A Syntactic Theory of Software Architecture

Thomas R. Dean and James R. Cordy

***Abstract*— In this paper we introduce a general, extensible diagrammatic syntax for expressing software architectures based on typed nodes and connections and formalized using set theory. The syntax provides a notion of abstraction corresponding to the concept of a subsystem, and exploits this notion in a general mechanism for pattern matching over architectures. We demonstrate these ideas using a small example architecture language with a limited number of types of nodes and connectors, and a small taxonomy of architectures characterized as sets of patterns in the language.**

***Index Terms*— Software architecture, software structure, pattern matching.**

I. Introduction

GARLAN AND SHAW [7], [10] argue that larger systems require a higher level of abstraction. Larger and more complex systems are not easily handled using current techniques. They claim that simple identification of system structures and types is not sufficient; in order to facilitate meaningful analysis and comparison, they must be expressed using a uniform notation. Our work provides a first, syntactic level approach to addressing this *software architecture* level.

The motivation for our work is to provide a formal syntax that will serve as a framework within which we can discuss other issues, such as component and system behavior, and to compare and contrast different architectural styles. We use a diagrammatic representation in order to be consistent with traditional practice.

Our syntax must be useful at several levels. The first is to expose the architectural structure of individual systems. At this level it provides a foundation on which the semantics of individual components and the system as a whole may be based. To be general, it is important that the syntax should not bias the expression of a system to any particular paradigm of system organization.

At the second level, the framework must abstract details of particular components, and provide a means of categorizing architectural paradigms. Since our framework is syntactic, the architectural paradigms we classify are syntactic paradigms. To serve as a framework for an architectural theory, the mechanism must be open. That is, it must provide a means of including constraints on system classification based on semantics introduced at the first level.

Manuscript received February 1994; revised November 1994. Recommended by D. Perry and D. Garlan. This work was supported by Natural Science and Engineering Research Council of Canada (NSERC) and the Information Technology Research Centre of Ontario (ITRC).

The authors are with the Department of Computing and Information Science, Queen's University at Kingston, Kingston K7L 3N6, Canada.

IEEE Log Number 9419575.

At the third level, the framework must provide some means of expressing architectural paradigms. It is not enough simply to be able to state that a given system description is a member of a particular architectural class, it must also be possible to describe each paradigm in a manner that allows new systems to be instantiated from the paradigm. We believe that the approach presented in this paper provides a syntactic framework that works at all of these levels.

A. Outline

We approach the problem in three steps. We begin by identifying those characteristics of an architectural syntax necessary to our approach and present a simple diagrammatic language that illustrates these characteristics in an informal manner in Section II. The example language is intentionally incomplete—it lacks semantics, and features have been kept to a minimum in order to focus the readers attention on the approach rather than the details. Section III formalizes the example notation and defines several operations, including interface, abstraction and equivalence.

The second step is to provide a syntactic pattern matching mechanism that exploits the characteristics of the notation. The pattern mechanism is not tied to the example language, but will work with any architectural language with similar characteristics. It can be easily extended to take semantics into account. The pattern mechanism is described in Section IV.

The third step is to show how the pattern mechanism can be used to construct a taxonomy of architectural styles. For any given architectural language, the taxonomy will reflect those elements that can be distinguished in the language. Since only the syntax of the language is described, the taxonomy is limited to the syntactic structures that may be distinguished by the language. We show an example taxonomy expressed as patterns of our simple example language. Surprisingly, even with the limited syntax of our example language, a rich taxonomy can be built encompassing most of the common architectural paradigms. The taxonomy is described in Section V.

Section VI provides some conclusions and directions for future research.

II. Architectural Syntax and an Example Notation

The essential characteristics of an architectural syntax upon which our method depends are: the use of syntactic types to characterize the major kinds of components and connections in a system; a structural definition for interfaces of systems; and a notion of equivalence based on the structural interfaces. These characteristics form the basis of the pattern matching mechanism which is at the foundation of our work.

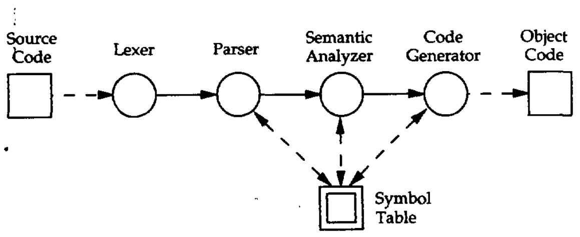


Fig. 1. Canonical compiler structure.

This section describes types, interfaces and abstraction in an informal manner through the introduction of an example architecture language. The notation is formalized in the next section, where equivalence is also defined.

A. Types

The pattern mechanism relies on the use of syntactic types to distinguish between the elements of a system. Types abstract some aspect of a component or connections intended function or behavior in much the same way as data types abstract function and behavior in programming languages. Types may also impose syntactic constraints on the permissible relations between elements of the system. For the purposes of understanding system structure paradigms, we are interested only in the form of the system, not in its function.

The choice of types in the example language introduced below is intentionally limited. This was done to emphasize the method rather than the particular types chosen. Other realizations of our approach may have more types of elements, either as separate types, or as subtypes of the existing element types. These alternate notations may also provide stricter interpretations of the types. That is, the semantics of the new types may impose more restrictions on the properties common to instances of the types.

B. An Example Architecture Language

The syntax of our example language is based on typed, directed multigraphs. A typed, directed multigraph provides typed nodes and typed edges and permits more than one edge of a given type between nodes. We extend the multigraph to allow edges with arbitrary arity: these may connect any number of nodes.

We introduce our example language using example systems that motivate the choice of the types. Fig. 1 shows our architectural representation of the canonical compiler structure typically used in undergraduate compiler courses. It is not a complete representation of a real compiler since the error stream is not represented, nor are multiple input files.

Rectangles represent memory elements of the system. Simple rectangles are *Files*. Files are malleable data repositories that are intended to be accessed sequentially. The symbol composed of nested rectangles is a *Random Access Repository*—or simply, *Repository*. The repository type models memories that may be modified and accessed at random (e.g., shared memory and databases). In the example, it models the symbol table that is shared between the parser, semantic analyzer and code generator. Circles represent *Tasks* , which are the active

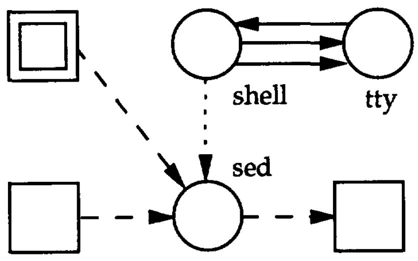


Fig. 2. Possible shell characterization.

components of the system. At present, tasks are the only active component type.

