizer,
              lexical_analyzer_module!symbol_table)
```

D. Putting The Pieces Together

The three individual architecture hierarchies can be flattened to two levels because faithful interpretations are transitive. Then, they can be composed horizontally to form the composite compiler architectures at levels 1 and 2. The level-3 compiler architecture can be formed in a similar fashion.

It is worth noting that a series of refinements can result in a deep hierarchy that need not be saved explicitly. The sequence of steps in deriving a concrete architecture are important, but the intermediate architectures themselves may not be. We saw this in the development of the symbol table.

[10] The nested `lexical_analyzer_module` can be flattened by a restructuring pattern so that patterns can be applied directly.

We also observe that it is possible to adopt a hybrid approach to architecture development in which parts of the architecture are developed by means of refinements and other parts are specified completely by hand. In the latter situation, refinement patterns can be used to validate the correctness of the putative implementation architectures through a straightforward matching procedure. Correct hierarchies can be composed no matter how they were developed, provided the composition is faithful.

IX. Application to a Power-Control System

The approach presented in this paper has been used to design an architecture for an operational power control system implemented in 200,000 lines of FORTRAN 77 code . The system is used by Tokyo Electric Power Company, Inc. to achieve efficient administration of power-supply systems in Tokyo, Japan. The power-control system was developed by Meidensha Corporation and its architecture is considered a company asset. Originally, the details of the architecture were represented informally in several loosely connected documents. This created a difficult situation for Meidensha Corp. because they wanted to expand their business in control systems to other areas with similar requirements, which would require minor modifications to the reference architecture. With no formalized architecture, such an expansion would certainly lead to duplication of effort and unnecessary errors in implementation.

Our objective was to formalize the reference architecture in terms of company styles and at two levels of detail, and to guarantee that the concrete architecture is correct with respect to the abstract architecture. This task was completed successfully. The abstract architecture was stated in terms of a dataflow style, and the concrete architecture was a combination of a call-return style, a (structured) shared-memory style, and a special process synchronization style for DEC VMS operating systems. Twelve patterns were used in the development; each was used many times.

Pattern 1 was used for decomposing functional components into modules; Pattern 6 was used to implement dataflow as a shared variable. Domain-specific refinement patterns were needed to handle two distinctive features of the concrete power-control architecture—heavy use of shared memory and process synchronization by an enabling signal. The shared memory did not have a uniform structure. Dozens of dataflows were implemented by a single record containing one field for each flow. Some dataflows were implemented as a record structure containing the data and a one-bit enabling signal, and others as a message channel plus a signaling channel. A collection of variables containing one bit are packaged into a bitstring for efficient communication. Variants of Patterns 2 and 3 were used to aggregate individual variables into records.

This successful experience strongly suggests that, in the domain of power control, only a small number of patterns is required. This allows the cost of pattern verification to be amortized across many applications in the power-control domain. We know that many of the patterns are relevant in other domains as well, and believe that only a modest number of new patterns will be needed in many application areas.

X. Related Work

The field of architecture-driven software development will not reach its full potential until it is possible to refine and compose architectures incrementally, flexibly, and in ways that preserve the desired properties. Ideally, deep properties of an architecture, such as relative correctness, should be preserved. This requires that an architecture hierarchy be represented formally and the mapping between the levels be precise and explicit. We review related work in the areas of refinement, correctness, and composition.

Previous approaches to specification refinement have concentrated on the preservation of functional properties, which occurs when the mapping between specifications is a theory interpretation. The mapping often is complicated by a change in data representation. This can be taken into account by adapting the technique of Hoare [12] to relate the types in the abstract and concrete specifications. An analogous problem arises in architecture refinement when there is a change in style. We have introduced the notion of a style mapping to related the styles in the abstract and concrete architectures.

We are not the first to recognize the importance of schematic transformations in stepwise refinement. In [10], Gerhart gives several examples of schema transformations that preserve functional correctness. We define schema transformations that preserve architecture correctness. The two forms of refinement are complementary. An architecture refinement hierarchy describes system organization — its components, interfaces, and connections. Functional refinement is used to develop the behavior of the system components in the architecture. In both instances, schemas can be used to increase the reusability of designs and proofs.

Of course, the utility of architecture hierarchies has been recognized for some time. For example, in the 1970s Jackson [13], Yourdan and Constantine [20], DeMarco [7], and others describe system architectures and, more recently, architectural description has been the basis for commercial offerings. However, previous work has given little attention to the mapping between levels of abstraction. We formally defined the interpretation mapping required in architecture correctness proofs in terms of a specific name mapping and a general, reusable style mapping. The mapping also provides the basis for traceability of architectural design decisions, which is useful in practice.

