Separating Structure from Function in the Specification and Design of Distributed Systems

Kenneth J. Goldman

A distributed system is viewed as a collection of functional components and a unifying structure that defines relationships among the components. In the paper, we advocate a particular approach to distributed system specification and design in which the structure of a distributed system is specified separately from the functional components. This permits one to reason about individual functional components in isolation, and encourages one to make explicit not only the input/output behavior of the functional components but also the logical placement of these components within the overall structure of the system. We describe a new software tool for the... Read complete abstract on page 2.
Separating Structure from Function in the Specification and Design of Distributed Systems

Kenneth J. Goldman

Complete Abstract:

A distributed system is viewed as a collection of functional components and a unifying structure that defines relationships among the components. In the paper, we advocate a particular approach to distributed system specification and design in which the structure of a distributed system is specified separately from the functional components. This permits one to reason about individual functional components in isolation, and encourages one to make explicit not only the input/output behavior of the functional components but also the logical placement of these components within the overall structure of the system. We describe a new software tool for the specification, design, and simulation of distributed systems that supports this separation of structure and function. The tool, called the Spectrum Simulation System, is based on the I/O automation model of Lynch and Tuttle with extensions for shared memory, superposition, and dynamic process creation. Spectrum consists of a programming language for describing functional components of distributed systems, a configuration mechanism for specifying system structures, and software for simulating the executions of the specified systems. Functional components, described as I/O automats, may be parameterized by the configuration defined separately. This separation gives rise to remarkable versatility in specifications and design.
Separating Structure from Function in the Specification and Design of Distributed Systems

Kenneth J. Goldman

WUCS-92-31

September 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899
Separating Structure from Function in the Specification and Design of Distributed Systems

Kenneth Goldman*
Department of Computer Science
Washington University
St. Louis, MO 63130
kgj@cs.wustl.edu

September 22, 1992

Abstract

A distributed system is viewed as a collection of functional components and a unifying structure that defines relationships among the components. In this paper, we advocate a particular approach to distributed system specification and design in which the structure of a distributed system is specified separately from the functional components. This permits one to reason about individual functional components in isolation, and encourages one to make explicit not only the input/output behavior of the functional components but also the logical placement of these components within the overall structure of the system.

We describe a new software tool for the specification, design, and simulation of distributed systems that supports this separation of structure and function. The tool, called the Spectrum Simulation System, is based on the I/O automaton model of Lynch and Tuttle with extensions for shared memory, superposition, and dynamic process creation. Spectrum consists of a programming language for describing functional components of distributed systems, a configuration mechanism for specifying system structures, and software for simulating the executions of the specified systems. Functional components, described as I/O automata, may be parameterized by the configurations defined separately. This separation gives rise to remarkable versatility in specification and design.

Categories and subject descriptors from Computing Reviews: concurrent programming (D.1.3), requirements/specifications (D.2.1) methodologies and tools, formal definitions and theory (D.3.1), design (D.2.10), program verification (D.2.4), models of computation (F.1.1), simulation languages (I.6.2)

General terms: software engineering, theory of computation

Additional keywords and phrases: configuration, structural specification, software design tools, distributed algorithms, distributed systems, I/O automata,

*Portions of this research were conducted at the Massachusetts Institute of Technology Laboratory for Computer Science and supported in part by the National Science Foundation under Grant CCR-85-11443, by the Office of Naval Research under Contract N00014-85-K-0168, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125, and by an Office of Naval Research graduate fellowship, and portions of this research were conducted at Washington University and supported in part by the National Science Foundation under grant CCR-91-10099.
1 Introduction

We view a distributed system as a collection of functional components and a unifying structure that specifies how those functional components relate to one another. The functional components may be hardware devices, or they may be software processes. Each has a well-defined external interface that defines how it interacts with its environment. The structure of the system defines how many of each kind of component exist in the system and how the components are related. If the functional components are the parts of an automobile, then the unifying structure is a description of how the parts are assembled. We should be able to check each part independently and then reason about the assembly, assuming that each part works properly. This kind of abstraction is powerful, for it gives us the ability to concentrate on different aspects of the problem at a time, rather than having to cope with all the details at once. In fact, it gives us the power to create subassemblies and then, treating each subassembly as an abstract unit, combine the subassemblies into a larger structure.

The structure of a distributed system can be very complex. Functional components may be composed hierarchically into larger structures, components may communicate with one another through messages or shared memory, and some components may even be allowed to see inside of other components. Furthermore, unlike in the automobile analogy, new functional components (the "parts") may be created dynamically in a distributed system as the computation unfolds, and the structure of the system must expand to include the relationships between these new components and those that existed before. From a design standpoint, the ability to separately describe the functional components and the structure of a distributed system has clear advantages: It permits local reasoning about the individual components, it encourages us to state rules about how components may be combined in terms of their external behaviors, it enables us to study the structure of the system while treating the functional components as abstract entities, and it allows us to "plug in" new functional components as replacements for existing ones without changing the structure of the system.

Achieving this separation is not necessarily easy though, especially when processes may be created dynamically and the structure of the system evolves over time. Traditionally, programming languages have blurred the distinction between structure and function. For the most part, languages have concentrated on the definition of functional components and let the structure of the system be defined (usually at run-time) from within those components using system calls provided by the operating system. For example, UNIX processes may, at run-time, create other processes, allocate shared variables, and create sockets for message-passing communication [5]. More recently, some languages have begun to provide constructs for creating multiple processes and communicating among them. Examples include the task construct in ADA [38] and the coenter construct in Argus [30]. Still, the structure of the system is not defined as a separate unit, but is defined as the result of the activities of the processes in the system. A disadvantage of this is that, without a separate description of how the parts fit together, it becomes very difficult to reason about the system as a whole. That is, one must understand what the processes do in order to understand how the system is structured. In these traditional languages, there is no established framework or organization on which one can depend when reasoning about the system at a high level.

Recently, however, there has been growing interest in developing new programming languages that achieve a separation of the structural information from the functional components. Gelernter
and Carriero have identified a class of languages called *coordination languages*, which they describe as the “glue” that binds together the various computational components of a system [12]. They argue that a complete programming language should consist of two pieces: a computation language (for defining the functional components) and coordination language (to bind them together), and they claim that the orthogonality and generality provided by this approach would be a major benefit to software engineering.

Several programming languages have been constructed that embrace the approach of separate computation and coordination languages. For example, Linda [1] is a coordination language that allows multiple processes, possibly written in different computation languages, to communicate through a shared tuple space. Linda provides the constructs in and out that allow processes to remove and insert elements of a shared tuple space in order to communicate and coordinate their activities. This permits one to write a program that communicates with an abstract environment, so that one does not need prior knowledge of the other modules in the system. That is, since each program module interacts with its environment through the tuple space, it need not be concerned with establishing communication with particular other modules; each module finds the information it needs in the tuple space without regard to which other module placed it there. Although this “open” unstructured approach to communication frees the programmer from thinking in terms of senders and receivers, it provides little help in specifying and understanding the logical structure of the system.

A more typical approach to coordination languages, and the approach we adopt here, is to provide separate mechanisms for establishing explicit relationships among program modules. For example, Darwin [26, 34, 28], a generalization of Conic [27, 28], is a configuration language that allows one to manage message-passing connections between the *ports* of various processes in a dynamic system. Each process is expressed in a computation language that is largely independent of the configuration language, except that it provides port declarations that comprise its connection interface for use by Darwin. Durra [6] provides, within a single language, separate mechanisms for describing computation modules, input and output ports, and channel connections between ports for large-grained distributed computation. Similarly, Hermes [45], provides a facility whereby values may be passed across ports that are connected from one module to another. Polylith [40, 41] provides module interconnection constructs for creating bindings among modules in a distributed system. Darwin and Polylith associate a predefined semantics with their connections. In Darwin, it is a message-passing semantics. In Polylith, it can be a remote procedure call or a message-passing semantics. In this work, we take a more general approach in which connections define *logical* relationships that are used to *parameterize* the computation modules. The semantics of our connections are not predefined, but instead depend upon the particular application.

Here, we are interested not only in programming languages, but are more generally interested in tools that support the entire process of specifying, designing and reasoning about distributed systems while maintaining a separation of structure and function. Such design tools may be formal models in which one can specify problems and construct correctness proofs, or they may be software tools that support algorithm design and simulation or verification. In fact, we are interested in both: software tools that rest on a formal foundation. When specification is accomplished in a formal model and design and simulation takes place using software tools based on that model, it is easier to integrate the tasks of algorithm specification, design, debugging, analysis, and proof of
correctness.

A number of formal models have been developed for the study of concurrent systems. Examples include CSP [22], CCS [36], Statecharts [18, 19], UNITY [10], and Swarm [42, 43]. Some of these models have accompanying programming languages or simulation tools. For example, the Occam system [39] provides an implementation of a subset of CSP, the Statemate system [17] provides a simulation tool for systems described with Statecharts, and a programming language has been developed for the Swarm model as well. Another example is the COSPAN system based on its own s/r (selection/resolution) model [20]. However, these tools, and the models they support, typically provide little separation of structure and function. For example, parallel Occam processes are created using a PAR construct. In order to know what processes might be created in a system, one must analyze the program text to see what PAR statements might be executed. This is another example of a programming language that requires one to understand what the processes do in order to understand how the system is structured. At the other extreme, Statecharts are entirely structural. With Statecharts, one specifies a system in terms of a hierarchical finite state diagram, so it is precisely the structure of the system that defines the computation. The notion of structure and function are entirely intertwined. In Swarm, transactions access a global tuple space, so the communication structure is "open," as in the Linda model. In the s/r model of COSPAN, the communication structure is based on modules exporting and importing variables, where in each step of the execution, all modules synchronously read their imported variables and make a state change based on the values observed. Since the the set of processes and their communication structure is static, it is possible to determine the structure of the system in advance, but only by inspecting the headers of each module. The structure is not defined by a separate mechanism.

In this paper, we present the Spectrum Simulation System, a new distributed system specification and design tool that separates the notion of structure and function. As a formal foundation of our work, we use the I/O automaton model of Lynch and Tuttle [32, 33]. Briefly, an I/O automaton consists of a signature (a set of input and output actions), a possibly infinite set of states with a distinguished set of start states, a nondeterministic transition relation, and a partition of the output actions into equivalence classes that define the processes within the automaton that must be given fair chances to take steps.

We choose the I/O automaton model for several reasons. We mention these reasons here and discuss them in more detail in Section 2 and throughout the paper. First, the separation of inputs and outputs, and the fact that input actions are always enabled, make the I/O automaton model convenient for describing distributed systems. Second, the model's shared action semantics is well-suited to a separation of structure and function. That is, communication among I/O automata does not require explicitly naming the other automata; instead, when an automaton produces an output, all the other automata with that action in their signature take a step as well. Therefore, providing the necessary shared action communication requires only that the mechanism for constructing automaton signatures be flexible enough to make use of structural information from the system configuration. Third, the model provides a number of features (some original, and some added later) that allow one to describe a wide variety of systems. These features, discussed in Section 2, include composition, superposition, atomic shared memory, dynamic process creation, and time-constrained automata. Finally, we have chosen the I/O automaton model because it has been used successfully for describing a wide variety of nontrivial distributed algorithms (for examples, see
[2, 3, 4, 11, 16, 21, 31, 47], providing evidence that the model could be quite useful to designers of practical systems.

In spite of the fact that the shared action semantics of the model is well-suited for the separation of structure and function, the same features of the model that provide its expressive power also present significant challenges in constructing a configuration mechanism. The most difficult challenges arise in maintaining orthogonality among the various model features, and specifically in maintaining orthogonality between dynamic process creation and each of the other model features. For example, when a new process is created in the system, we must be able to determine from the configuration whether this new automaton is part of a composition (and which one), whether it shares variables shares with other automata (and which ones), whether it is part of a superposition (and which one), etc. The configuration mechanism must allow all of this structural information to be expressed in a convenient and flexible way, and since dynamic process creation implies there may be an infinite set of processes that may potentially be created during a system execution, the configuration mechanisms must provide a concise notation for defining such information for all of these potential processes.

Our approach to meeting these challenges is to provide a way for a module's interface and behavior to depend upon its place in the configuration. That is, we provide the ability to define general module types whose specific instances make use of contextual information in order to determine communication patterns and coordinate their activities. More specifically, we introduce the notion of a parameterized automaton type that differs from an ordinary I/O automaton in that its signature and transition relation are not fixed, but instead depend upon a set of attributes that are assigned values when an instance of that type is created. These values are assigned by a configuration mechanism that provides for hierarchical composition and attribute inheritance, and provides a pictorial syntax for constructing finite static specifications of infinite dynamic structures.