The types of the connections are indicated by the graphical characteristics of the arrows. Fig. 1 shows two types of connectors, *Streams* and *Memory Access Connectors*. Streams are represented as solid arrows and are binary connections between tasks. They represent the direct exchange of data between tasks and may be unidirectional or bidirectional. As with all of the connectors, the implementation or the protocol is not represented. The stream may be implemented over a network, and the protocol may involve message passing. Our notation only shows the type of the connection between the two tasks. In the example, all of the streams are unidirectional. The arrows with short dashes represent memory access connections. They are binary connections between a task and a memory component. As with streams, they may be unidirectional or bidirectional.

Fig. 2 shows the possible characterization of a Unix shell creating a *sed* (stream editor) process. The task in the upper center of the diagram represents the shell. It has a single read stream and two write streams to the task that represents the terminal driver. The two write streams are distinct and represent *sdout* and *stderr.*

The dotted arrow between the shell task and the sed task is an *invocation* connector. It represents one process starting another process. This connection and the *production* connection are used to model dynamic changes to the system structure. The production connection models the creating of a task by another and is represented by a curved arrow. An example of production is the link phase of a compiler, or a task that produces another custom task for a problem.

The rectangle with the grid in Fig. 2 represents a *table.* Tables are intended to model long term data that undergoes few if any changes, such as static system data. The name "table" is not intended to imply any particular internal structure or format. For example, the table in Fig. 2 represents the script file that contains the edit commands for the stream editor.

Fig. 3 shows a characterization of several tasks communicating over a network. The arrows with the double heads are *procedure* connections. These represent the connections in layered systems. In the example, the two tasks at the bottom of the figure represent the kernel layer. The other tasks represent tasks using the kernel layer to communicate over the network.

The network is represented by a *message* connector, depicted as a line with two double bodied arrows connecting the tasks to the line. The message connector is the one type in

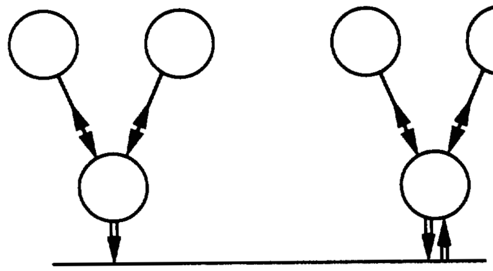


Fig. 3. Procedure and message connectors.

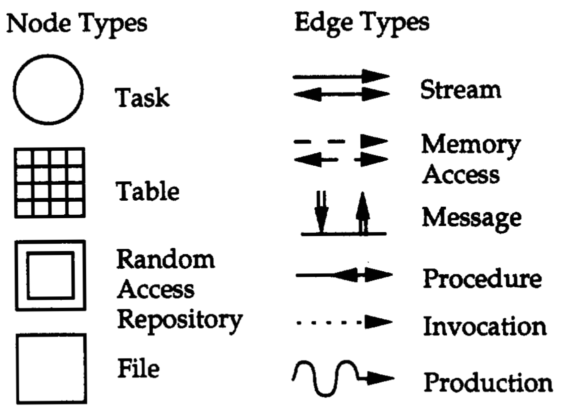


Fig. 4. Summary of notation symbols.

our example types that may connect an arbitrary number of tasks. Each task may have one or more read and write sites on the connector, represented by the double bodied arrows. In the example, the tasks on the left may only send data to the tasks on the right, since the kernel process on the left has only a write site on the message connection.

As with all connectors, the message connector is not restricted to networks. All of the tasks may be on the same machine. The message connection models a software structure where a task may use a single connection to send information to one or more other tasks, either singly, or as a group. The purpose of the message connector in our example notation is to show that architectural languages should not be restricted to binary, or even fixed arity connectors. Architectural languages should allow the representation of a wide variety of structures, and not limit the user to a particular set of paradigms.

Fig. 4 provides a summary of the symbol types used in our simple example architecture language.

C. Partial Systems

If we are to be able to analyze larger system architectures, it must be possible to describe the parts of a system separately. One such part of a system is an *incomplete connection.* An incomplete connection is a connection that is part of a system, but is missing a node. An example is a memory access connector connected to a task, but not connected to a memory node. Since a message connector may connect an arbitrary number of nodes, it can not be an incomplete connection. It may, however, be an *expected connection.* An expected connection is an indication that a connector of that type may be used to connect the partial system to other systems. Expected connections are shown in gray. A *partial* system is one that contains incomplete connections or expected connections.

Expected connections of a partial system may be thought of as variable connections. That is, they are place holders that may be bound to a connector to incorporate the subsystem into another system. Incomplete connectors may be thought of as connecting one or more variable nodes. An incomplete connector may be used to connect two or more partial systems by binding the expected connections of the systems to the incomplete connector, and binding the variable nodes of the incomplete connector to nodes of the subsystems.

Although any partial system could be described, our notation distinguishes two types of partial systems: *connector subsystems*, which fulfill the role of connectors and *nodal subsystems*, which play the role of nodes. The two types of partial systems correspond to the two types of abstraction provided by the notation, which are described next.

D. System Interface and Abstraction

Individual elements of a given system may represent entire systems. This idea is not new and has been used in more than one approach including that of Abowd, Allen, and Garlan [1], [2]. An important nontraditional aspect of our notation is that it supports encapsulation of subsystems as connectors as well as components.

The concepts of interface and abstraction are tightly coupled in our approach. The interface of an element and the subsystem it represents (or abstracts) must match. Although this idea not new, it forms one of the cornerstones of our pattern matching mechanism. It also provides the means to incorporate semantic information into pattern matching. Pattern matching is explained in Section IV.

Our *interface* function transforms a partial system into the minimum system that may be used in the same way as the original system. In our example notation we use a simple rule: when a single primitive can replace a partial system, we use the symbol for that primitive to represent the abstraction of that subsystem. This special case is called *homogeneous* abstraction. The general case, *heterogeneous* abstraction, is handled separately. We discuss both these types of abstraction for both nodal and connector subsystems.

E. Homogeneous and Heterogeneous Abstraction

When the only nodes inside an abstracted subsystem that have connections outside the group are of the same type, the abstraction is said to be *homogeneous*, and the symbol used for the group is the symbol of the type. Fig. 5 shows an example use of homogeneous abstraction to abstract a task and table subsystem into a task abstraction. The figure represents a possible characterization of the lexer phase of a canonical compiler, where the table is used to drive some skeleton algorithm.