Recently, another form of a mapping between architectures has been developed for the Rapide architecture definition language [14], [15]. Rapide is used to define executable architectures based on distributed event processing. Two architectures are related by mapping concrete events to abstract events. Event mappings provide the basis for comparative simulation, a technique that complements static modeling.

The standard criterion for functional correctness is not applicable to architectures because of the completeness assumption. A similar completeness assumption is made widely in the database community for analogous reasons, see Reiter [19]. However, Reiter allows only finitely many objects, so a "domain closure axiom" can be used to enumerate the domain of discourse. No similar technique can be applied here because, in general, an architecture can be infinite. For example, we allow quantification over infinite types (such as integers) and dynamic architectures with an unbounded number of processes and connections. Because of the completeness assumption, an abstract architecture must be faithfully interpreted in the concrete architecture.

In [17], Moriconi and Hare study the relative correctness of two architectures under the completeness assumption. They make the simplifying assumption that an architecture can contain only a fixed, finite number of objects. Broy [5], Brinksma [4], and others have applied the standard approach to correctness to architectures. Broy's component refinements turn out to be conservative (and, hence, faithful) 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.

We appear to be the first to observe that, in an architectural correctness proof, it is important to establish the semantic correctness of the relevant style mappings. The importance of reasoning about connectors was recognized by Allen and Garlan [3], who formalize them in a subset of CSP [11] and then proved absence of deadlock. In [18] we define the meaning of connectors axiomatically in a temporal logic and prove both fairness and safety properties of an implementation of the dataflow connector in shared memory. Garlan et al [8], [9] also have done important work on identifying and exploiting architectural styles. We build on their work, developing schematic style mappings and schematic refinements involving style-to-style transformations.

Composition has been studied recently by Abadi and Lamport [1], [2]. Their results are semantic and applicable to any domain, whereas ours are syntactic and specialized to the domain of software architecture. It is easy to state general criteria for the correctness of horizontal composition of architectures. However, it requires a difficult proof that it is not possible to infer new facts about the composite abstract architecture from the composite concrete architecture. Therefore, we defined a new specialized form of horizontal composition that requires only very simple syntactic checks. Broy [5] gives three operators for composing functional-style architectures, but does not consider the composition of architectures involving multiple styles. Vertical composition in a hierarchy of architectures is immediate provided each level in the hierarchy is correct with respect to the immediately preceding level.

XI. Conclusion

We have described a stepwise refinement methodology for the development of a heterogeneous hierarchy of architectures that are relatively correct under a particular completeness assumption. We introduced the notion of an architecture refinement pattern as the principal vehicle for codifying reusable solutions to routine architectural design problems. Once an architecture refinement pattern is proved correct, instances of it can be used in a particular development with no further proof. Patterns are compositional and can be proved in isolation. Subsystem architectures are compositional provided they overlap only in certain ways. The methodology was used successfully to explicate the architectural design of an operational power-control system.

To develop a theory of correctness for architecture refinement, we adapted the technique of faithful interpretation that was introduced in an earlier paper for after-the-fact verification of complete architectures [18]. A new proof technique for checking faithfulness was presented. The interpretation mapping between architectures was simplified by decomposing it into an architecture-specific name mapping and a general style-to-style mapping. We are not aware of this distinction being made elsewhere in the literature. It is important because a style mapping and its proof, both of which can be complex, can be reused in validating any pattern involving the two styles. In contrast, a name mapping is simple, specific to a pattern, and cannot be validated independent of the pattern.

An important premise behind our work is that at least the dominant styles of architectural design can be generalized to partially interpreted schema and most architecture refinements for these styles can be generalized to transformations on schema. We believe that a small number of architectural styles are sufficient for a large number of application domains, and that only a modest number of refinement patterns are needed between each pair of styles. This assertion is supported to some degree by the experiences reported in this paper regarding the compiler and power-control architectures.

Some methodological implications of our faithfulness requirement are worth mentioning. First, architectural styles should clearly differentiate among different architectural concepts. Consider a transaction on a distributed database system, which is an atomic operation logically but rarely is a physically atomic operation. If the abstract "transaction" connector is refined into a two-phase commit protocol involving a series of data transmissions, the refinement will not be faithful unless the purpose of the two-phase commit is taken into account in the design of the style. For example, the commit protocol can be modeled in terms of special "control" connectors that are distinct from the connector that models the transfer of data from the database to the designated site. Then, the abstract flow of data will be the same as the concrete flow, even though there is extra preparatory activity in the concrete architecture. Second, architects can, but should not, circumvent the completeness assumption by adding concepts to a concrete architecture that are unrelated to those in the associated abstract architecture. A correctness criterion could be defined that disallows this, but it would be too restrictive for both design and composition. It is the sort of thing that is unlikely to happen by accident. However, the only real safeguard is the careful scrutiny of each refinement pattern.