Our approach generalizes the idea of establishing connections with a predefined semantics among modules with predefined interfaces. Here, the sets of input and output actions of a module are themselves a function of the module's context, its place in the configuration. That is, we define our functional components in a flexible way that allows them to adapt to the context in which they are placed. For example, consider an automaton type that has one input action and two output actions in its signature for each of its neighbors, where each of the output actions is in a separate equivalence class. One instance of that automaton type may have two "neighbors" in the configuration, and as a result may have six actions in its signature and two equivalence classes (processes), whereas another instance of the same automaton type may have three neighbors in the configuration, and as a result may have nine actions in its signature and three equivalence classes. Furthermore, the meaning of each connection in the system structure may vary according to its use within the modules it connects. The structural specification of the system can carry with it semantic information that is dependent upon the particular application domain. Thus, one may specify not only communication channels, but also logical relationships among the components.

The remainder of this paper is organized as follows. In Section 2, we review the formal foundations for this work. This includes the I/O automaton model [32, 33], plus extensions of that model to support atomic shared memory [13], superposition [15], dynamic process creation [29], and time [35]. Then, in Section 3, we present an overview of the Spectrum Simulation System. Section 4 describes the language in which functional components are described as I/O automaton
types in Spectrum, and Section 5 describes the Spectrum configuration mechanism used to define the system structure as instantiations of the I/O automaton types and relationships among them. For both the language and configuration mechanism, we explain how complex relationships among components, such as shared actions and shared variables, are established, and explain how these seemingly static relationships are integrated with dynamic process creation. In the course of our description of Spectrum, we present several small examples. This is followed, in Section 6, by a general discussion of the benefits of the kind of separation we have achieved. We also say a few words about the current implementation and suggest some possible directions for future work. Gel-ernter and Carrier [12] remark that "it would be nice to have a theoretical foundation for general coordination." We conclude, in Section 7, with a discussion of how our work may help to provide such a foundation.

2 Background

In this section, we describe the I/O automaton model of Lynch and Tuttle [32, 33], the formal foundation of our work. There are several important reasons for basing a distributed algorithm development tool on a formal model. Formal models help us to cope with the inherent difficulties in reasoning about distributed systems and allow us to prove safety and liveness properties of our algorithms. Informal reasoning and simulation are useful for discovering errors quickly and easily, and are also useful for gaining insights about algorithms and studying their performance under various conditions, but informal arguments and simulations are inadequate substitutes for careful proofs, since anything short of a complete proof is likely to miss "bad" executions — executions in which the particular choice of process step interleaving leads to the violation of the specification. Therefore, one would like a simulation tool to offer a semantics that is consistent with a formal model so that correctness proofs may be carried out precisely for the algorithm being developed using the tool, so that insights gained from simulations can be used in the correctness proofs, and so that assertions used in the proofs may be checked during simulation. In general, basing the development tool on a formal model makes it possible to integrate distributed algorithm specification, design, simulation, testing and debugging, analysis, and proof of correctness within a single framework.

We choose the I/O automaton model, in particular, because it is a simple model with a crisp semantics that is well-suited for asynchronous distributed systems in which autonomous processes may issue output actions at any time, but must be prepared to handle inputs whenever they occur. In fact, the I/O automaton model differs significantly from CCS [36] and CSP [22] precisely in that input and output actions in the I/O automaton model are distinguished, and an I/O automaton cannot block an input action from occurring. In that sense, I/O automata are similar to Jonsson's I/O-systems [24]. Furthermore, the shared action semantics of the model lends itself to a separation of structure and function. Since automata share actions by name, the structure of the shared action communication can be entirely specified in terms of the signatures.

Another reason for choosing the I/O automaton model as the basis of a design tool is that its features are compatible with software engineering techniques. For example, the semantics of I/O automaton composition provides a formal framework for modular system design and encapsulation. Additional features of the model, such as superposition (for layered systems), atomic shared memory, dynamic process creation, and time-constrained automata enhance the model's expressive
power for describing large distributed systems.

The following brief introduction to the I/O automaton model includes a review of the basic model, as well as a discussion of techniques for modeling shared memory, superposition, dynamic process creation, and time. The details of the model and these techniques may be found elsewhere [33, 13, 15, 29, 35].

2.1 I/O Automata

An I/O automaton is essentially a nondeterministic (possibly infinite-state) automaton with an action labeling each transition. An automaton’s actions are classified as either ‘input’, ‘output’, or ‘internal’. An automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

Formally, an action signature $S$ is a partition of a set $acts(S)$ of actions into three disjoint sets $in(S)$, $out(S)$, and $int(S)$ of input actions, output actions, and internal actions, respectively. We denote by $ext(S) = in(S) \cup out(S)$ the set of external actions. We denote by $local(S) = out(S) \cup int(S)$ the set of locally-controlled actions. An I/O automaton $A$ consists of five components: an action signature $sig(A)$; a set $states(A)$ of states; a nonempty set $start(A) \subseteq states(A)$ of start states; a transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ with the property that for every state $s'$ and input action $\pi$ there is a transition $(s', \pi, s)$ in $steps(A)$; and an equivalence relation $part(A)$ partitioning the set $local(A)$ into a countable number of equivalence classes. Each class of a relation may be thought of as a separate process. We refer to an element $(s', \pi, s)$ of $steps(A)$ as a step of $A$. If $(s', \pi, s)$ is a step of $A$, then $\pi$ is said to be enabled in $s'$. Since every input action is enabled in every state, automata are said to be input-enabled. This means that an automaton is unable to block its input.

An execution of $A$ is a finite sequence $s_0, \pi_1, s_1, \ldots, s_n, s_{n}$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of alternating states and actions of $A$ such that $(s_i, \pi_{i+1}, s_{i+1})$ is a step of $A$ for every $i$, and $s_0 \in start(A)$. The schedule of an execution $\alpha$, denoted $sched(\alpha)$, is the subsequence of $\alpha$ consisting of the actions appearing in $\alpha$. The behavior of an execution or schedule $\alpha$, denoted $beh(\alpha)$, of $A$ is the subsequence of $\alpha$ consisting of external actions. The sets of executions, finite executions, schedules, finite schedules, behaviors, and finite behaviors are denoted $execs(A)$, $finexecs(A)$, $sачeds(A)$, $finschemas(A)$, $behs(A)$, and $finbehs(A)$, respectively. A particular action may occur several times in an execution or a schedule; we refer to a particular occurrence of an action as an event.

2.2 Composition

We can construct an automaton that models a complex system by composing automata that model the simpler system components. When we compose a collection of automata, we identify an output action $\pi$ of one automaton with the input action $\pi$ of each automaton having $\pi$ as an input action. Consequently, when one automaton having $\pi$ as an output action performs $\pi$, all automata having $\pi$ as an action perform $\pi$ simultaneously (automata not having $\pi$ as an action do nothing).

In order to ensure that at most one system component controls the performance of any given action, we require that any collection of automata to be composed must be compatible, meaning that the automata have disjoint sets of locally-controlled actions and that no automaton in the
collection may have as an input action an internal action of another automaton.

The composition $A = \prod_{i \in I} A_i$ of a collection of compatible automata $\{A_i\}_{i \in I}$ has as its signature the composition of the signatures of the components, defined as follows. The output actions of $A$ are all the output actions of the components $\{A_i\}_{i \in I}$, the internal actions are all the internal actions of the components, and the input actions are the remaining actions of the components. Thus, when we compose signatures, an action that is an output of one component and an input of another becomes an output action of the composition. In our proofs, it is sometimes convenient to hide such actions. If $A$ is an automaton and $\Sigma$ is a set of output actions of $A$, then $\text{Hide}_\Sigma(A)$ is the automaton differing from $A$ only in that $\text{out}(\text{Hide}_\Sigma(A)) = \text{out}(A) - \Sigma$ and $\text{int}(\text{Hide}_\Sigma(A)) = \text{int}(A) \cup \Sigma$. We also define the set of equivalence classes of the composition to be the union of all the sets of equivalence classes of the component automata.

Each state of the composed automaton $A$ is a vector of states of the component automata. Given an execution $\alpha = \bar{s}_0 \pi_1 \bar{s}_1 \ldots$ of $A$, let $\alpha|A_i$ (read "$\alpha$ projected on $A_i$") be the sequence obtained by deleting $\pi_i \bar{s}_i$ when $\pi_i \notin \text{acts}(A_i)$ and replacing the remaining $\bar{s}_i$ by $\bar{s}_i[i]$, the $i$th component of the state vector $\bar{s}_i$.

2.3 Fairness

Of all the executions of an I/O automaton, we are primarily interested in the ‘fair’ executions — those that permit each of the automaton’s primitive components (i.e., its classes or processes) to have infinitely many chances to perform output or internal actions. The definition of automaton composition says that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the essential structure of the system’s primitive components. In the model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. A fair execution of an automaton $A$ is defined to be an execution $\alpha$ of $A$ such that the following conditions hold for each class $C$ of $\text{part}(A)$:

1. If $\alpha$ is finite, then no action of $C$ is enabled in the final state of $\alpha$.

2. If $\alpha$ is infinite, then either $\alpha$ contains infinitely many events from $C$, or $\alpha$ contains infinitely many occurrences of states in which no action of $C$ is enabled.

We denote the set of fair executions of $A$ by $\text{faiexecs}(A)$. We say that $\beta$ is a fair behavior of $A$ if $\beta$ is the behavior of a fair execution of $A$, and we denote the set of fair behaviors of $A$ by $\text{fairbehs}(A)$. Similarly, $\beta$ is a fair schedule of $A$ if $\beta$ is the schedule of a fair execution of $A$, and we denote the set of fair schedules of $A$ by $\text{fairscheds}(A)$.

2.4 Shared Memory

The I/O automaton model has been extended to allow modeling of shared memory systems, as well as systems that have both shared memory and shared action communication [13]. One benefit of providing both communication mechanisms in the same model is more flexibility in system design. For example, we might choose to use shared memory for communication among processes running on the same processor and choose to use message passing to model communication between processes running on different processors.
A special class of actions called *shared memory actions* is used to model atomic accesses to shared memory. Each shared memory action contains information that corresponds to the contents of the shared memory before and after the action occurs. Shared memory actions are used by "shared memory automata" to access the shared variables. Each *shared memory automaton* must be prepared to handle any value it may observe in the shared variables, so the preconditions of an output action may not depend on the values of the shared variables.

Since shared memory automata are special cases of I/O automata, all the I/O automaton model definitions (notably composition and fairness) apply to shared memory automata as well. Shared memory automata operate in a system in which the environment is free to change the contents of the shared memory at any time. However, a *closeout* operator \( C \) is provided for taking a shared memory automaton \( A \) and a set of variables \( X \) and producing a new shared memory automaton \( C(A, X) \) in which the given set of variables is made private (absorbed into the local state of the composed automaton). This discussion should be sufficient to understand the foundation of Spectrum's shared variable semantics. For more details on the shared memory extensions, see [13].

### 2.5 Superposition

Superposition, an asymmetric relationship that allows one system component to observe the state of another, appears in a number of models for distributed systems. Bougé and Frances [8] argue in favor of the use of superposition as a language construct by describing a number of important applications for its use, and they offer a compositional approach to superposition with a syntactic representation in CSP. In the UNITY model, defined by Chandy and Misra [10], a program consists of a set of *statements* that access a global shared memory. At each step in the execution, a statement is selected and executed, possibly updating the memory. Superposition in UNITY is defined to be a program transformation that adds a layer on top of a program, while preserving all the properties of the underlying program. Essentially, the transformation modifies the underlying program by augmenting it with a set of new variables and additional statements. In order to preserve the properties of the underlying program, the additional statements must not write to the original variables, although they may read them.

The I/O automaton model has been extended previously to support superposition [15]. We summarize the definitions here. When one module is superposed on another, we want to prevent the higher level module from interfering with the lower level one. That is, we want the higher level module to observe (but not modify) the state of the underlying module. In the I/O automaton model, this amounts to ensuring that in the superposition the higher level module does not place constraints on how the lower level module may modify its own variables. Therefore, we define superposition only when the higher level module is "unconstrained" for the variables of the lower level module. If \( X \) is a set of variables with domain \( \text{dom}(X) \), we say that \( A \) is *unconstrained* for \( X \) iff \( A \) is an I/O automaton such that \( X \) is a subset of the state components of \( A \) and the transition relation of \( A \) places no restrictions on the values of the variables in \( X \) following any action. Note, however, that the set of locally-controlled actions enabled in a given state of \( A \) may depend on the values of the \( X \) variables in that state.