For *heterogeneous* abstraction, a group of nodes and the connections between the nodes is replaced by a single node, the type of which is *heterogeneous node*. The new node assumes

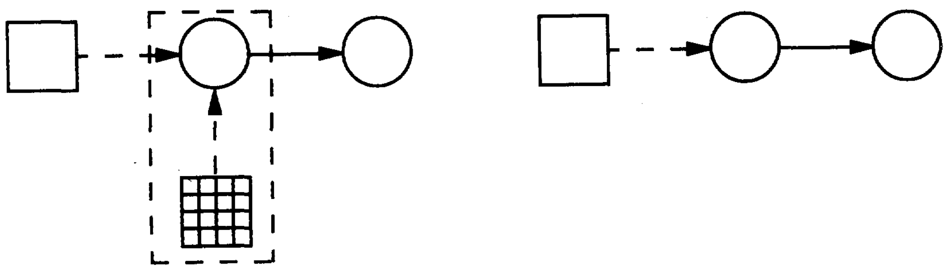


Fig. 5. Homogeneous abstraction for nodes.

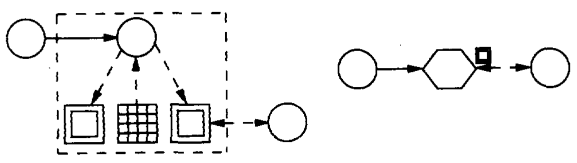


Fig. 6. Heterogeneous abstraction for nodes.

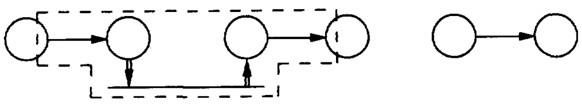


Fig. 7. Homogeneous abstraction for connectors.

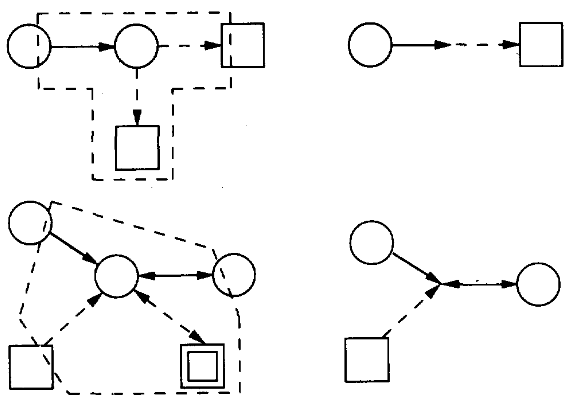


Fig. 8. Heterogeneous abstraction for connectors.

all of the external connections of the group and preserves the types of the connections. The symbol for heterogeneous nodes is the hexagon. Fig. 6 shows an example of heterogeneous abstraction. Several memory nodes and a task are abstracted. The small diagram of a repository is shown next to the junction of the memory access connector and the hexagon representing the group to indicate that the connector involves a repository. If the node was a file or table, then a small file or table would be shown next to the junction of the connector and the hexagon.

As with abstraction for nodes, there is a homogeneous case for connector abstraction. When there are two connectors involving nodes outside the system, and both connectors are the same type and direction, the new abstract connector is represented by a single arrow of the type. (Since a message connector can connect an arbitrary number of nodes, there is no concept of an incomplete message connector. Hence, there is no homogeneous abstraction for message connectors.) Fig. 7 shows an example of homogeneous connector abstraction. In this example, the message connector, the tasks connected to it, and the streams are replaced by a stream.

Fig. 8 shows two examples of heterogeneous abstraction for connectors. The first is a connection that translates a stream into a file. The other shows that heterogeneous abstraction is not limited to binary connectors. In both cases, the connector subsystem outlined by the dashed polygon on the left is replaced by a composite connector on the right. The composite connector is treated as a single connector.

III. Formalization

This section presents a mathematical model of the example notation and types which will be used to define operations, one of which is pattern matching. We define several basic operations in this section. Pattern matching is deferred until the next section. The model is not limited to the types we have chosen, and may be augmented with other types, provided that the definitions of the operations are also augmented.

This particular formalization is specific to our example language, although it can easily be extended to richer languages. It does not support any fixed arity connectors other than binary, and the arbitrary arity of the message connector requires some special handling. The main purpose of the formalization is to precisely define the operations *interface* , *equivalence*, and *specialization*. Any formalization that defines these three operations may be used in our pattern matching approach.

Our model is based on set theory. Elements of the graph are represented using sets and relations. The primitives of the notation are represented using the following sets:

N	the set of nodes in the system;
C	the set of connectors in the system;
MS	the set of message sites in the system;
V	the set of variables in the system.

The nodes are the tasks and memory elements of the system. The set of message sites is used to model the arbitrary arity of the message connectors. *Message Sites* link message connectors and the tasks they connect.

Variable elements are used to handle partial systems. Variable nodes are used to model incomplete connections and variable connectors represent expected connections. The set V identifies the variable elements of N and C. We define the tuple $EL = \langle N, C, MS, V \rangle$ as the *elements* of the system.

The elements of N, C, MS, and V are given types using the tuple *Types* = ⟨*Task*, *File* , *Repository*, *Tbl* , *Hetero*, *MetaMorph* , *Message*, *Stream* , *FileAcc*, *TblAcc* , *RepAcc*, *Proc* , *Prod*, *Invoke* , *Read*, *Write* , *Bidir*, *MRead* , *MWrite*⟩. The sets *Task*, *File*, *Repository*, *Tbl*, *Hetero*, and *MetaMorph* partition the set N. The sets *Message*, *Stream*, *FileAcc*, *TblAcc*, *RepAcc*, *Proc*, *Prod*, and *Invoke* partition the set C. The set *Read*, *Write*, *Bidir*, *MRead*, and *MWrite* are used to provide additional attributes of the elements. The *Types* tuple is used to refer to these sets as a group when discussing operations on systems. The examples of systems shown in this paper enumerate the nonempty sets explicitly. The unspecified sets are assumed to be empty and the Types tuple will be assumed to be the list of explicit and assumed sets.

Four binary relations model the connections between elements. These are the following:

$src \subseteq N \times C$	Nodes that are the source of binary connectors.
$dst \subseteq N \times C$	Nodes that are the destination of binary connectors.
$has \subseteq N \times MS$	Nodes that have message sites.
$msg \subseteq MS \times C$	Message sites associated with message connectors.

The relations *src* and *dst* are used to model the source and destination of binary connections. In the example notation, the source of a connection is always a task. Since the notation may be changed to include more types, we use the more general set *N* as the domain of the *src* relation.