We have completed an initial implementation of our methodology sufficient to demonstrate its feasibility. The tool accepts as input a collection of refinement patterns, an abstract architecture, and a concrete architecture. The tool matches instances of the patterns on the abstract and concrete architectures with no user intervention. It makes no attempt to generate instances at this time. One correct composition of refinements is found, if it exists, although in general there may be many possible correct compositions. Specific failures are reported if there is not complete coverage. Any constraints on the application of a refinement pattern are checked automatically. This tool was used in the compiler and the power-control application.

Future work involves the development and evaluation of a handbook of architectural refinement patterns. Good designers tend to use well-established architectural styles, including both basic idioms (such as pipe-filter, client-server, and layering) and reference models (such as the ISO OSI 7-layer model [16]). We are now expanding our library to relate more styles as well as to elaborate more configurations involving the styles in the paper. Eventually, we would like to have a large enough library to support "industrial strength" architecture design. For example, we would like to be able to start with an abstract architecture for a large system, in say a dataflow style, refine it into architectures in a dominant commercial style, such as client/server, and then refine that architecture into an implementation-level architecture that specifies the exact forms of communication. In developing a pattern library, we will be concerned with more than correctness. In particular, we want to use architectural refinement patterns to achieve a greater degree of system predictability. For example, it would be useful to have refinement patterns that optimize performance for specific processors or, more generally, for a given computing and network environment.

Our longer-term objective is to develop a practical architecture synthesis tool that is driven by a broadly useful pattern library. The tool will enforce a design discipline similar to the one enforced by commercial hardware synthesis tools. These tools gain much of their power from the use of clearly defined and reusable styles: typically, register-transfer, logic, and gate-level styles. A pattern library of the sort proposed in this paper is expected to enable effective synthesis of software architectures.

Appendix

I. Lower Level Compiler Architectures

The textual specifications for the two implementations of the compiler architecture make extensive use of imported types and styles, which are not defined in this paper. The specifications have a straightforward translation into logic. The following is the full level-2 specification.

```
compiler_L2: MODULE
        [char_iport: SEQ(character) -> code_oport: code]
      IMPORT character, code, token, binding, ast
          FROM compiler_types
      IMPORT Function FROM Functional_Style
      IMPORT Pipe, Finite_Stream, Connects
          FROM Process_Pipeline_Style
      IMPORT Variable, Reads, Writes
          FROM Shared_Memory_Style
      IMPORT Start_After_Finish_Of
          FROM Batch_Sequential_Style
   COMPONENTS
      lexical_analyzer_module: MODULE
              [char_iport: SEQ(character)
                 -> token_oport: Finite_Stream(token)]
         EXPORTING lexical_analyzer, symbol_table
            IMPORT character, token, binding
                FROM compiler_types
            IMPORT Function FROM Functional_Style
            IMPORT Variable, Reads, Writes
                FROM Shared_Memory_Style
         COMPONENTS
            lexical_analyzer: Function
               [char_iport: SEQ(character)
                  -> token_oport: Finite_Stream(token)]
            symbol_table: Variable[SEQ(binding)]
         CONFIGURATION
            write_bind:
              Writes(lexical_analyzer, symbol_table)
            read_bind:
              Reads(lexical_analyzer, symbol_table)
      END lexical_analyzer_module
      parser:
        Function[token_iport: Finite_Stream(token) -> ]
      analyzer_optimizer:   Function[ -> ]
      code_generator:       Function[ -> code_oport: code]
      abstract_syntax_tree: Variable[ast]
    CONNECTORS
      token_pipe: Pipe[Finite_Stream(token)]
    CONFIGURATION
      token_flow:
        Connects(token_pipe, token_oport, token_iport)
      read_bind:
      Reads(analyzer_optimizer,
              lexical_analyzer_module!symbol_table)
      write_base_ast: Writes(parser, abstract_syntax_tree)
      read_base_ast:
        Reads(analyzer_optimizer, abstract_syntax_tree)
      write_full_ast:
        Writes(analyzer_optimizer, abstract_syntax_tree)
      read_full_ast:
        Reads(code_generator, abstract_syntax_tree)
      precedence_1:
        Starts_After_Finish_Of(analyzer_optimizer, parser)
      precedence_2:
        Starts_After_Finish_Of(code_generator,
                                 analyzer_optimizer)
END compiler_L2
```

The level-3 compiler architecture employs a common implementation of the batch-sequential style. In particular, the batch processing in the level-2 compiler is implemented in terms of a main program and subroutines, as illustrated in Figure 11. This implementation is justified by Pattern 5, which was presented in the body of the paper.