Let \( X \) be a set of variables with domain \( \text{dom}(X) \), and let \( A \) and \( B \) be strongly compatible automata such that \( A \) is unconstrained for \( X \). Then we define the *superposition of \( A \) on \( B \) with respect to \( X \)*, denoted \( C = \mathcal{S}(A, B, X) \), as follows:
• $\text{sig}(C) = \text{sig}(A) \times \text{sig}(B)$ (usual signature composition),

• $\text{states}(C) = \text{states}(A)$,

• $\text{start}(C) = \{(p, x) \in \text{start}(A) : x \in \text{start}(B)\}$,

• $\text{steps}(C) = \{((p', x'), \pi, (p, x)) \text{ such that the following conditions hold:}\$
  
  1. $\pi \in \text{sig}(C)$
  2. if $\pi \in \text{sig}(A)$, then $((p', x'), \pi, (p, x)) \in \text{steps}(A)$
  3. if $\pi \in \text{sig}(B)$, then $(x', \pi, x) \in \text{steps}(B)$
  4. if $\pi \notin \text{sig}(A)$, then $p = p'$
  5. if $\pi \notin \text{sig}(B)$, then $x = x'$, and

• $\text{part}(C) = \text{part}(A) \cup \text{part}(B)$.

Informally, the signature of the superposed automaton $C$ is the composition of the signatures of $A$ and $B$. The states of $C$ are the same as the states of $A$, and the set of start states of $C$ is the set of all start states of $A$ such that the values of $X$ agree with some start state of $B$. The most interesting part of the superposition definition is the construction of the set of steps. It says that any step of $C$ for an action of $A$ must also be a step of $A$. Similarly, any step of $C$ for an action of $B$ must be a step of $B$, when projected on the variables in $X$. Essentially, the actions of $A$ and $B$ are enabled just as before, with automaton $B$ controlling the values of the variables in $X$. The last two conditions of the $\text{steps}(C)$ construction simply prevent steps involving only $B$ from modifying the private variables of $A$, and steps involving only $A$ from modifying the variables in $X$. That is, if a step of $C$ does not involve an action of $A$, then the private state variables of $A$ must not be modified by the step. Similarly, if a step of $C$ does not involve an action of $B$, then the values of the variables in $X$ are unchanged by the step.

In a step for an action shared by $A$ and $B$, the private state of $A$ is modified according to the transition relation of $A$, while the state of $X$ is modified according to the transition relation of $B$. This should agree with one's intuition about the semantics for such shared actions.

Since the superposition of one I/O automaton on another produces a new I/O automaton, all the standard definitions and results for I/O automata (for fairness, compositionality, etc.) immediately carry over to superposed automata. Superposition does not affect the set of executions of the underlying module, thus preserving all properties of that module.

2.6 Dynamic Process Creation

Dynamic process creation is the ability to create new processes during the course of a computation. We adopt the definitions for modelling dynamic process creation in the I/O automaton model presented by Leo [29]. Under these definitions, an I/O automaton system in which automata are created and destroyed can be modeled as an I/O automaton system in which the set of all automata that could potentially be created during any execution of the system are in existence at the beginning. This permits us to specify structural information even for those processes that have not yet been "created". However, only those automata that have been created take steps. That
is, each automaton has a subset of input actions that are designated creation actions. A given automaton does not take any output steps or internal steps until one of its creation actions occurs. Furthermore, before a creation action occurs for a given automaton, all of its other input actions are ignored (do not result in a state change in that component). Once an automaton is created, all subsequent creation actions for that automaton are ignored.

We say that an automaton not yet created is dormant, and that a creation action "wakes up" a dormant automaton, after which we say that the automaton is awake. For each automaton, one may define a predicate on its state that indicates whether or not it is awake. This predicate becomes true once a creation action has occurred. If an automaton is the composition of a collection of automata, that automaton is said to be awake if any of its components is awake. Therefore, a composition may be awake even if some of its components are still dormant.

In the design of the Spectrum Simulation System, support for dynamic process creation has considerable impact on the configuration mechanism. Typically, systems with dynamic process creation are modelled with I/O automata as infinitely many different automata that potentially could be created in a system execution. Which automata are actually created depends upon the input from the environment and the way in which the computation unfolds. This approach allows us to reason about a system knowing the structure of the relationships among the different automata. In order to support this view of dynamic process creation, the Spectrum configuration mechanism provides features supporting finite descriptions of infinite structures. These features, and their relationship to the other aspects of the configuration mechanism, are described in Section 5.

2.7 Time

The definition of an I/O automaton execution is stated in terms of the sequence of events that occurs in a system, and is not concerned with assigning real times to those events. That is, the original model is concerned only with what things happen in the system and in what order they happen. However, in time-sensitive applications, it is important to reason about not only what events will occur in a system, but also when they will occur. For example, it may be important to know that there is a fixed upper bound on the time between a particular input event and a corresponding output event; it may not be enough just to know that the output event will eventually happen.

Because the eventuality properties that may be proven using the original I/O automaton model are not always enough, Merritt, Modugno and Tuttle have defined time-constrained automata, an extension of the I/O automaton model to support reasoning about time [35]. We summarize these extensions here. Interested readers are referred to their paper for details and a discussion of related work.

In the extended model, the central concept is the timed execution, an ordinary I/O automaton execution together with a timing that associates a real time with each state in the execution such that the sequence of times is nondecreasing and only finitely many states are assigned times in any time interval. The time assigned to a state in an execution denotes the real time at which the automaton enters that state. Because only finitely many states may be assigned times in a given interval, composition of time-constrained automata is limited to finite collections of automata.

A timing property is defined to be a predicate on timed executions. These are commonly used to specify the allowable timings for an automaton's executions. A timed automaton, then, is a pair \((A, P)\), where \(A\) is an I/O automaton and \(P\) is a timing property. The timed executions of timed
automaton \((A, P)\) are exactly those timed executions of \(A\) that satisfy the timing property \(P\).

To model the step time of a process, one may define a *boundmap* that associates with each class of an I/O automaton a real time interval that specifies the lower and upper bounds on the step times of that process. More specifically, if a boundmap defines the interval \((l, u)\) for class \(C\), then if an action in class \(C\) becomes enabled at time \(t\), then either an action from \(C\) will occur between time \(t + l\) and \(t + u\), or a state is reached before time \(u\) in which no action from \(C\) is enabled. Using a boundmap, then, it is possible to prove that all timed executions of a given time-constrained automata (that obey the restrictions of the boundmap) satisfy a given timing property.

### 2.8 Problem Specification

An important reason for using a formal model is, of course, the ability to formally specify a problem and then prove that a particular solution solves that problem. In this section, we describe the mechanism by which problems are specified in the I/O automaton model. This mechanism is used as the formal foundation for the "spectator," a device we present in Section 6 that that allows one to check executions of an I/O automaton system against the problem specification.

A 'problem' to be solved by an I/O automaton is formalized as a set of (finite and infinite) sequences of external actions. An automaton is said to *solve* a problem \(P\) provided that its set of fair behaviors is a subset of \(P\) and the automaton is said to *implement* the problem \(P\) if its set of finite behaviors is a subset of \(P\). Although the model does not allow an automaton to block its environment or eliminate undesirable inputs, we can formulate our problems (i.e., correctness conditions) to require that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs.

We want a problem specification to be an interface together with a set of behaviors. We therefore define a *schedule module* \(H\) to consist of two components, an action signature \(\text{sig}(H)\), and a set \(\text{scheds}(H)\) of *schedules*. Each schedule in \(\text{scheds}(H)\) is a finite or infinite sequence of actions of \(H\). Subject to the same restrictions as automata, schedule modules may be composed to form other schedule modules. The resulting signature is defined as for automata, and the set \(\text{scheds}(H)\) is the set of sequences \(\beta\) of actions of \(H\) such that for every module \(H'\) in the composition, \(\beta[H']\) is a schedule of \(H'\). Notice that every I/O automaton can be thought of as a problem specification, where the interface is the external signature of the automaton and the set of behaviors is the set of fair behaviors of the automaton.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors. Informally, a module *preserves* a property \(P\) iff the module is not the first to violate \(P\); as long as the environment only provides inputs such that the cumulative behavior satisfies \(P\), the module will only perform outputs such that the cumulative behavior satisfies \(P\). A formal definition and a proof that a composition preserves a property if each of the component automata preserves the property are given elsewhere [33].

### 3 Design Goals

The main subject of this paper is the Spectrum Simulation System, a new research tool for the design and study of distributed algorithms. In this section, we discuss the design goals and their
implications, and then provide a brief overview of the design. Spectrum was designed with the following primary goals in mind.

1. Integrate the process of distributed algorithm specification, design, simulation, testing and debugging, analysis, and proof of correctness within a single formal framework.

2. Support the design of a wide variety of distributed algorithms.

3. Facilitate insightful study of distributed algorithms through experimentation and exploration.

The first goal implies that the simulation language and its semantics (as well as the implementation of the simulator) must be faithful to the formal model. Any departure from the formal model would jeopardize effective integration of the two. For example, it is only possible to mechanically check executions of an algorithm against properties stated in the proof if the semantics of the simulation match the semantics of the model. By remaining faithful to a formal model, we also benefit from a well-defined semantics on which to base the language and implementation.

The second goal implies that the formal model must have sufficient expressive power. In fact, since the initial implementation of Spectrum [14], we have redesigned the system and added several features to the original I/O automaton model, including shared memory [13] and superposition [15], as well as incorporated into the design of Spectrum extensions to the model developed by others for dynamic process creation [29] and time [35].

The third goal, of course, implies that language features such as static type checking and a rich collection of built-in data types are needed in order to ensure that users spend their effort on experimenting with algorithms rather than finding obscure program errors. Support for hierarchical encapsulation is required so that algorithm components may be studied individually or replaced with other components, and so that simulations can be studied at different levels of detail. Also, supporting the construction of user-defined debugging and analysis devices is important. Ideally, these devices should be defined as separate modules not only to avoid the confusion and clutter of cluttering embedded debugging statements, but also to ensure that the debugging and analysis devices do not interfere with the algorithm's execution. The need for experimentation also requires special support for controlling and studying the simulation, such as automatic detection of invariant violations, a choice of scheduling options, simple graphical mechanisms for configuring systems, and support for constructing visualizations of executions.

Most importantly, however, the third goal implies that the design must be based upon a separation of concerns so that the algorithm being studied (the functional components) and the system configuration (relationships among those components) may each be modified independently. For example, making a simple modification of the configuration, such as adding a new functional component to the system or changing the arrangement of the functional components, should be possible without the need to modify to the program describing the functional components themselves.

Having discussed the overall design goals and motivation for the Spectrum Simulation System, we now turn to a description of the system itself. Spectrum consists of a programming language for specifying functional components of a system, a configuration mechanism for specifying the relationships among those components, and a simulator for generating system executions. Spectrum users express distributed algorithms as collections of I/O automata and then simulate them directly, in a way that is faithful to the semantics of the formal model.
We will discuss the details of the programming language and configuration mechanism separately, but we begin with a high-level description of the relationship between the two. In Gelernter and Carriero's terms [12], the programming language may be considered the "computation language" and the configuration mechanism the "coordination language." The Spectrum programming language is used to express I/O automaton types, the building blocks of I/O automaton systems. An automaton type defines the signature, states, transition relation, and action partition of potentially many different automata. Having defined a collection of automaton types in the language, one separately supplies a configuration that specifies a set of automaton instances and defines relationships between them. For each instance, the configuration specifies the automaton type of that instance and specifies values for its attributes. The attributes may be referenced in the automaton type to determine the signature and transition relation of that instance. In other words, the automaton types are parameterized by the configuration data. In this way, the configuration breaks symmetry among automaton instances of the same automaton type.

The configuration is represented as a graph, where nodes represent automaton instances and edges represent relationships among them. For example, in a configuration of several automata arranged in a ring, one might use directed edges to specify which of the automata are neighbors. Alternatively, edges might be used to represent parent-child relationships in a nested transaction system.

In addition to defining instances, a configuration may define automaton types that are formed by composing collections of other automaton types under the usual rules of I/O automaton composition. Instances of these composed types may have attributes that are inherited by their components. To capture this composition hierarchy, the configuration graph is itself hierarchical. Each node representing an instance of a composed type contains, in turn, another graph that specifies the component automata of that composition and the relationships among them.