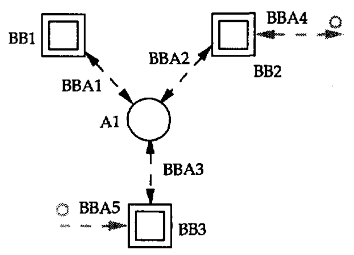
The *has* and *msg* relations model message connections. The *has* relation associates message sites with tasks, and the *msg* relation associates the message sites with message connectors. We define the tuple *Connections* = ⟨*src*, *dst*, *has* , *msg*⟩ as the connections of the system. A system is then defined as the tuple *System* = ⟨*EL*, *Types*, *Connections* ⟩.

Fig. 9 shows an example of a system and its model. The expected connectors are represented by variables. A variable connector is a member of the range of at most one of the *src* or *dst* relations. As mentioned previously, the source of binary connections is always a task, thus the tuple ⟨*BB2,BBA4*⟩ is part of the *dst* relation.

A. Well-Formedness

Restrictions on the sets and relations define well-formedness. A system is a *well-formed* system if and only if these restrictions hold for the model of the system. These restrictions fall into three categories. The first group describes subset restrictions between sets and domain restrictions for the relations. For example, only a member of the *File* subset of *N* may be related to the *FileAcc* subset of *C* by the *dst* relation. There are fourteen of this kind of restriction.

The second category of restrictions, of which there are ten, limits the cardinality of the relations. As an example, only one



N = { A1, BB1, BB2, BB3 } V = { BBA4, BBA5 }
C = { BBA1, BBA2, BBA3, BBA4, BBA5 } Task = { A1 }
RepAcc = { BBA1, BBA2, BBA3, BBA4, BBA5 } Repository = { BB1, BB2, BB3 }
Bidir = { BBA1, BBA2, BBA3, BBA4 } Write = { BBA5 }
src = { <A1,BBA1>, <A1, BBA2>, <A1, BBA3> }
dest = { <BB1,BBA1>,<BB2,BBA2>, <BB2,BBA4>, <BB3,BBAA3>, <BB3,BBA5> }

Fig. 9. Example translation of a subsystem.

node of a system may be the source of a binary connector.

The last category of restrictions are general restrictions that involve arbitrary relations between elements. One example is that *V* is a subset of *N* or *C*, but not both. This represents the restriction that a partial system is either a nodal subsystem or a connector subsystem (since we have no meaningful abstraction for a mixed subsystem). Another example of this kind is the restriction that all connector subsystems must have at least two variable nodes. There are four of these restrictions. The complete set of well-formedness restrictions is given in [4].

B. Interface and Equivalence

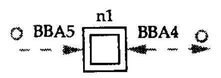
We define several operations on the semantic model. The *interface* operation is the formal equivalent of the interface function described informally in the previous section. The interface of a complete system is always a single task. The interface of a nodal subsystem is a single node with the same expected connections. The type of the node is given by the abstraction rules for nodes. Similarly, the interface of a connector subsystem is a connection whose type is given by the abstraction rules for connectors given in the previous section.

The interface operation produces a system with a new nonvariable node or connector and all of the variable elements of the argument system. The resulting system relations (*src, dst, has, msg*) contain only the entries involving the variable elements, with nonvariable elements replaced by the newly created nonvariable element. The formal definition of this function is provided in [4].

Fig. 10 gives the interface of the system shown in Fig. 9. The nonvariable elements of the system have been replaced by a single node, *n1*. The variable elements remain the same, but are connected to the new node.

The interface of a system is used to define system equivalence. System equivalence, in turn, is used to define pattern matching. A related concept, *specialization*, is also based on the interface of a system. System specialization is used to define subpattern matching.

Two systems are *equivalent* if and only if they have the same interface. That is, two systems are equivalent if we can construct a bijection between their interfaces as follows:



N = { n1 } C = { BBA4, BBA5 }
Repository = { n1 } V = { BBA4, BBA5 }
RepAcc = { BBA1, BBA2, BBA3, BBA4, BBA5 }
Bidir = { BBA4 } Write = { BBA5 }
dest = { <n1,BBA4>, <n1,BBA5> }

Fig. 10. Interface of Fig. 9.

- the types of all the related elements are the same and,
- all instances of the connection relations (*src, dst, has, msg*) between elements of one interface hold between the related elements of the other interface.

We use the form A ≡ B if the system A is equivalent to the system B.

A system A is a *specialization* of a system B if and only if there exists an injection from the element of the interface of A to the elements of the interface of B that preserves types and connection. We define *generalization* as the converse of specialization. That is, system B is a generalization of system A if and only if system A is a specialization of system B.

This definition of equivalence and specialization relies on the syntactic definition of interface. If semantics are incorporated into the notation, they may also be used to refine the definition of equivalence and specialization. This will also implicitly refine the pattern matching mechanism. The semantics incorporated into the refined definitions must be limited to the semantics of the types, not the semantics of individual elements.

IV. Pattern Matching

The definitions of the previous section form the basis of our pattern matching mechanism. We present pattern matching in three steps. The first defines simple pattern matching using only the rules of abstraction. The second step extends pattern matching with constructs similar to regular expressions in string languages. This step is defined in Section IV-A. The last step adds the equivalent of context free grammars to express recursive patterns. It is presented in Section IV-B. Section IV-C presents one possible way of evaluating the best match when more than one pattern matches a given system.

Two definitions needed for simple pattern matching are *minor systems* and *singular minor systems.* System B is a *minor system* of system A if and only if its elements and relations are a subset of those of system A. A *singular minor system* is a minor system that contains a single nonvariable element. Heterogeneous connector subsystems are treated as a single element.

A system is a *simple pattern* for another system (the *target*) if, and only if, the target system can be partitioned into a set of minor systems corresponding to the elements of the pattern. Thus, a system P is a simple pattern for a system Q if, and only if, P ≡ Q and there exists an injection ϕ from the set of singular minor systems of P to the set of minor systems of Q and an injection ρ from the message connectors of P to the message connectors of Q such as follows.

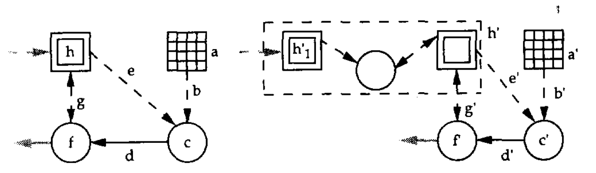


Fig. 11. Example pattern match.

- For all ϕ-related minor systems A of P and B of Q, A ≡ B.
- The range of ϕ partitions Q.
- The range of ϕ are connected systems.
- The ϕ-related minor systems of Q are connected in the same way as the singular minor systems of P.

The systems chosen to be the range of ϕ must partition Q. That is, all elements of Q must be an element of one of the minor systems in the range of ϕ. Each of these minor systems of Q must also be connected, preventing the abstraction of several unrelated elements. For example, several repositories could not be grouped into a single repository unless some subsystem joining them was also included.