The wiring at level 3 is constrained by the temporal-precedence assertions at level 2.

```
precedence_1:
  Starts_After_Finish_Of(analyzer_optimizer, parser)
precedence_2:
  Starts_After_Finish_Of(code_generator,
                           analyzer_optimizer)
```

We have to make sure that the transfer of control satisfies this temporal ordering of the computation. Two ap-

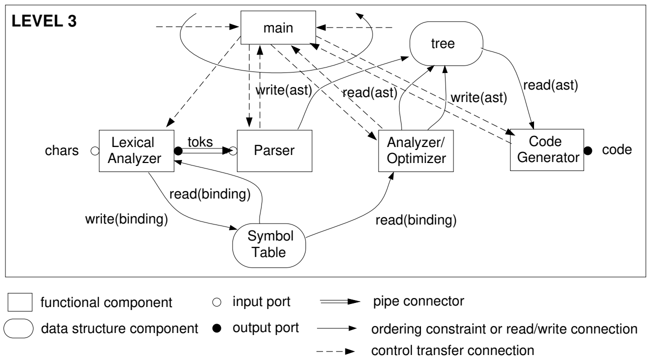

Fig. 11. Third level in architecture hierarchy for compiler

plications of Pattern 5 can be used to guarantee that the ordering relations are satisfied independently. The horizontal composition of the two applications of Pattern 5 guarantees that the composite architecture satisfies both orderings.

The composite level-3 architecture is given below.

```
compiler_L3: MODULE
        [char_iport: SEQ(character) -> code_oport: code]
      IMPORT character, code, token, binding, ast
        FROM compiler_types
      IMPORT Function FROM Functional_Style
      IMPORT Pipe, Finite_Stream, Connects
        FROM Process_Pipeline_Style
      IMPORT Variable, Reads, Writes
        FROM Shared_Memory_Style
      IMPORT Enabling_Signal, Sender, Receiver, Before
        FROM Control_Transfer_Style
   COMPONENTS
      main: Function[ -> ]
      lexical_analyzer_module: MODULE
           [char_iport: SEQ(character)
              -> token_oport: Finite_Stream(token)]
         EXPORTING lexical_analyzer, symbol_table
            IMPORT character, token, binding
              FROM compiler_types
            IMPORT Function FROM Functional_Style
            IMPORT Variable, Reads, Writes
              FROM Shared_Memory_Style
         COMPONENTS
            lexical_analyzer: Function
               [char_iport: SEQ(character)
                  -> token_oport: Finite_Stream(token)]
            symbol_table: Variable[SEQ(binding)]
         CONFIGURATION
            write_bind:
              Writes(lexical_analyzer, symbol_table)
            read_bind:
              Reads(lexical_analyzer, symbol_table)
      END lexical_analyzer_module
      parser:
        Function[token_iport: Finite_Stream(token) -> ]
      analyzer_optimizer:   Function[ -> ]
      code_generator:
        Function[ -> code_oport: code]
      abstract_syntax_tree: Variable[ast]
   CONNECTORS
      token_pipe:     Pipe[Finite_Stream(token)]
      start_main, start_lex, start_parse, parse_finish,
         start_opt, opt_finish, start_gen, gen_finish,
         main_finish: Enabling_Signal
   CONFIGURATION
      token_flow:
        Connects(token_pipe, token_oport, token_iport)
      read_bind:
        Reads(analyzer_optimizer,
                lexical_analyzer_module!symbol_table)
      write_base_ast:
        Writes(parser, abstract_syntax_tree)
      read_base_ast:
        Reads(analyzer_optimizer, abstract_syntax_tree)
      write_full_ast:
        Writes(analyzer_optimizer, abstract_syntax_tree)
      read_full_ast:
        Reads(code_generator, abstract_syntax_tree)
      rcvr_start_main:   Receiver(start_main, main)
      sndr_start_lex:    Sender(start_lex, main)
      rcvr_start_lex:
        Receiver(start_lex,
                   lexical_analyzer_module!lexical_analyzer)
      sndr_start_parse:  Sender(start_parse, main)
      rcvr_start_parse:  Receiver(start_parse, parser)
      sndr_parse_finish: Sender(parse_finish, parser)
      rcvr_parse_finish: Receiver(parse_finish, main)
      sndr_start_opt:    Sender(start_opt, main)
      rcvr_start_opt:
        Receiver(start_opt, analyzer_optimizer)
      sndr_opt_finish:
        Sender(opt_finish, analyzer_optimizer)
      rcvr_opt_finish:   Receiver(opt_finish, main)
      sndr_start_gen:    Sender(start_gen, main)
      rcvr_start_gen:
        Receiver(start_gen, code_generator)
      sndr_gen_finish:   Sender(gen_finish, code_generator)
      rcvr_gen_finish:   Receiver(gen_finish, main)
      snrd_main_finish:  Sender(main_finish, main)

      start_main_before_lex:
        Before(start_main, start_lex)
      start_main_before_parse:
        Before(start_main, start_parse)
      start_parse_before_finish:
        Before(start_parse, parse_finish)
      finish_parse_before_start_opt:
        Before(parse_finish, start_opt)
      start_opt_before_finish:
        Before(start_opt, opt_finish)
      finish_opt_before_start_gen:
        Before(opt_finish, start_gen)
      start_gen_before_finish:
        Before(start_gen, gen_finish)
      finish_gen_before_main:
        Before(gen_finish, main_finish)
END compiler_L3
```

The associations between these two levels are

```
precedence_1 --> finish_parse_before_start_opt
precedence_2 --> finish_opt_before_start_gen
```

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] M. Abadi and L. Lamport, "Conjoining Specifications", Technical Report 118, Digital Systems Research Center, Palo Alto, California, December 1993.

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

[4] 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.

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

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

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

[8] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments", *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994.

[9] 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.

[10] S.L. Gerhart, "Knowledge about programs", *Proceedings of the International Conference on Software Reliability*, Los Angeles, California, April 1975, pp. 88–95.

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

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

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

[14] D. Katiyar, D.C. Luckham, and J. Mitchell, "A type system for prototyping languages", *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, Portland, Oregon, 1994.

[15] 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.

[16] G.R. McClain, editor, *Open Systems Interconnection Handbook*, McGraw-Hill, New York, N.Y., 1991.

[17] 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.

[18] M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures", *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994.

[19] R. Reiter, "Deductive Question-Answering on Relational Databases", in *Logic and Data Bases*, H. Gallaire and J Minker, Eds., Plenum Press, 1978, pp. 149–177.

[20] 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.

Mark Moriconi received a Ph.D. degree in computer science from the University of Texas at Austin in 1978.

He joined the Computer Science Laboratory of SRI International in 1978 and has been its Director since 1989. Prior to joining SRI, he was a research scientist at the University of Texas at Austin and a research assistant at USC Information Science Institute. His main research interests are in the use of formal methods in software development. He is currently working on formal methods for architecture-based software composition.

Dr. Moriconi is a member of the Association for Computing Machinery, and the IEEE Computer Society. He is on the editorial board of *IEEE Transactions on Software Engineering* and has served on numerous technical program committees in the areas of software engineering and formal methods. He is General Chair for the upcoming ACM SIGSOFT '96 Symposium on Foundations of Software Engineering, which will have a special focus on software architecture.

Xiaolei Qian received the B.Sc. degree from Xian Jiao Tong University, Xian, China, in 1982, and the M.Sc. and Ph.D. degrees from Stanford University, Stanford, CA, in 1984 and 1989, respectively, all in computer science.

She has been a Computer Scientist in the Computer Science Laboratory at SRI International since 1991. Before joining SRI, she was a Member of the Technical Staff at AT&T Bell Laboratories, and a Computer Scientist at Kestrel Institute. Her research interests include software architectures, semantic interoperation and integration of heterogeneous databases, and database security. She is also interested in database programming languages and formal methods.

R. A. Riemenschneider received the B.S. (*summa cum laude*) degree in physics and mathematics from Miami University in 1973 and the M.A. degree in mathematics from the University of California at Berkeley in 1975.

He joined the Computer Science Laboratory of SRI International in 1991 as a Senior Software Engineer, where he performs research and development on applications of logic to software engineering. Prior to joining SRI, he was a Senior Research Scientist at Advanced Decision Systems, a founder of Reasoning Systems, a Computer Scientist at Systems Control Technology, and an Instructor at the University of California at Berkeley and the California State University at Hayward.

Mr. Riemenschneider is a member of the Association for Symbolic Logic, the Association for Computing Machinery, and the IEEE Computer Society.