It may be helpful to think of an automaton type as a program and an automaton instance as a single invocation of that program. Each instance of a given automaton type has the same program, but that program may reference information present in the configuration. Thus, two instances of the same automaton type may have different initial states, signatures, transition relations, and partitions. This separation of structure from function pervades the design of the Spectrum Simulation System and helps us to achieve our design goals by permitting experimentation with algorithms by varying the system configuration independently.

We now turn to the details of the programming language (Section 4) and the configuration mechanism (Section 5), including the provisions in each for shared memory, superposition, dynamic process creation, and time-constrained automata.

4 Specifying Functional Components

The Spectrum programming language is the first executable language based on the I/O automaton model. In the literature, the transition relations of I/O automata typically have been described using variations on the "precondition/effect" notation of Lynch and Tuttle [33] based on Dijkstra's guarded commands. In this notation, each action has a precondition that maps each state of the automaton to a boolean value, and the action is enabled in exactly those states in which the precondition is true. (Since input actions are always enabled, their preconditions are taken
to be true in all states.) Similarly, each action has an effect that defines the new state of the automaton based on the action and the state from which the action occurs. However, the notations used to express these preconditions and effects have, until now, been rather ad hoc, and authors usually have resorted to prose to define the data types and initial values of state components, as well as the partition of locally controlled actions. Spectrum provides well-defined constructs for expressing each of the five basic components of an I/O automaton: the signature, states, initial states, nondeterministic transition relation, and partition of locally-controlled actions.

**Automaton types:** Functional components are specified in the Spectrum programming language as automaton types. Automaton types are not automata in that their signatures and behaviors are not fully defined until the automaton type is instantiated and its attributes are assigned values. An automaton may be instantiated as part of the configuration process, or it may be instantiated by a creation action at run-time, but regardless of how an automaton is instantiated, the attribute information is treated the same way. So, in defining an automaton type, it is not necessary to specify whether instances of that type will be created dynamically at run-time or statically as part of the configuration process. We now explain how automaton types are defined and discuss how attribute values are used once an automaton is instantiated. The actual mechanisms by which automata are instantiated and attributes values are assigned are described with the configuration mechanism in Section 5.

Automaton types are defined in Spectrum using a straightforward syntax that we illustrate by example. A partial grammar for the language is contained in Appendix A. Every automaton type, such as the channel automaton type shown in Figure 1, has a type name assigned by the programmer. Using a syntax reminiscent of parameters to a procedure, each automaton type may have a set of named attributes. For example, the automaton type channel in Figure 1 is defined to have two attributes: id and faulty. Attributes are generally immutable. That is, their values are fixed by the configuration. However, for automaton types whose instances may be created dynamically, we allow attributes to be declared as mutable, which allows them to be updated when the automaton is created.

**Data types:** All attributes and variables are assigned types. Built-in base types include automaton_id, integer, real, boolean and string, and aggregate types include tuple, set, multiset, array, sequence, and graph. The expected operations are provided for each type. Infix notation is provided for commonly used operations, and syntactic sugar is provided for constructing values for aggregate types. Constructed types may be arbitrarily nested and assigned mnemonic names. For example, in Figure 1, the type name buffer is defined to be a set of messages, where a message is defined to be a tuple consisting of an automaton id and a string. Recursive type definitions are not permitted. The current implementation of the Spectrum parser provides static type-checking, based on structural equality.

**State components:** The variables that represent the state components of an automaton are declared at the beginning of the automaton type definition. To prevent the use of undefined variables, each data type has a predefined default value. However, an INIT construct is provided whereby the programmer may define the initial state of an automaton by assigning values to the
TYPE message tuple(chan: automaton_id, text:string) % type declarations
TYPE buffer set(message) %

AUTOMATON channel(id: automaton_id, faulty: boolean) % name and attributes

STATE pending: buffer
  up: boolean % state components

INIT pending := { }
  up := TRUE % initialization

INPUT send(msg: message) WHERE msg.chan = id % a 'send' input
  EFF set_insert(pending, msg) % buffers the message

CLASS

OUTPUT receive(msg: message) % a 'receive' output
  PRE pending != { }
    up = TRUE % nonempty buffer
  SEL msg := set_random(pending) % and currently up
  EFF set_delete(pending, msg) % select a message

INTERNAL fail(chan: automaton_id) % an internal action
  PRE faulty = TRUE
    up = TRUE % channel is faulty
  SEL chan := id % and currently up
  EFF up := FALSE % argument is own id

  % channel fails

Figure 1: A Spectrum automaton type definition.
state components, possibly on the basis of the automaton's attribute values. For example, the state components of the channel automaton type defined in Figure 1 are a set pending that contains the set of messages waiting to be delivered by the channel and a boolean up that indicates the working/non-working status of the channel. The set pending is initially empty and the variable up is initially true. When appropriate, the attribute values may be referenced (but not modified) as part of state component initialization.

An automaton's state components are normally modified explicitly as part of an action of the automaton. However, an additional construct, called the MAINTAIN clause, is provided for updating the state state components after every action of an automaton. The MAINTAIN clause is somewhat similar to the ALWAYS construct of UNITY [10] and the MAINTAIN construct of IC* [9], except that their constructs are more powerful in that they denote constraints on the state. In fact, in IC* the entire execution is determined exclusively on the basis of these static invariants, plus differential invariants (that relate one system state to the next) and conditional invariants (that may be enforced only in certain portions of the execution). Our MAINTAIN clause is simply a sequence of statements that is used to update local variables after each action. Typically, a MAINTAIN clause is used to maintain "pseudo-variables," variables that are a function of the remaining state components. For example, if the channel had, as an additional state component, the integer variable num_pending, we might use the clause

```
MAINTAIN
    num_pending := set_size(pending)
```

to update the value of num pending equal to the size of the pending set after each action.

Shared memory: In Section 2.4, we described extensions to the I/O automaton model that permit automata to communicate through atomic accesses to shared variables. The atomic accesses are modelled as output actions in which the automaton may read and/or update the shared variables. In the Spectrum programming language, this is supported with declarations of shared variables that may be read and/or written in the effects of output actions. Since an automaton must be prepared to handle any value it may observe in a shared variable, shared variable references are not permitted in preconditions.

Strictly speaking, the programming language does not specify which automata share variables. Shared variable declarations in an automaton type simply identify variables that may be modified by the environment of an instance of that type. It is the configuration that determines which automaton instances actually share variables. This is specified with a closeout mechanism, described in Section 5.

Superposition: To support superposition, as defined in Section 2, the Spectrum programming language provides a simple construct for declaring certain state components as "unconstrained." Unconstrained variables are like other state components, except that they may not be updated by the automaton. In a system configuration, an automaton $A$ may be superposed on another automaton $B$ only if the types of the state components of $B$ match the types of the unconstrained variables of $A$. In the superposition, whenever $B$ changes the values of its state components, the changes are visible to $A$ in its unconstrained variables.
**Invariants:** To assist with verification, the Spectrum programming language permits the declaration of certain predicates as “invariant,” meaning that they should hold true in every state of the execution. The simulator does not guarantee that these hold, but checks, after every step of the automaton, that the conditions hold and issues a warning if an invariant violation occurs. The predicates in an invariant may reference the automaton state, as well as the attributes, unconstrained variables, and shared variables.

**Signature:** The signature of an automaton type is defined in terms of the attributes of that type. As in the model, each action of an automaton is designated as an INPUT, OUTPUT, or INTERNAL action. For example, the automaton type channel in Figure 1, has send actions, receive actions, and fail actions that are input actions, output actions, and internal actions respectively.

A list of named arguments may be declared for each action name. Therefore, a given action name may correspond to many different actions, one for each possible assignment to the action’s arguments. However, we often want to include only a subset of those actions in the signature of an automaton. That is, we may want to include a particular action in the signature only if its arguments have certain values. Spectrum’s WHERE clause is defined for this purpose. For example, in the channel automaton type of Figure 1, we do not want all send actions to be in the signature of every instance. Instead, for each instance of that type, we want in the signature only those send actions for which the chan argument is the automaton id of the instance. The WHERE clause in the example accomplishes this by specifying a predicate on the arguments of that action in terms of the id attribute of the automaton type. A given send action, then, is in the signature of an instance of the channel automaton type if and only if the predicate is true.

An implicit argument, owner of type automaton id is associated with every action in the signature and has as its value the automaton id of the automaton instance producing the event as an output action.

**Dynamic process creation:** Recall, from Section 2, that we model dynamic process creation in terms of activating or awakening a process that already exists. That is, a configuration may contain a dormant I/O automaton that becomes awake when a particular kind of input action, called a creation occurs at that automaton.

To support this view of dynamic process creation, the Spectrum programming language permits certain input actions to be designated CREATION actions. In accordance with the model, a dormant automaton takes no steps until the first time a creation action occurs; then the automaton becomes awake and subsequent creation actions are ignored at that automaton. If any of an automaton’s attributes are declared as mutable, then those may be assigned values as part of the creation action, usually on the basis of the action arguments. To prevent inconsistencies, only immutable may be referenced in the WHERE clauses of creation actions.

Once the set of creation actions is specified for an automaton type, the actual instantiation of automata, whether dormant or awake, is left to the configuration. In other words, we maintain the separation of concerns, and specify only the functional components in the programming language and leave the structure of the system to the configuration. Since the set of automata that may be created in a system is potentially infinite, the configuration mechanisms must enable one to define a finite description of an infinite structure. These mechanisms are described in Section 5.
Partition: Recall that fairness in the I/O automaton model is defined in terms of a partition of the locally controlled actions of an automaton. In any fair execution, each class of locally controlled actions in the partition must be given a chance to take a step infinitely often. In other words, if a class continually contains an enabled action, then eventually an action from that class must occur. In Spectrum, the partition is specified by grouping the locally controlled actions into classes using the CLASS construct. All the actions listed under a given CLASS heading belong to the same class in the partition. For example, in Figure 1 the receive output actions and the internal action fail are in the same equivalence class of the partition.

Occasionally, we do not wish to include all the actions with a given name in the same class. That is, we may want two actions with the same name to appear in a different class, depending on the values of the action arguments. To accomplish this, we allow a class block to be parameterized by the attribute values of the automaton. The parameters may be referenced in describing the actions of that class, and their associated precondition, selection, and effect clauses. A domain of values for each parameter is specified in terms of a finite set. At run-time, a class is created for each combination of parameter values. In other words, each element of the Cartesian product of the parameter domains is used to define a class in which the parameters take on the corresponding values. For example,

```
CLASS (number:{1,2,3}, letter:{"a", "b"})
  OUTPUT announce(j:integer, z:string)
    PRE true
    SEL j := number
       z := letter
    EFF announced := announced + 1
```

would result in the creation of six classes, one for each of the possible different combinations of values for the integer parameter number and the string parameter letter. Each class, then, would be responsible for the output action that would "announce" that combination. Thus, in every fair execution, each combination would be announced infinitely often.

The domain for a class parameter usually is not specified as a constant, as in the previous example, but rather is expressed in terms of the attributes of the automaton. For example, a typical domain for a class parameter is the set automaton.ids for the adjacent automata in the configuration graph.

Time: Recall, from Section 2, that the process step times in time-constrained automata are modelled by defining a boundmap that associates a real-time interval with each class of an automaton. This interval defines a lower and upper bound on the step time for that process.

In keeping with our separation of structure and function, it is important that one be able to define an algorithm independent of the speed of the particular processor on which it will run. Therefore, it is important that the boundmap be derived from the attributes of the automaton instances, as defined in the configuration.

To support this separation, we allow the header of a class block to contain an optional construct STEP TIME(1, u), where 1 and u are real number expressions that define the lower and upper bounds on the step time for that class. These values may be defined as constants, but typically will be based
on the values of attributes of the automaton instance. Thus, we may define in the configuration a collection of automaton instances, each running the same algorithm but at different speeds.

**Transition relation:** We conclude our presentation of the Spectrum programming language with a description of how the transition relations for I/O automata are expressed. The language constructs that Spectrum provides for defining transition relations are similar to the "precondition-effect" notion of Lynch and Tuttle. However, there is one important difference. In the literature, I/O automaton descriptions traditionally include action arguments as part of the action name, allowing a given automaton to have infinitely many actions. To specify which of these actions are enabled, one provides preconditions that may refer to the action arguments. This is rather impractical for a real programming language, however, since it might require considering all possible values of the action arguments in order to determine which actions of an automaton are enabled. Since the data types of action arguments may have very large (or infinite) domains, treating the argument as part of the action name and allowing separate preconditions for each would be computationally infeasible. We cannot, in general, evaluate the preconditions for infinitely many different actions in finite time.