The final restriction on the two injections is that the chosen minor systems of Q must be connected in the same way as the singular minor systems of P. That is, if a connector subsystem and a nodal subsystem of P are joined (say a task and a stream), then the corresponding connector subsystem and nodal subsystem of Q must also be joined. The injection ρ is used to map message connectors in P to message connectors in Q since they may not be connector subsystems.

Fig. 11 shows two systems, one of which is a pattern for the other. The pattern system is the system on the left and the matched system is the system on the right. The letters indicate the elements associated by the injection. The only nonsingular minor subsystem matched in the target system is formed by the two repositories and the task that connects them (h'). It is enclosed in dashed lines to identify the matched components. The bidirectional memory access indicated by g' in the target system is not part of the subsystem h' since it crosses the boundary.

In practice, most systems will not be matched in their entirety by the taxonomy patterns. For example, the canonical compiler contains components that can be characterized as a pipe and filter system. It also has overlapping components that can be characterized as a shared memory system. Any useful method of characterizing system structures should be able to handle structures embedded within systems. We define a *subpattern match* as a matching of a pattern system to a system embedded within another system. Thus, we can view the compiler as either an instance of a pipe and filter system or of a shared memory system. The definition is identical to pattern match except that specialization is used instead of equivalence, and the range of the pattern match injection does not have to partition the target system. That is, not all elements of the target system must be matched.

Although pattern and subpattern matching are useful, more expressive patterns are required. If, in the example in Fig. 1, a bidirectional memory access connection was added between the repository h'_1 and the task f', the pattern would no longer

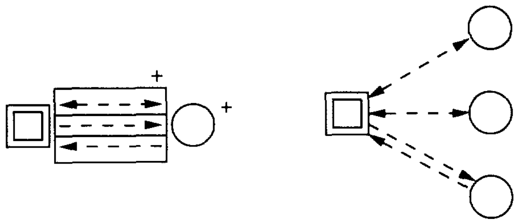


Fig. 12. Central repository pattern and an example matched system.

match. There is no element in the pattern to correspond to the extra connection, and no way to repartition the target system to produce a match that includes the new connector. To handle this problem we augment the patterns with symbols that indicate families of patterns. The augmented pattern matches a system if any of the patterns generated from the augmented pattern match the system. There are two complimentary approaches. The first, based on regular expressions uses simple alternation and repetition operators. The second is based on graph grammars [5] and is used to express more complex repetitive structures.

A. Regular Expressions

Fig. 12 shows the pattern for the central repository class of system architectures and a system matched by the pattern. The rectangle around the repository access connectors with dividing lines provides alternation. The "+" symbol is used to indicate one or more repetitions of a pattern element. In the example, the "+" next to the alternation means one or more memory access connections between the repository and a task.

The operator has a higher precedence when applied to a node than to a connector. In the example, the "+" applied to the task is expanded before the "+" applied to the connection element. Therefore, the systems recognized by the pattern are systems composed of a single repository (or system that abstracts to a repository) and several tasks (or systems that abstract to tasks), and each task is connected to the repository by one or more memory access connectors.

The regular expression operator binds to the closest element. When the operator is near the point where a connection joins to a node, the operator applies to the node.

Fig. 13 gives the pattern for a distributed repository and an example of a system that is matched by the pattern. The difference between this pattern and the previous one is that the repository is also modified by the "+" repetition operator and the alternation for the memory access connectors is now modified by the "*" repetition operator. This pattern matches one or more repositories that are connected to one or more tasks. Not all pairs of repositories and tasks need be connected ("*" means zero or more).

Two other operators are provided, the "?" and "!" operators. As in regular expressions for strings, the "?" operator indicates an optional element. In all of the cases presented so far, an element of the system matches if there is a minor system with the same interface. The "!" operator restricts the match to be a singular minor system. This operator may be applied to connectors and to any grammatical construct. It may also be combined with the "* ," "?," and "+" operators.

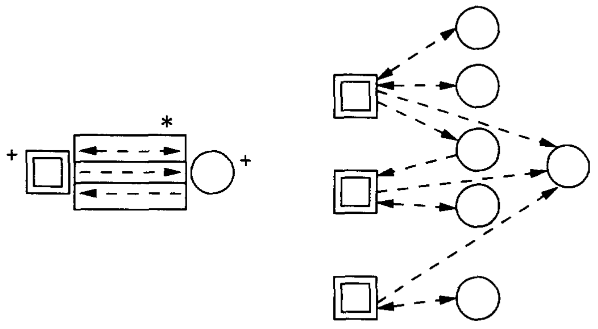


Fig. 13. Distributed repository pattern and example matched system.

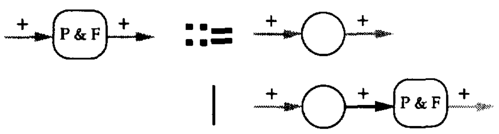


Fig. 14. Simple pipe and filter system without feedback.

These repetition operators provide parallel repetition in the patterns. That is, the repetition of elements connected to the same element or group of elements. Two examples of parallel repetition are multiple streams between two filters, and multiple transaction programs accessing a single database. Recursive repetition involves an element that will connect, in turn, to the elements generated by the pattern. An example is the pattern for a pipe and filter system. The next section describes a method to specify recursive patterns.

B. Grammar Productions

To handle recursive repetition, we extend the pattern mechanism to provide the equivalent of context free grammars. This is done using a limited version of graph grammars [5]. The full power of unrestricted graph grammars is not required for specifying the patterns we are interested in. Unrestricted graph grammars provide arbitrary rewriting of graphs. We are interested only in the subset of graph grammars that provides syntactic recognition.

Fig. 14 shows a pattern that matches simple pipe and filter systems with no feedback between the filters. The rounded rectangle is used to represent nonterminal nodes. A simple pipe and filter system without feedback is a task with two multiple stream connections, or a task connected by a multiple stream connection to a simple pipe and filter system without feedback.

All rules have a single nonterminal element on the left hand side. The nonterminal has a context that consists of primitive elements of the notation. This context consists only of the primitive elements that may directly connect to the nonterminal element. The right hand side is an arbitrary system that may contain nonterminal elements. The interface of the right hand side must be identical to the context of the left hand side.

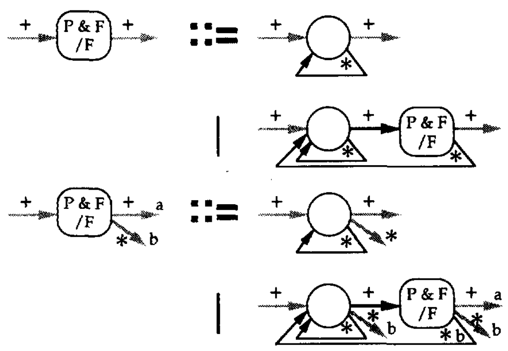


Fig. 15. Pipe and filter system with feedback.

There may be more than one rule for a given nonterminal, and rules may have different contexts. Fig. 14 shows a short form for multiple rules with the same context. Each of the alternate right hand sides is separated by a vertical bar. Rules may only be applied if the context of the embedded nonterminal matches the context of the rule.

When more than one element of the context is of the same type and has the same attributes then the context elements must be labeled. These labels are local to the rule and cannot be used to govern the application of subsequent rules to nonterminals embedded within the right hand side. All possible bindings of ambiguous context elements are tried when generating a pattern system.

When a production is applied to a nonterminal node, only the ends of the connectors connected to the nonterminal node are changed (to the new nodes introduced by the production). The other ends of the connections remain connected to the same nodes.

As with conventional grammars, more than one rule may be provided for a given nonterminal. The same nonterminal node may be defined for more than one context. However, only those productions which have the same context as the nonterminal node may be applied.

Although the capabilities of the regular expression operators may be provided by the grammar mechanism, we believe that the regular expression operators provide a concise representation of some structures. The grammar version of the structures would not be as clear. Instead, we show how they may be combined.

The system matched by applying the "?," "+," or "*" operators to a nonterminal is equivalent to replicating the nonterminal before expanding it. That is, not all of the systems matched by the nonterminal must be matched by the same parse.

The addition of repetition operators requires some changes to the rules governing the context of left and right hand sides of grammar productions. Context elements that have been modified with the regular expression operators may be combined by labeling them with the same label. That is, a single context element from the left hand side may be represented by more than one context element on the right-hand side subject to the following rules.

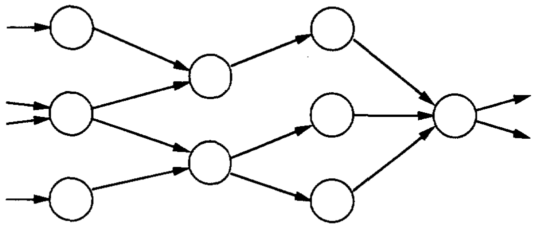


Fig. 16. Example pipe and filter system.

- Elements with the "*" operator may be split into multiple elements of the same type, each modified with the "*" operator.
- Elements with the "+" operator may be split into multiple elements of the same type, each modified with the "+" operator.

Two cases must be explained. The first is matching the context of the left- and right-hand sides. This is done by combining those elements on the right hand side with the same label (subject to the rules above) and checking to make sure that the contexts match. The second case is determining the contexts of embedded nonterminals to decide which rule to apply. The context of the embedded symbols may also be labeled and a similar procedure is followed. The elements with the same labels are merged and the rule with the same context is selected. Fig. 15 shows an example.

The figure shows the pattern for a pipe and filter system with feedback. To do this, the system generated by a nonterminal must be able to communicate with all of the filters generated by previous grammar rules. Examples of both types of context merges are given in the fourth rule. The context of the left hand and right hand sides match since the two streams modified with the " *" operator are both given the label "b." The second set of rules is used for subsequent grammar productions of the nested nonterminal, because both streams modified by the "* " operator have the label "b."

C. Strength of Pattern Match

Some systems may be matched by more than one pattern. For example, the pattern given in Fig. 14 only matches a pipe and filter system composed of a single sequence of tasks without feedback. If we modify the pattern, adding "+" operators to the tasks, we get a pattern that matches pipe and filter systems that are comprised of stages of tasks. Fig. 16 shows an example system matched by the new pattern. This pattern also matches the same systems matched by the pattern in Fig. 14. Thus, it is advantageous to have some metric of the *strength* of the match. The ideal definition of the strongest match is the pattern that matches the fewest systems, although it still might match an infinite number of systems. In this section we define an approximation.

Suppose we have two pattern systems that match a given target system. We start by defining a partial order between corresponding elements of the two patterns that match the same minor system of the target system. The partial order is defined based on how the elements have been modified

$* \leq +$	$* \leq !*$	$!* \leq !+$
$+ \leq ?$	$? \leq !?$	$!+ \leq !?$
$+ \leq \sim$	$\sim \leq !\sim$	$!+ \leq !\sim$
$? \leq {}^\circ$	$* \leq {}^\circ$	

Fig. 17. Partial order for regular expressions.

by the regular expression operators, and the "!" operator. This relation, for which we use the "$\leq$" symbol, is given in Fig. 17. For elements a of one pattern and b of another, $a \leq b$, if b is a better match than a. The symbol "$\sim$" is used to represent an unmodified element and the symbol "$^\circ$" is used to represent an element missing from the pattern. Since a partial order is transitive and reflexive, the transitive and reflexive closures of Fig. 17 are also defined. We extend the partial order from individual elements of the two patterns to the entire patterns in the obvious way. One pattern match is stronger than the other if the partial order holds for all elements of the two patterns.

V. A Taxonomy of System Architectures

This section describes a small taxonomy of software architectures based on our example architectural language. Since it is based on the example notation, it is limited to syntactic differences in the topology and types of system descriptions. Even so, it provides a variety of structure classes, and can represent many of the types of architecture described in the literature. The purpose of such a taxonomy is to provide a syllabus of useful system structures. The next step, discussed in the section on future work, is to evaluate the taxonomy against real systems and add information on the applicability of each class to different types of problems.

Each class of any taxonomy developed using our approach is described as a set of patterns. A given system is a member of the class if it is matched by one of the patterns in the set. The classes of our taxonomy are derived from two sources, Garlan and Shaws paper on higher level abstractions [7] and the book *Coordinated Systems* [6]. The taxonomy currently contains six classes, two of which have several subclasses. Some of these patterns have already been shown in figures. Fig. 18 gives an outline of the taxonomy.

The pipe and filter system with nonoverlapping feedback is similar to that with general feedback, but the feedback connectors may not overlap. They may be between successive sets of the filters, or nested. The bidirectional pipe and filter system is similar to the unidirectional case without feedback, but the streams may be bidirectional, or they may be unidirectional in either direction. The message network is a set of tasks that are connected by one or more message connectors. There are no restrictions on the number of message connectors, or the number of tasks.