We solve this problem in the Spectrum programming language by defining our transition relations as follows. With each locally controlled action of the automaton, we associate a **precondition clause** (PRE), a **selection clause** (SEL), and an **effect clause** (EFF). Since input actions of I/O automata are always enabled, these have only effect clauses. To make programming easier and enhance readability, the clauses for each action type immediately follow the corresponding entry in the signature, as shown in Figure 1.

The reason for using three clauses (precondition, selection, and effect) instead of the usual two (precondition and effect) is that it allows us to express the preconditions in a way that is independent of the action arguments. In a sense, Spectrum splits the traditional precondition into the precondition and selection clauses. A precondition is a boolean expression (or conjunction of boolean expressions) that depends upon the state variables of the automaton, and possibly also the automaton attributes and class variables, but does not depend upon or assign to the action arguments. The action arguments are not bound during precondition evaluation. If the precondition of an action type evaluates to true in a given state, then an output action of that action type is said to be enabled in that state. The precondition clause determines if there exists any assignment to the arguments of the action that would result in an enabled action, and it is the selection clause that selects the particular argument values for an enabled action. That is, the purpose of the precondition is to answer the following question: "Is there some assignment to the arguments of the action type that would give rise to an enabled action in the current state?" If we think of each action type as a set of actions, the precondition in our language determines whether or not this set of actions contains at least one action that is enabled in the current state. Given that the precondition is satisfied, the selection clause is used to determine (possibly at random) the particular values that are assigned to the arguments of the action.

A selection is a sequence of assignments to the arguments of the action. The assignments are executed sequentially and may reference (but not modify) state and/or configuration data. In addition, once an argument has been assigned a value, it may be referenced in later assignments within the selection clause. (In the current implementation, one must assign values to all arguments,
<table>
<thead>
<tr>
<th>clause type</th>
<th>Attribute</th>
<th>State</th>
<th>Shared</th>
<th>Unconstrained</th>
<th>Class</th>
<th>Action</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>R</td>
<td>R/W</td>
<td>—</td>
<td>—</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>MAINTAIN</td>
<td>R</td>
<td>R/W</td>
<td>—</td>
<td>R</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>INVARIANT</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>INPUT</td>
<td>WHERE</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>n/a</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>EFF</td>
<td>R</td>
<td>R/W</td>
<td>—</td>
<td>R</td>
<td>n/a</td>
<td>R</td>
</tr>
<tr>
<td>CREATION</td>
<td>WHERE</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>n/a</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>EFF</td>
<td>R/W</td>
<td>R/W</td>
<td>—</td>
<td>n/a</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>WHERE</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>PRE</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SEL</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R/W</td>
</tr>
<tr>
<td>EFF</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>WHERE</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>PRE</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SEL</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R/W</td>
</tr>
<tr>
<td>EFF</td>
<td>R</td>
<td>R/W</td>
<td>—</td>
<td>R</td>
<td>R</td>
<td>—</td>
<td>R</td>
</tr>
</tbody>
</table>

R = read access, W = write access, n/a = not applicable

Figure 2: Variable access restrictions.

even if a WHERE clause on the output action restricts some arguments to a single possible value; after selection, the system simply checks that the WHERE clause is satisfied by the arguments selected.

An effect clause is used to derive the new state of the automaton from the old state, based on the current state and the action arguments. It consists of a sequence of statements that update the state of the automaton. Again, the statements are executed sequentially and effects of earlier updates are observed by later ones.

As an illustration, consider the transition relation for the automaton type of Figure 1. The output action receive in the channel automaton is enabled if any message is in the set pending. If a receive action is scheduled, then the selection clause is executed to choose a particular message from the set to be the argument of the action. Separating the argument selection from the precondition in this way avoids impractical situation of needing to bind candidate values to action arguments in the precondition. Only one precondition evaluation is needed for each action name, rather than one evaluation for each possible assignment to the arguments. In addition, argument values need not be assigned when the precondition is tested, but only when that particular action type is chosen for the next step of the execution.

To enhance readability, Spectrum programs may contain definitions of user-defined functions to be called from within precondition, selection, and effect clauses. These functions may have a mixture of mutable and immutable arguments, and may optionally return a value. Conditional and iteration constructs are provided for defining precondition, selection, and effect clauses, as well as for use in user-defined functions.
Enforcement: In order to minimize errors at run-time, the Spectrum language is designed to support extensive compile-time checking. In addition to syntax and type compatibility checks, the compiler can also enforce a variety of usage restrictions that arise due to the semantics of the programming language and the I/O automaton model. For example, the compiler can check that the state variables of an automaton are not modified within a precondition, and that shared variables are referenced only in the effects of output actions. These variable usage restrictions are summarized in Figure 2. Each table entry indicates how a particular kind of variable may be accessed (read or write, or both) within a given clause type. A dash indicates that no access is permitted, whereas a “not applicable” entry indicates that no such variables are defined within the scope of the given clause type. Note that to prevent an inconsistent view of the configuration data, only mutable attributes may be written by the EFF clauses of creation actions, and only immutable attributes may be referenced in the WHERE clauses of creation actions. All of these restrictions, as well as type compatibility, are enforced by the current implementation of the Spectrum compiler.

5 Specifying System Structures

Having described Spectrum’s “computation language” for defining the functional components (I/O automaton types) of a distributed system, we now turn to Spectrum’s “coordination language” for defining the structure (configuration) of a distributed system. In other words, having presented the mechanisms for specifying the parts, we now describe the mechanisms for specifying the assembly.

Recall that a configuration specifies a set of automaton instances and the relationships between them. Each functional component in an I/O automaton system is an instance of an automaton type, but a set of automaton types is not a complete specification of a distributed system. Each automaton type is parameterized by a set of attributes that determine how it interacts with the other automata in the system. It is the configuration that specifies not only the set of automaton instances, but also these attributes that determine the structural organization of the system.

Configuration information is specified separately from the functional components, but the mechanisms themselves are greatly affected by the semantics of the programming language and the I/O automaton model on which it is based. Obviously, the configuration mechanisms must be able to support all the features of the programming language in a way that is faithful to the model. For example, the programming language provides support for shared actions, shared memory, superposition, and dynamic automaton creation. That is, the language provides a mechanism for specifying signatures (parameterized by attributes), for declaring variables that potentially may be shared with other automata, for declaring unconstrained variables that may represent state information of another automaton, and for defining creation actions by which an automaton may be awakened. However, it is the configuration that assigns values to the attributes that parameterize the automaton types, defines which automata share which variables, specifies which automaton instances are superposed on others, and specifies the set of automata that may be awakened during the course of the computation and the relationships among them. In general, for each method of communication supported by the programming language, the configuration mechanisms must be able to specify which functional components communicate with each other by that method.

The configuration mechanism, in addition, must support certain operators of the model, such as I/O automaton composition. Furthermore, the configuration mechanism must ensure that the
structural information is specified in a way that is faithful to the I/O automaton model. For example, a collection of automata may be composed only if their signatures are strongly compatible, and one automaton may be superposed on a second only if the types of the unconstrained variables of the first match the types of the state components of the second.

Integrating the different kinds of structural information so that they can be expressed naturally, easily, and with an economy of mechanism is perhaps the most interesting part of the design of a configuration language. In the case of specifying the structure of I/O automaton systems, or distributed systems in general, the most challenging aspect of the configuration mechanism is the support for dynamic process creation. When a system consists of a static collection of components, it is not difficult to imagine straightforward mechanisms for describing this collection and for specifying the communication structure. However, when one needs to specify the structural relationships among an infinite collection of processes that may potentially be created, the mechanisms are not so obvious.

As an example of the interactions that may occur between dynamic process creation and other aspects of the structural specification, consider a system that contains two collections of automata, each collection sharing a different variable called \( x \). Now, suppose that a new automaton is created in the system and that the new automaton has declared a shared variable \( x \). Does the new automaton share the \( x \) variable from the first collection or the second collection, or does it access a completely new variable \( x \)? Similar questions arise for shared actions and for superposition, and in each case the answer depends on the intended structure of the system. A configuration language must provide a natural set of mechanisms for specifying structural relationships among all components that potentially may be created in a system, not just those that are in existence at the beginning of the execution.

Perhaps the most interesting aspect of the configuration mechanism is not the particular choice of operators, but the way in which they are integrated and unified. Attribute inheritance, composition, and the shared memory closeout operator are all handled as part of the same hierarchical configuration structure. The superposition operator is handled with a slightly different structure, but results in a unit that fits within the hierarchy for the other operators. Dynamic process creation merges neatly with all of the above constructs and operators by making use of a device that serves as a placeholder for automata that may be created within the hierarchical structure. Finally, time constraints are handled within the existing attribute mechanism without any need for additional specialized support. An appreciation of the way in which all of the constructs and operators are integrated is best gained by understanding the details of the configuration language, which we now present.

Spectrum's configuration language consists of a set of simple, yet very general, mechanisms for specifying the configuration of a distributed system. In this configuration language, the mechanisms for describing the communication structure of the system (including shared actions, shared memory, and superposition) are integrated with both the mechanisms for describing the hierarchical composition of system components and the mechanisms for describing the relationships among components that are created dynamically. We provide a pictorial syntax for specifying configurations, and illustrate the ideas with several examples.
The configuration graph: We describe the configuration of a distributed system as a hierarchical graph, in which vertices denote logical components of the system and edges denote relationships among these components. The graph is hierarchical in that functional components may themselves have an internal structure. That is, a given vertex in the configuration graph may, in fact, represent another entire configuration of subcomponents. The nodes and edges of the configuration graph are labeled. Each node of the graph is labeled with information (attribute values) to be associated with the corresponding component, and each edge of the graph is labeled with information about the type of relationship represented by that edge. In order to make graph connectivity information available to the functional components, the set of labeled edges incident to a node is made available as an attribute of that node.

Each vertex in the configuration graph has a number of standard attributes in addition to the optional user-defined attributes. Included among the standard attributes are a unique identifier (the automaton id) and the type of the instance represented by that vertex. If a vertex represents an instance of an automaton type defined in the Spectrum programming language, then we say that the vertex represents an instance of a base type.

Each edge in the configuration graph has an associated data type that defines the domain of values that may be associated with that edge. These associated values qualify the relationship between the two automata connected by that edge. For example, an edge may have an associated integer that represents the cost of communication between the two incident automata. In addition to the edges themselves, the values associated with the edges may be used to parameterize automaton types. That is, the values associated with each edge incident to an automaton can be used to determine the signature, classes, and transition relation of that automaton. For the sake of generality, the semantics of an edge depends entirely on how it is used as an attribute within automaton type definitions. Edges are simply treated as an attribute of an automaton type. There is not a fixed interpretation for each edge type. Rather, configurations are constructed with an understanding of how each edge type will be interpreted by the automaton instances incident to that edge. Edges may represent communication links, parent-child relationships, or any other kind of relationship. Edges may be directed or undirected (bidirectional), depending on the intended semantics.

Configuration edges are made available in the definition of automaton types with the use of a special data type, called config_edge. Configuration edges, or more typically sets of configuration edges, may be included in the attribute list of an automaton type. When the configuration is defined, the values of these attributes for a given instance are determined by the edges in the configuration graph incident to that instance.

A pictorial syntax: We describe configurations with a pictorial syntax that lends itself to building configurations with an interactive graphical interface. We believe that the pictorial representation is important because it permits the system designer to take predefined “parts” (the automaton types) and “plug them together” in a way that makes the structure immediately visible and easily understood. We describe our pictorial syntax for system configurations informally using examples. We describe a syntax for a set of operators with well-defined semantics. However, just as we commonly provide a formal syntax for a textual programming language in the form of a grammar, it probably would be useful to have a formal description of our pictorial syntax. We have not yet
attempted this, but graph grammars [37, 44] appear to be a promising approach.

In our pictorial syntax, we represent the vertices of the configuration graph as icons and the directed edges of the graph as lines with arrow heads indicating direction. The shape of a vertex indicates its type. Each base type (an I/O automaton type defined in the Spectrum programming language) has a particular shape, so it is easy to tell the type of each instance from looking at the configuration graph. For each instance of a base type, we create a port for each set of configuration edges that appears as an attribute of that base type. In the pictorial syntax, these ports are shown as "nubs" on the icons representing the automata. Edges, then, are specified by drawing lines that connect the ports of two different automata. Other attribute values for each instance are displayed in a table next to that instance\(^1\). For example, Figure 3 shows two instances of a process automaton type (represented by squares) connected to an instance of the channel automaton type (represented as a circle) that was described in Section 4. The channel automaton type has an attribute `faulty` that is assigned the value `true` in this instance. A partial definition of the process automaton type is shown in Figure 4, where configuration edge information is used to parameterize the signature, transition relation, and classes of the automaton. The process receives messages from channels connected to it by incoming configuration edges and sends messages to channels connected to it by outgoing configuration edges. Making use of the parameterized class construct, messages for each outgoing edge are handled by a separate class to ensure fairness.

The hierarchical structure of the graph is represented in our pictorial syntax by allowing a vertex in the graph to denote an instance of a sub-configuration. That is, one may allow configurations to appear as components of other configurations. Like base types, instances of sub-configurations are identified by shape in the pictorial representation. Each configuration, other than the top level configuration, has its own associated shape and may be instantiated multiple times. For example, in Figure 5, the top level configuration is as in Figure 3, except that an entire subconfiguration (the octagon configuration) appears on each end of the channel automaton. Each instance of the octagon automaton type represents a collection of instances of triangle configurations, and each triangle configuration contains instances of several base types. Ordinarily, recursive instantiations

\(^1\)For aesthetic reasons, an interactive configuration system might not display the attribute tables at all times, but instead treat them like "pop up" menus.
TYPE message tuple(chan: automaton_id, text:string)

AUTOMATON process(id: automaton_id, 
    neighbors:set(config_edge(integer))) % unique id
    % logical connections

STATE pending: mapping(automaton_id, set(string)) % a message set for
    % each neighbor

INIT map_init(pending, { }) % empty set is default

INPUT receive(msg: message) % receive messages
    WHERE set_element(neighbors, <msg.chan, id>) % from incoming channels
    EFF ... % handle the message

CLASS (x: out(config_edge))
OUTPUT send(msg: message) % send messages
    PRE map_eval(pending, x) != { } % on outgoing channels
    SEL msg := <x, set_random(map_eval(pending, x))> % select a message
    EFF set_delete(map_eval(pending, x), msg.text) % remove from buffer

Figure 4: Using configuration edges as attributes.

are not allowed. For example, the triangle configuration must not contain itself, and the octagon and triangle configurations must not contain each other. However, an exception to this rule is made for dynamic process creation, which we explain later. Each configuration is said to be the parent of its sub-configurations in the composition hierarchy. We say that a configuration \( A \) is an ancestor of configuration \( B \) if either \( A = B \) or there exist a sequence of configurations \( c_1, c_2, \ldots, c_i, c_n \) such that \( c_1 = A, c_n = B \), and each \( c_j \) is the parent of \( c_{j+1} \). For each \( c_j \) and \( c_{j+1} \) in this chain we say that \( c_{j+1} \) is a deeper ancestor of \( B \) than \( c_j \).

Composition: Any configuration, whether the top-level configuration or a sub-configuration, may be designated a composition, provided that it meets certain consistency requirements such as type-compatibility for shared variables. Since each instance’s unique automaton id is associated with each of its locally-controlled actions, we are guaranteed that the signatures are strongly compatible. Unlike ordinary configurations, a number of operators may be applied to configurations that have been designated as compositions. These operators, to be explained later, include a close-out operator for shared variables and a superposition operator for constructing layered systems. In our pictorial syntax, compositions are distinguished from ordinary configurations with a thick (or double) border. For example, each instance of the triangle configuration in Figure 5 is defined to be a composition of two squares and a circle.

We are currently investigating additional operators on compositions, such as hiding and renaming. In the specification of a large system, it would be convenient to allow a given action name to be used for different purposes in different parts of a system. However, one needs support
Figure 5: A hierarchical configuration
for encapsulating those names in order to avoid a "name clash," where the two uses of the same name come into conflict at some level in the composition hierarchy. A hiding operator would allow actions that are shared by components of a composition to be treated as internal actions of that composition. A renaming operator, which could also be applied to instances of base types, would allow the assignment of different names to the external actions of a composition. We expect that the abstraction and encapsulation provided by hiding and renaming operators would be very useful for the specifying large systems.

**Attribute values and inheritance:** An attribute consists of an attribute name and an associated data type, the attribute type. Each instance of a base automaton type has a set of attributes that is defined in its automaton type definition, but the attribute values are defined by the configuration. For example, the channel automaton of Figure 1 has an attribute called faulty that is either true or false. When a channel is instantiated in the configuration, one may explicitly define a value for the faulty attribute at that automaton in order to indicate whether or not the channel is faulty. As another example, an automaton type might have as one of its attributes a set of configuration edges of a particular type. The contents of this set, for a given instance of that automaton type, would be defined by the edges of the configuration graph that are incident to the vertex representing that automaton instance.

For convenience in specifying complex system structures, it is useful to have collections of automata that agree on certain attribute values. These common attribute values could be used to parameterize the signatures of the automaton types in order to define shared actions among the collection, or the common attributes might be used to specify some other relationship among the automata. Consider three collections of automata, each collection consisting of automata responsible for accessing a different distributed database. We would like the name of the database associated with each collection to appear as an attribute of each of its automata so that the appropriate database access requests will be recognized by those automata. However, it would be rather inconvenient to explicitly define the value of the database name attribute at each automaton in each collection. Therefore, we allow attributes values to be associated with configurations and provide a mechanism by which these values can be inherited by configuration components.

To support attribute value inheritance, each Spectrum configuration has a set of attributes that is the union of the sets of attributes of its components. Each of these attributes may be assigned a value (or left undefined). Attribute values are then inherited down the configuration hierarchy. An attribute of an automaton instance takes on the attribute value associated with the deepest ancestor of that instance that has an explicitly-defined value for that attribute. That is, we take the value of an attribute from the closest ancestor for which it is defined.

Our approach of inheriting attribute values from ancestors with like-named attributes is simple and flexible, since it allows one to specify a value for an attribute of a configuration and then explicitly make exceptions by explicitly assigning values for that attribute at some of the components of that configuration. However, like the name-based approach to shared actions, the naming-based approach has scalability problems. As automaton systems grow large, it is likely that one may want to use a given attribute name for different purposes at different sub-configurations in the system. However, at the least common ancestor of these sub-configurations, the names would clash, since they would have different meanings and possibly different types. Therefore, just as we are
investigating the use of a renaming operator for changing the names of the external actions of a composition, we are considering a renaming operator for changing the name of an attribute of a configuration. We envision that one name for the attribute would be used within the configuration, while the ancestors of that configuration would use a different name for the same attribute.

Inheritance of edge information in the configuration graph is handled differently from the inheritance of other attribute values. This is because configuration edges specify a relationship between two automata, whereas other attributes are individually assigned. Since compositions are merely an abstraction, the only automata that actually "do anything" in an I/O automaton system are the instances of base types. Therefore, like other attributes, the relationships specified by edges in the configuration graph can have an affect on the system only if that information is somehow made available to the instances at the bottom of the configuration hierarchy. For example, an edge that represents a communication link between two compositions in a configuration graph can affect the computation only if at least two instances, one in each of the two sub-configurations, inherit that edge and communicate with each other as a result.

Just as for ordinary attributes, we would like the ability to selectively inherit edge information in the composition hierarchy. That is, we do not necessarily want all the components of a sub-configuration to inherit the edges of that sub-configuration, but instead would like to choose which components inherit that information. To support this, Spectrum provides an abstraction mechanism called a port. The idea of establishing connections with ports appears similar in flavor to mechanisms that have been provided in other systems, such as the port mechanisms in Darwin [26, 34, 28] and DEVS [48] the channel construct in Occam [39], and the import and export of variables in COSPAN [20], except that Spectrum ports have no predefined semantics. That is, Darwin, DEVS Occam, and COSPAN each have a particular communication semantics associated with their connection mechanism. In Spectrum, however, the meaning attached to an edge is strictly a matter of how that edge information is used by the incident automaton instances. Spectrum ports are strictly a syntactic device, allowing one to "bundle" collections of edges from components of a sub-configuration and connect them to other ports. The meaning is derived from how the edge information is used to parameterize the automaton types.

The edge inheritance mechanism is described most easily "bottom-up." Recall that for each instance of a base type, we create a port for each set of configuration edges that appears as an attribute of that base type, and that each edge incident to that instance is associated with ("connected to") a particular port. In addition, one may associate any number of ports with each sub-configuration. Since each sub-configuration in the configuration hierarchy is itself represented as a configuration graph, there may be edges connecting various components within the sub-configuration. In addition to these edges, one may also create edges that connect components of a sub-configuration to the ports of that sub-configuration. It is the edges that connect components to the ports of the parent sub-configuration that define how the edges incident to that sub-configuration will be inherited by its components. In general, we say that two automata A and B share a (directed) edge iff there is a sequence of ports p_1, p_2, ..., p_n such that p_1 is a port of A, p_n is a port of B, for each pair p_j, p_{j+1}, there is a (directed) edge from port p_j to port p_{j+1}, and there exists exactly one pair of ports p_j and p_{j+1} in the sequence such that p_j and p_{j+1} appear at the same level in the configuration hierarchy.

As an example, in Figure 6, the octagon and triangle sub-configurations have ports connected to
some of their components. In the case of the octagon, the port connection indicates that each edge incident to the port of an instance of the octagon is, in fact, incident to the leftmost component of the octagon sub-configuration. Therefore, given the definition of the process automaton type in Figure 4, the left process in each instance of the triangle configuration may send messages to the right process through their shared channel, and in addition inherits the connections of the triangle. Continuing up the configuration hierarchy, the leftmost triangle in each instance of the octagon may send messages to each of the two other triangles through the corresponding channels and also inherits the edges of the octagon. Thus, rewriting the hierarchy as a flat structure and inheriting all the edges would result in the configuration shown in Figure 7 that has instances of the base types only. (When an automaton has only one port, we use a shorthand notation in which the nub representing the port is omitted and edges are drawn directly to the icon.)

Since ports represent sets of configuration edges that are attributes of automaton instances at the bottom level of the configuration hierarchy, and since attributes are typed, it is necessary to type-check the configuration. That is, we must ensure that all edges drawn to a particular port are of the same type, and that the types of edges associated with ports for instances of base types match the types of the corresponding attributes. The current implementation supports only one port per automaton instance. Therefore, in the current implementation, each automaton type may have only one set of configuration edges as an attribute. We are planning to support multiple ports in the future. Such support will require that our pictorial syntax have support for both distinguishing
among the ports of a given instance and establishing a correspondence between ports and attributes.

**Shared memory:** Recall that the Spectrum programming language provides the ability to declare variables that may be shared among several automata. These variables may be accessed in "shared memory actions," special output actions that read and/or update the shared memory variables in an atomic step.

In the configuration, we specify which automaton instances actually share variables. This is specified with a closeout operator (defined in Section 2.4) that may be applied to compositions. Consider, first, the lowest level of the composition hierarchy, where compositions that consist only of instances of base types. For each of these lowest level compositions, we define a set *exposed* that initially consists of the union of all the sets of shared variables declared within its components. The set is "exposed" in the sense that any of these variables may be modified externally to each of those automata. Now, if we would like to specify that some subset of these variables is available only to those automata within the composition, we may apply the closeout operator on that subset. Applying this operator makes those variables private to the composition, so that only the components of the composition may read and write the variables. Thus, each time the closeout operator is applied, some of the shared variables are removed from the set *exposed*.

Now, as we move up the composition hierarchy, we may treat the set of exposed shared variables in a higher-level composition just as we treated the declared shared variables at the lowest level. That is, for each composition, we can define the set *exposed* to be the union of the exposed shared variables of its components. Closeout may be performed on a subset of these variables, and some may remain exposed up to the next level. Thus, we are able to provide a convenient scoping mechanism for shared variables.

In our pictorial syntax, the union of the sets of exposed variables of the components of a configuration are shown at the upper right corner of that configuration's definition. The variables to which closeout has been applied are shown in lower case letters, whereas the variables exposed to the next level are highlighted in capital letters. For example, in Figure 8, the process automata (the squares) have three shared variables $x$, $y$, and $z$. The variable $z$ is closed out within the composition defined by the triangle type, while the variables $x$ and $y$ are exposed to the next level of the composition hierarchy. The composition defined by the octagon consists of three instances.
of the triangle type sharing common variables \( x \) and \( y \), where \( y \) is private to each instance of the octagon composition and \( x \) is exposed to the next level of the hierarchy. Finally, at the top level, the closeout operator is applied to the shared variable \( x \). Thus, there is one variable \( z \) shared by all the processes, there is a variable \( y \) for each of the two instances of the octagon composition that is shared by the three triangle compositions in each, and there are six different variables \( z \), one for each pair of processes in the six instances of the triangle configuration. Just as for attributes, a graphical interface might treat the list of exposed and closed out variables as a "pop-up" menu in order to avoid excessive screen clutter during the configuration process.