The pattern for the layered system is shown in Fig. 19. It consists of two productions and is similar to the simple pipe and filter system. The difference is that the procedure call communication primitive is used, and the nonterminal node is replicated. The layered random repository pattern is similar to

Class	*Notes*
Pipe and Filter	
Unidirectional	
Without Feedback	
Simple	Fig. 14
With Feedback	Fig. 15
Non-overlapping Feedback	
Bidirectional	
Simple	
Random Repository	
Central	Fig. 12
Layered	
Distributed	Fig. 13
Message Network	
Layered	Fig. 19
Knowledge Interpreter	Fig. 11, add '+' to connectors
Client-Server	Fig. 12, Task instead of Repository

Fig. 18. Taxonomy outline.

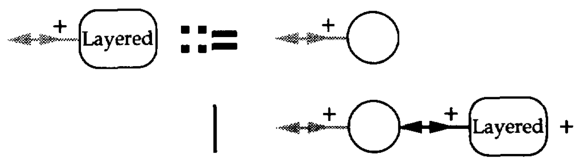


Fig. 19. Layered system pattern.

the pipe and filter pattern. It may be described as a sequence of repositories where each repository is connected to the next by one or more tasks and memory access connections.

The client server pattern is similar to the central repository pattern, but the central entity is a task instead of a repository. The distinction between these two structures is impossible without the types provided by our notation or behavioral information for the central component.

VI. Conclusions and Future work

We have presented a formal, syntactic theory of software architecture based on typed nodes and connections. There are several reasons we believe this to be an appropriate representation for system structure. The first is that it uses a syntactic technique analogous to that used to express programming language structure: regular expressions and context free grammars. It is essentially a diagrammatic form of extended BNF notation, and handling of attributes and semantics can be added using techniques analogous to attribute grammars and denotational semantics for programming languages. The second advantage of the notation is that it is easily extensible to include new primitive element types.

We have shown how types and abstraction can be used to represent system structure and to categorize architectural styles. Even though the technique we use is syntactic, the types allow us to model the intended semantic roles that are important to the structural architecture of the system. Our technique works with any graph-based notation that uses types in this way. This paper presented an example notation that illustrates some of advantages of types, and a taxonomy based on the notation and our pattern matching technique. Practical applications of our approach would likely use a richer set of types, and with a more precise characterization of the semantic properties each type represents.

This paper makes two contributions to the field of software architecture. The first is the pattern matching mechanism, which provides a general means of describing and recognizing classes of software architecture. The other is an initial notation and taxonomy that illustrates these characteristics while remaining open to the addition of other language features.

A taxonomy based on a more general architectural language that includes semantics would provide a means of comparing system structure classes, of classifying the structure of existing systems, and of understanding software architecture in general. The taxonomy can provide a syllabus that may be used to design new systems. Instead of emphasizing a single structure class, designers may choose to use different structures in different parts of a system. The classification of the structure in existing systems may assist in analyzing those systems for the purpose of software maintenance. But most importantly, a taxonomy of system structures may provide a better understanding of the roles of components and their interaction in much the same way that current data structure taxonomies enhance the understanding of procedural programs. We believe our classification framework to be a first step toward this goal.

A. Future Work

We envision several ways in which this research may be extended. Among them are extending the number and scope of element types, attributes, modeling dynamic systems, and providing a reasoning framework.

B. A Richer Set of Types

While the types we have used to motivate our pattern mechanism are capable of describing the connection architecture of a range of systems described in the literature, the notation is by no means complete. In addition, the interpretation given for each of the types was incomplete and informal. For the purpose of our pattern matching mechanism, the existence of separate types is more important than the particular interpretation of the types. Two obvious extensions to the research are to provide a formal definition for each type, and to add new types of nodes and connections. The new definitions of the types would restrict the interpretation of the type, specifying in more detail the properties common to instances of the types. New types of elements may be added as distinct types, or as subtypes of the existing types.

If the new types are added as distinct types, the taxonomy must be expanded to use the new types. This may involve new structure classes, or the additional patterns for existing structure classes to include the new types. With more types, the interpretation of the existing types (i.e., mapping to real world design or implementation artifacts) may be narrowed. The existing types may be completely replaced by a new set of types, particularly if the application domain provides its own element types.

If the new types are added as subtypes of existing types, then the taxonomy needs no modification for the existing classes. Subclasses of the taxonomy may be refined to use the subtypes. For example, a subtype of the central repository structure class may be refined to use a database subtype of repository. While the properties of each subtype must be consistent with the properties of the parent type, the subtype will add additional constraints on the properties of its instances. Thus, the subtypes will provide stricter interpretations of the parent types.

C. Attributes

Adding attributes to the elements of the notation may be useful in specifying design or implementation information. The attributes can be treated as orthogonal to adding new types, and may provide a means of specifying stricter interpretations. Examples of attributes are the buffer size of a stream or the locking protocol for a repository that represents shared memory. If the attributes are incorporated into the definitions of interface, equality and specialization, then they can also be used to refine the taxonomy.

D. Dynamic Architectures

We view the dynamic changes to the structure of a system as orthogonal to the structure of the system. In this way, we may model systems whose dynamic behavior changes its classification within the taxonomy.

A possible approach is to model each singular minor system that can change its interface as a finite state machine. Each interface configuration is a state, and the allowable changes between configurations determine the transitions of the state machine. The model of a Unix shell would have two states. The first is a task that reads from a stream and writes to two streams. The second state is a task with the same stream characteristics but also invoking another task. The transition function simply moves between the two states.

This simple model has some advantages. We can extend the power of the model by substituting push down automata or Turing machines for the finite state machines. Another way is to provide a means of specifying the events that trigger the transitions. Systems can be modeled as compositions of these automata into a single automaton.

E. A Framework for Reasoning

There are two ways in which the notation may be used as a framework for reasoning about systems. The first is to use it as a framework for specifying the behavior of the systems. The notation describes the structure of the system and a notation such as CCS [9] or CSP [8] is used to specify the behavior of elements or groups of elements. For example, a repository may be modeled as a process in these notations, and locking mechanisms and protocols could be defined. Another model that may be used is provided by Allen and Garlan [3]. This method specifies connections as a set of roles and glue logic that specifies the relationship between the roles. Components have ports that may be associated with roles of connectors. CSP is used to specify the ports, roles, and glue logic. A slightly different approach taken by Abowd, Allen, and Garlan [1] uses the Z language [11] to specify the ports, roles, and glue logic.