Because like-named shared variables are identified with each other, we require that the types of their like-named shared variables must agree among components of a composition. This type compatibility is enforced in the current implementation. Just as for attributes, we expect that in specifying large systems it would be convenient to rename exposed variables as one proceeds up the composition hierarchy in order to avoid the name clash problem. We are currently investigating this possibility.

Superposition: Superposition is the operator that allows one automaton to "see inside" the state of another. Recall, from Section 4 that one may declare as unconstrained certain state components of an automaton type. If one creates an instance of this automaton type, then it is possible to superpose that instance on top of any automaton whose state type matches the unconstrained variables. The result is that the higher level automaton can, at all times, see the values of the state components of the lower level automaton. As defined in the model, when an action is shared by the higher and lower layers of a superposition, the values for the unconstrained variables used in
computing its new state (the effect clause) are the values of those variables in the old state of the underlying automaton.

In our pictorial syntax, we represent a superposition with a dashed line. For example, in Figure 9, the triangle configuration is defined as the superposition of a square or an ellipse. Sometimes, we may wish the lower level of a composition to be the composition of several automata. That is, we may want the higher layer to be able to observe the collective states of a number of instances of base automaton types. This is especially useful when one wants to construct an automaton to monitor invariants on the global state of a system. In order to support this, we define a state type not only for each instance of a base type, but also for each composition.

The state type of a composition is determined directly from the state types of its components. It consists of one variable for each automaton type represented in the composition, where that variable is a multiset whose elements each represent the state of one automaton instance of the corresponding automaton type. For example, if a composition contains three automata of type $A$ and two automata of type $B$, then its state type contains two multisets, one for $A$ and one for $B$, and these multisets would contain three elements and two elements, respectively. The type of the elements of each multiset is determined by the state type of the corresponding automaton type. Note that this definition of state type for a composition accommodates dynamic process creation, since new automaton instances may be represented as additional elements of the corresponding multisets. In this example, if a new automaton of type $A$ were added to the composition, this would result in a fourth element in the multiset for $A$.

Sometimes, one may want the higher level automaton in a superposition to be a composition. However, under the usual I/O automaton composition definition, the composition of two automata
with unconstrained sets of variables does not result in an automaton that is unconstrained for the union of those sets of variables. This is handled with a relaxation operator that forces an automaton to be unconstrained for a particular set of variables [15]. We are investigating the use of a relaxation operator in the Spectrum configuration mechanism, but currently we require that the higher level automaton in a superposition be an instance of a base type, and not a composition.

Dynamic process creation: In order to specify the logical structure of a distributed system with dynamic process creation, it is necessary not only to specify the relationships among the system components in existence at the beginning of the execution, but also to specify the relationships among all the system components that might potentially be created during the execution. This set of potentially created components may be infinite in some systems, and although only some of them will ever come into existence, it is necessary to describe the structural relationships among all of them. These structural relationships include their relative positions in the composition and superposition hierarchies, the variables shared among them, logical connections, and attribute values.

To support structural specifications for systems with infinitely many potential components, we extend the notion of a configuration as a hierarchical graph to that of an infinite hierarchical graph. To accomplish this, we introduce a simple mechanism called the seed that allows one to write concise specifications of these infinite graphs. In its simplest form, a seed represents an automaton instance that is “waiting to be created.” In other words, it is a placeholder for an automaton; when a creation action occurs for that automaton, the seed expands into the automaton instance it was meant to be. The placement of the seed indicates the potential automaton’s place in the composition, superposition, and closeout hierarchies so we can tell to what composition the potential automaton belongs, its position in a layered system, and the set of variables it shares with other automata. In addition, the seed provides us with an object to which we may attach attribute information and with which we may form logical connections to other instances or seeds.

Using seeds as placeholders for instances of base types provides us with a way to associate structural information with automata not yet created, but in a system with infinitely many potential automata, it is not possible to specify all of these placeholders directly. Therefore, we allow seeds to represent not only potential instances of base types, but also entire sub-configurations. These sub-configurations may, in turn, contain seeds that represent still more sub-configurations, possibly recursively, provided that each sub-configuration contains at least one “non-seed” element. The mechanisms for attribute inheritance already described can be used to specify the attribute information for all of these potential system components without the need to specify each one. In this way, by specifying each type of configuration, we can specify concisely an infinite hierarchical graph that contains all the necessary structural information for a distributed system with dynamic process creation. So, the expansion of a seed results in the instantiation of a whole sub-configuration whose component’s attributes and logical relationships are already defined within the infinite hierarchical graph of the configuration. In our pictorial syntax, we represent a seed just as an ordinary automaton instance, except that a dot is placed in its center to indicate the delayed instantiation.

Seed expansion during execution follows an orderly progression. If we view the configuration hierarchy as a (possibly infinite) tree, we can define the current configuration of a system to be the maximum subtree of the configuration hierarchy such that each internal node is an instantiated
configuration and each leaf node is either an instance of a base type or a seed. We define the fringe of the current configuration to be the set of seeds in that tree. In other words, if we start with the top-level configuration and search down the configuration hierarchy tree, the fringe contains the first seed encountered along each branch of the search. In Spectrum, all dynamic process creation takes place at the fringe. Each non-seed component of a seed on the fringe in the current configuration is called a creation candidate. Whenever a creation action occurs for an automaton that is a creation candidate, then the seed containing that automaton is expanded. As a result, the seed is promoted from the fringe to the set of instantiated configurations and the seeds within it become elements of the fringe. The automaton for which the creation action occurred is awakened, any further creation actions for that automaton are ignored, and the automaton may begin to receive other input actions and perform locally-controlled actions. However, any other automata in the sub-configuration represented by the expanded seed remain dormant until their corresponding creation actions occur.

Our orderly progression of configuration expansion is a slightly more limited view of dynamic process creation than that used in the formal I/O automaton model. The limitation is that only those instances currently on the fringe in a running system are eligible for creation, whereas the formal model would allow any instance among the infinite collection of potential automata to be created at any time. From an implementation point of view, this would be impractical in general, since we would need to check for each event whether that event was a creation action for any automaton in the infinite configuration. The fringe semantics avoids this problem, since there are always finitely many seeds on the fringe. From a practical standpoint, we do not see this limitation in expressive power creating any serious problems in defining the configurations of real systems, since one would expect (as is the case in the examples we have seen) that dynamic process creation takes place in an orderly fashion in real systems, and that a process will request the creation of another processes only if that other process is somehow logically related to the first. This logical relationship would be captured naturally in the configuration in a way that would result in the necessary seed being on the fringe. For example, in a nested transaction system, such as those described in [31], transactions are created dynamically in a tree structure that unfolds from the root; transactions create sub-transactions that are their immediate children, as opposed to transactions that are several levels below them in the nesting structure.

We use an example configuration for a nested transaction system to illustrate the seed mechanism for specifying the structure of distributed systems with dynamic process creation. Figure 10 shows a configuration for a nested transaction forest. Initially, the configuration consists of (1) an automaton instance (the square in the figure) that represents an environment that creates top level transactions, and (2) a seed for a transaction (the dotted triangle). An instance of a triangle configuration is defined to contain a transaction automaton (the circle), a triangle seed that may expand into its sibling transactions, and another triangle seed that may expand into its descendants. The connections indicate the parent-child relationships in the nested transaction forest. The series of panels (a-e) in Figure 11 are successive configurations that might result from seed expansion in an execution that begins with the configuration of Figure 10. Each expansion results in one new transaction automaton and one new seed. Connectivity is inherited through the ports of the seeds.

Note that although all the transactions in this example have the same automaton type, it would be possible to specify a different dynamic configuration with more structure on the types of children.
of the different transactions. For example, one might specify that the root transactions were all circles, that all children of root transactions are ellipses, and that all octagonal transactions are leaves (perhaps access transactions) in the transaction forest.

In some dynamic systems, it would be difficult to specify the attribute values for all the potential automata at configuration time. For example, in a database transaction system with a data object $x$, one might want a different potential automaton for each possible write access to $x$, where an attribute value of that automaton indicates the value that is to be assigned to object $x$ by that access. Certainly if $x$ may take on any integer value, it would be impossible to specify each of these different access automaton seeds. Therefore, we allow a configuration to contain seeds in which some attribute values are left undefined until the actual instantiation takes place. That is, when we leave some attribute information for a seed undefined in a configuration, we are letting that seed represent a collection of seeds, one for each possible combination of attribute values for the undefined attributes. On creation, then, the arguments of the creation action determine which of these seeds is expanded. It is for this reason that we allow assignment to attribute information in the effect of a creation action. In order to specify additional structure on the system, attribute values in a dynamic system may be inherited. For example, in the triangle configuration defined in Figure 10, it would be possible to specify that some attributes of the circle automaton are inherited from the parent configuration. In this way, the environment might set different attribute values for each top level transactions, but all their descendants' attributes would have values in common.

Attribute inheritance, including inheritance of connections through ports in the configuration graph, is the same for dynamically created automata as it is for static automata. However, some special treatment must be given to the connection information for those automata on the fringe.
As new automata are created in a running system, their connectivity information can affect the signatures, transition relations, and classes of other automata. For example, consider an automaton $A$ that provides some service for its neighboring automata in the configuration graph in a way that is fair to each neighbor. If automaton $A$ is constructed so that it contains one equivalence class in its partition for each of its neighbors in the configuration graph, and a new automaton $B$ is created with an edge incident to $A$, then automaton $A$ will gain (automatically) a new equivalence class to serve the new automaton $B$. We say that an edge is visible to an automaton instance or a creation candidate if the opposite end point of the edge is an instantiated automaton. It is the edges visible to an automaton instance (or a creation candidate) that determine the connection attribute information at that automaton. This definition of visibility applies to all attributes, regardless of whether they are mutable or immutable. Thus, even if a set of configuration edges is an immutable attribute of an automaton, the subset of edges that are actually visible to the automaton may change over time.

Although automata are created in the course of Spectrum simulations, automata are never destroyed. In a sense, this is in keeping with the formal model, where all automata are in existence at the beginning and then some are awakened. Those that complete their task become dormant but do not "go away." However, in the interest of efficient use of memory resources, we are investigating the use of a destruction action that would be an output of an automaton and result in its deletion from the configuration. This would permit the space occupied by dormant automata to be reclaimed, but more importantly it would help to keep the configuration clean, uncluttered by automata not currently in use.
**Time:** No special configuration support is needed for defining the boundmaps of time-constrained automata. We simply use the standard attribute mechanism to define the lower and upper bounds on step times for processes. For example, an automaton type might have an real-valued attribute called `clock-rate` that is used to determine the bound map for its processes. In the configuration, we simply specify the value for this attribute in order to specify the process step times. The Spectrum simulator, then, must guarantee that the executions produced obey the timing constraints specified by the boundmap.

Attribute inheritance is particularly useful for specifying boundmaps. For example, we might have a collection of automata modelling processors that run at the same speed. Using inheritance, we can specify the actual processor speed only once and have that value determine the step times for all the automata in the collection.

## 6 Discussion

Having presented the design of the Spectrum Simulation System, we now describe some specific ways in which the separation of structure and function in Spectrum are helpful in the specification and design of distributed systems. We then say a few words about the current Spectrum implementation and suggest some directions for future work.

One of the most important advantages of separating structure and function clearly is the ability to modify the two independently. One may substitute functional components within a system without modifying the configuration, and one may modify the configuration without going back to the program text. For example, one might substitute a communication channel that reorders messages for a FIFO channel, or one might try a given algorithm in various network topologies.

In addition to flexibility in constructing and modifying system designs, the separation of structure and function is helpful in supporting program verification. From the point of view of constructing correctness proofs, it is helpful to have a structural specification on which one can depend while reasoning about the functional components. For example, with *a priori* structural knowledge about the system, it is possible to immediately rule out certain possibilities for module interaction that might otherwise have to be proven impossible based on the behaviors of the modules themselves. Furthermore, from the point of view of program testing, the separation of structure and function is also helpful in supporting the verification effort.