The other way the notation may be used as a framework for reasoning about systems is the attribute mechanism discussed

```
system( fig9,                                  %% Name
    [                                          %% EL
            [a1, bb1, bb2, bb3],               %% N
            [bba1, bba2, bba3, bba4, bba5],    %% C
            [ ],                               %% MS
            [bba4, bba5]                       %% V
    ], [                                       %% Types
            [a1],                              %% Task
            [ ],                               %% File
            [bb1, bb2, bb3],                   %% Repository
            [ ],                               %% Tbl
            [ ],                               %% Hetero Nodes
            [ ],                               %% Hetero Conns
            [ ],                               %% Message
            [ ],                               %% Stream
            [ ],                               %% FileAcc
            [ ],                               %% TblAcc
            [bba1, bba2, bba3, bba4, bba5],    %% RepAcc
            [ ],                               %% Proc
            [ ],                               %% Prod
            [ ],                               %% Invok
            [ ],                               %% Read
            [bba5],                            %% Write
            [bba1, bba2, bba3, bba4],          %% Bidir
            [ ],                               %% MRead
            [ ]                                %% MWrite
    ], [                                       %% Conn
            [                                  %% src
                    [a1,bba1], [a1,bba2],
                    [a1,bba3]
            ], [                                %% dst
                    [bb1,bba1], [bb2,bba2],
                    [bb2,bba4], [bb3,bba3],
                    [bb3,bba5]
            ],
            [ ],                                %% has
            [ ]                                 %% msg
    ]
).
```

Fig. 20. Prolog representation of Fig. 9.

previously. Since this information can be incorporated into the equivalence and specialization relations, we can use these relations to define equivalence preserving transformations. These transformations can be used to reason about dynamics, or about relations between different classes of system structure. Another important application of the extended notation is a maintenance theory for systems.

Garlan and Shaw [7] show an example of a system with different architectural interpretations. The system is the Hearsay-II speech understanding system. By changing the interpretation of different components, the system may be viewed as a blackboard model (Central Repository in our taxonomy) or an interpreter (Knowledge Interpreter in our taxonomy). In our current notation, this requires a change in the element types to reflect the new interpretation. However, with the addition of semantic interpretations of the symbols, and some set of semantic preserving transformations, the types of views advocated by Garlan and Shaw may be possible.

F. Prototype Implementation

A prototype implementation of the pattern matching algorithm has been built using Prolog. The system is represented as a predicate named *system* with four parameters: the name of the system, and three lists representing the three elements of the *System* tuple from Section III. Tuples and sets are represented as lists. The relations are lists of ordered pairs, each of which is represented as a two element list. Fig. 20 shows an example Prolog encoding of the system from Fig. 9. The system name is *fig9*. The "%" characters indicate the start of Prolog comments, which are terminated by the end of a line.

This representation allows more than one system to be loaded in the database at a time. It also permits temporary systems to be generated and held in variables without adding them to the Prolog factbase. For convenience, rules are provided for each of the element type subsets. These rules take a system as one argument and a Prolog symbol as the other argument and are satisfied if the symbol is part of the relation. As an example, the predicate *task(Sys,a1)* is satisfied if *Sys* is bound to the system in Fig. 20.

We have predicates that check that the well-formedness of a system, compute the interface of a system, and evaluate equivalence and specialization. These predicates are used to implement simple pattern matching and subpattern matching. We have an interface for the visual notation which is able to interact with the Prolog engine.

G. Limitations of the Technique

It is currently not known if our pattern mechanism is sufficient to describe all of the interesting structure classes. It does, however, handle a reasonable number of them. We are a little concerned about controlling the complexity of the descriptions. The graph grammars necessary to describe some involved structures may not be easy to understand. This may simply be a consequence of a complex structure and any representation of the structure would be just as complex.

The technique we have described is entirely syntactic. However, the syntax approach uses the types of the elements to abstract semantics common to instances of the type. We have also shown several ways in which more semantic information may be incorporated into our pattern matching approach.

ACKNOWLEDGMENT

The presentation of this paper was greatly improved by the suggestions of the referees.

REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan, "Using style to give meaning to software architecture," in *Proc. ACM SIGSOFT '93 Symp. Foundations of Software Eng.*, Redondo Beach, CA, 1993, pp. 9–20.

[2] R. Allen and D. Garlan, "Toward formalized software architectures," Carnegie Mellon Univ., School of Comput. Sci., Tech. Rep. CMU-CS-92-163, July 1992.

[3] ——, "Formalizing architectural connection," in *Proc. 16th Int. Conf. Software Eng.*, Sorrento, Italy, May 1994, pp. 71–80.

[4] T. Dean, "Software characterization using connectivity," Ph.D. dissertation, Dep. Comput. and Inform. Sci., Queen's Univ., Kingston, Canada, 1993.

[5] H. Fahmy and D. Blostein, "A survey of graph grammars: Theory and applications," in *11th Int. Conf. Pattern Recognition,*; also in *Pattern Recognition Methodology and Systems Vol. II.* The Hague, Netherlands: Sept. 1992, pp. 294–298.

[6] R. Fillman and D. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software.* New York: McGraw-Hill, 1984.

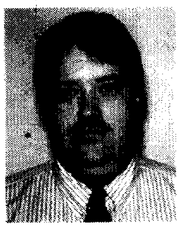
[7] D. Garlan and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering.* New York: World Scientific, 1993, Vol. 1, pp. 1–39.

[8] C. Hoare, "Communicating sequential processes," *CACM,* vol. 21, no. 8, pp. 666–677, Aug. 1978.

[9] R. Milner, "A calculus of communicating systems," *Lecture Notes in Computer Science 92.* New York: Springer-Verlag, 1980.

[10] M. Shaw, "Larger scale systems require higher-level abstractions," in *Proc. Fifth Int. Workshop on Software Specification and Design,* IEEE Computer Society, 1989, pp. 143–146.

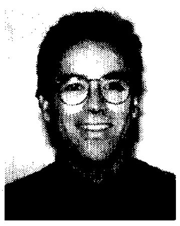
[11] J. Spivey, *The Z Notation: A Reference Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1989.



Thomas R. Dean received the M.Sc. degree from the University of Saskatchewan in 1988, and the Ph.D. degree from Queen's University, Kingston, Canada, in 1993.

He is an Adjunct Assistant Professor of Computing and Information Science at Queen's University. Current research interests include software architecture, design theory and design environments.

Dr. Dean is a member of the Software Technology Laboratory at Queen's University.



James R. Cordy is Associate Professor of Computing and Information Science at Queen's University, Kingston, Canada. He is the co-designer of the programming languages Concurrent Euclid, Turing and Turing Plus, the S/SL compiler specification language, the TXL transformation language and the visual language GVL. He is a co-author of the books *The Turing Programming Language: Design and Definition* (1988) and *Languages for Developing User Interfaces* (1992).

Dr. Cordy was Program Chair of ICCL'92, the 1992 International Conference on Computer Languages, serves on the program committees of ICCL'94 and CASE'95, and is a member of the editorial board of the *Journal of Programming Languages*.