Recall from Section 2 that a problem is specified in the I/O automaton model as a *schedule module*, which is an external interface together with a set of allowable behaviors. Because the functional components and the configuration are specified separately in Spectrum, it is possible to create additional I/O automata (based on the schedule module) for the express purpose of checking the correctness of system executions and insert them directly into a configuration *without interfering with the execution or needing to modify the functional components in any way*. These additional I/O automata, which we call *spectators*, have as input actions all of those actions of the system that appear in the interface of the schedule module. As the execution proceeds, the spectator maintains history information and observes each action, checking it against the set of legal executions. Since the set of legal executions is often defined inductively within a schedule module, it is usually a simple matter to transform these conditions into predicates that can be checked by the spectator as each action occurs. Typically, one specifies a set of invariants on the variables of the spectator.
If any of these invariants is violated, it is a signal to the algorithm designer that the correctness conditions have not been satisfied in the execution.

We illustrate the idea of a spectator in the context of the following schedule module for the mutual exclusion problem. Fix $n$, a positive integer, and let $I = \{1, 2, \ldots, n\}$. We define schedule module $M$ with $\text{sig}(M)$ as follows:

Inputs: UserTry\textsubscript{i}, $i \in I$  Outputs: Crit\textsubscript{i}, $i \in I$
UserExit\textsubscript{i}, $i \in I$  Rem\textsubscript{i}, $i \in I$

Schedule module $M$ interacts with an environment that may be thought of as a collection of $n$ user processes $u\textsubscript{i}$, $i \in I$, where each process $u\textsubscript{i}$ has outputs UserTry\textsubscript{i} and UserExit\textsubscript{i}, and has inputs Crit\textsubscript{i} and Rem\textsubscript{i}. A UserTry\textsubscript{i} action means that process $u\textsubscript{i}$ wishes to enter its critical section. A Crit\textsubscript{i} action by $M$ gives $u\textsubscript{i}$ permission to enter its critical section. A UserExit\textsubscript{i} action means that process $u\textsubscript{i}$ is leaving its critical section. Finally, the Rem\textsubscript{i} action gives $u\textsubscript{i}$ permission to continue with the remainder of its program. If $\beta$ is a sequence of actions of $M$, then we define $\beta|i$ to be the subsequence of $\beta$ containing exactly the UserTry\textsubscript{i}, Crit\textsubscript{i}, UserExit\textsubscript{i}, and Rem\textsubscript{i} actions. Before defining the allowable schedules of $M$, we define the set of well-formed sequences of actions of $M$. Let $\beta$ be a sequence of actions in $\text{sig}(M)$. We say that $\beta$ is well-formed iff for all $i \in I$, all prefixes of $\beta|i$ are prefixes of the infinite sequence UserTry\textsubscript{i}, Crit\textsubscript{i}, UserExit\textsubscript{i}, Rem\textsubscript{i}, UserTry\textsubscript{i}, Crit\textsubscript{i}, $\ldots$. This says, for example, that a process will not issue a try request while in its critical section.

We define the set $\text{scheds}(M)$, the allowable external behaviors of $M$, as follows. Let $\beta$ be a sequence of actions in $\text{sig}(M)$. Then $\beta \in \text{scheds}(M)$ iff the following conditions hold:

1. $M$ preserves well-formedness in $\beta$.
2. If $\beta$ is well-formed, then $\forall i, j \in I$, if Crit\textsubscript{i} and Crit\textsubscript{j} occur in $\beta$ (in that order), then UserExit\textsubscript{i} occurs between them.

Condition (1) says that the mutual exclusion algorithm is not the first automaton to violate well-formedness. Condition (2) says that no two processes are in their critical sections simultaneously, provided that the user processes preserve well-formedness.

We would like to write a spectator to check executions of an automaton system against the allowable behaviors specified by schedule module $M$. A spectator is simply an I/O automaton with no output actions that observes the actions taken by other automata. By writing spectators without output actions, we need not be concerned that a spectator could interfere with the execution of an algorithm. Furthermore, it is not sensible to have spectators report a detected error by means of an output action, because the scheduler might not give the spectator a chance to take a step until much later in the execution. Instead, we write a spectator so that one of its own invariants is violated whenever it detects an error. Conveniently, this interrupts the simulation immediately, so that the user may explore the source of the error.

The spectator in Figure 12 corresponds to Condition (2) of schedule module $M$, and could be used to check the executions of a mutual exclusion algorithm. In this spectator, the state component last-crit keeps track of the index of the process most recently in the critical section, and the component in-crit keeps track of whether the last input action was Crit or UserExit. When a Crit action occurs, the invariant error = false is violated if and only if no UserExit occurred since the last preceding Crit action. When a UserExit action occurs, the invariant is violated if and only
AUTOMATON CheckMutex
STATE last-crit: integer, in-crit: boolean, error: boolean
INVARIANT error = false
INPUT initially
  EFF last-crit := 0
  in-crit := false
  ok := true
INPUT Crit(a: automaton_id)
  EFF error := in-crit
  last-crit := a
  in-crit := true
INPUT UserExit(a: automaton_id)
  EFF error := ((in-crit = false) OR (last-crit != a))
  in-crit := false

Figure 12: A spectator for mutual exclusion.

if the argument of the action is not the index of the process currently in the critical section. It is
easy to see how these cases are derived from Condition 2 of schedule module \( M \). One could write
a similar spectator automaton type for checking that each user's execution is well-formed.

Note that a spectator depends only on the problem specification, and never on the algorithm
itself. That is, a spectator for a given problem specification could be used to check any solution to
that problem.

In addition to verifying that executions are correct, spectators can be helpful in the analysis of
algorithm efficiency. For example, one might use a spectator to count the number of messages sent
in an execution, or to keep track of the rates at which processes enter their critical sections in a
mutual exclusion algorithm. Again, because a spectator has no output actions, we know that such
analysis cannot interfere with the algorithm execution.

Other features of the Spectrum programming language are useful in conjunction with spectators.
For example, superposition is useful for allowing a spectator to observe not only the actions of a
system, but also the states. In this way, a spectator can detect a violation of an invariant on
the global system state before that inconsistency results in an incorrect system behavior. The
MAINTAIN clause, explained in Section 4 is useful in this regard. When a spectator is written as a
superposition, a single maintain clause can be used to update the spectator's local variables each
time an action occurs in the underlying system, whether or not that action is in the signature of
the spectator itself. This simplifies writing the spectator, since it is not necessary to include all the
actions of the underlying system in its signature in order to "catch" all the state changes.

We have found that the separation of structure and function provided by Spectrum is particu-
larly helpful for students who are first learning distributed algorithms. It forces the students to
think locally about what each part of the system does and then to think separately about how the
pieces fit together. So far, we do not have enough experience to say whether this approach will also
be beneficial to "seasoned" algorithm designers, although we expect that it will.

The first prototype implementation of Spectrum, without shared memory, superposition and
dynamic process creation, was completed in 1990 [14]. This prototype was written in C [25] and provided graphical mechanisms for constructing configurations and visualizing executions. A second prototype, with an improved language syntax and support for shared memory, superposition, and dynamic process creation, has been completed recently. The language parser for the second prototype is written in C (using yacc [29]) and produces a Scheme [46] file that is provided as input to the new Spectrum simulator, written in Scheme. In order to make some of the simulator’s utility functions available to the Spectrum compiler, we have translated these utilities into C with a Scheme-to-C compiler [7]. Currently, configurations are specified directly in Scheme, but work on graphical configuration and visualization capabilities is underway. Following sufficient experimentation with this second prototype and completion of the graphical configuration and visualization tools, we plan to make a version of Spectrum available for general distribution.

The design of the software tool itself is less interesting than the mechanisms we have described for specifying the functional and structural components of the system. The system is divided into three components: The compiler parses I/O automaton type descriptions and creates “intermediate code,” a scheme file, that is used as input to the simulator. The compiler checks for proper syntax and type compatibility, and it enforces all of the variable access restrictions. The configuration mechanism is a set of scheme functions that support creation of configurations, subconfigurations, compositions, superpositions, instances, and seeds, and provide the ability to closeout shared variables, assign attribute values to automata and create logical connections between automata. Currently, users must create a configuration file with these provided functions, but ultimately we plan to have a graphical configuration mechanism on top of these functions that will allow users to design the structure of their systems in our pictorial syntax. The simulator itself takes as input the intermediate code for the automaton types and the configuration file and produces as output executions of the automaton system described.

At each step of the execution, a scheduler picks a class containing some enabled action(s), an enabled action from that class is chosen and the selection clause for that action is executed in order to assign values to the arguments. Then, the set of automata having the action as an input is determined (based on the signatures, including the WHERE clauses). These automata, together with the automaton having the action as an output, are called the participant automata of the action. Following this determination, the action is executed at each participant. Executing an action at an automaton involves non only executing the EFF clause, but also executing the MAINTAIN clause and checking the predicates declared as INvariant. Since references to unconstrained variables in EFF clauses must use evaluate to the old values of those variables (the values before the step occurred), the order of execution of EFF clauses is top-down in the superposition hierarchy. However, the MAINTAIN and INvariant clauses are executed “bottom-up” in order for them to observe the new values. When a create action occurs for an automaton inside a seed on the fringe, the necessary instantiation takes place, including updating the attribute values (and perhaps the set of classes) for automaton instances that gain connections as a result of the creation. Following execution of the action at each participant, each class of each participant is checked to see if it now has (or no longer has) an enabled action, and the scheduler updates is data structures accordingly. In the case of the randomized or round-robin scheduler, this update simply amounts to inserting into (or deleting from) a list, but in the case of the time-constrained scheduler, new lower and upper bounds on the step times of the classes are computed.
The notion of fairness for a scheduler in a simulation system is rather odd. In automaton systems with only finite executions, every schedule is fair by definition, so in principle the particular scheduling algorithm does not matter. In automaton systems with infinite executions, we can only observe a finite prefix of each execution, since we must eventually stop the simulation. Since every execution is a prefix of a fair execution, again in principle any scheduling algorithm will do. However, this is rather unsatisfying and does not correspond to our intuition about fairness. Therefore, we have adopted the following definition in designing scheduling algorithms. We say that a scheduling algorithm for an I/O automaton system is fair if it can be proven (either with certainty or with high probability) that if the scheduler were permitted to run forever, then every class would be given a chance to take a step infinitely often. So, for example, a round-robin scheduler would fit this definition, since we can provide an upper bound on the number of steps that may occur between the time a class becomes enabled and the time it takes a step or becomes disabled. Similarly, a randomized scheduler using a uniform probability distribution can be shown to satisfy our fairness criterion with high probability.

An interesting question is how the ideas presented in this paper might be supported in a programming language, as opposed to in a simulation language intended only for specification and design. Clearly, support would be needed for writing process descriptions that are parameterized by the system configuration data. To support dynamic process creation fully, it would be necessary to have mechanisms that provide for the updating configuration information when new logical connections are created. In addition, it would be necessary for the run-time system of the language to direct IPC traffic "under the covers." That is, since functional individual processes may not explicitly name the other processes with which they may be communicating, it may be necessary for the run-time system to perform dynamic linking for remote procedure calls or to determine the recipient(s) of each message based on the configuration data. This capability does not appear to be out of reach, and the resulting separation would appear to be of significant value in program design and implementation.

7 Conclusion

One of the most important principles of system design is to separate different aspects of problem into smaller subproblems to be solved independently. In this paper, we have presented and illustrated a particular approach to the specification and design of distributed systems in which the functional components of a system are separated from the structure of a system rather than including structural information as part of the program text. In this approach, functional components are represented as program types (in this case, I/O automaton types) that are parameterized by configuration data described separately. The two primary advantages of this approach are flexibility and simplification of the design process. Flexibility arises from the ability to substitute or rearrange system components without modifying the program text. Simplification arises from the ability to think about each functional component separately, as a general program and without regard to the particular way in which it will be connected to the rest of the system, and the ability to then specify a (possibly dynamic) structure in which those components are to run.

Gelernter and Carrier have called for a general model of coordination:

It would be nice to have a theoretical foundation for general coordination. We would
like to see the following characteristics in such a model. First, a simple definition of computational space and time where a point in space is identified with a single locus of control (or a single Turing Machine), and a point in time is defined as the current states of many loci or TMs. Second, a model that becomes a TM when projected onto the time axis at some spatial point, and becomes a “current coordination state” when projected onto the space axis at some temporal point. ([12], p. 106)

The separation of concerns achieved within the Spectrum Simulation System appears a possible candidate for such a model. Clearly, the I/O automaton model provides a simple definition of computational space and time. Each point in space in an I/O automaton system is a single I/O automaton. Each point in time can be viewed as the collective states of all of the automata, together with the current configuration of the system. When we project onto the time axis at some spatial point, we see the local execution of a single automaton, complete with its local configuration information (attributes). When we project onto the space axis at some temporal point, we see a snapshot, the global state of the system and the current configuration. By viewing system coordination as a structure, rather than as an activity, we are able to accomplish a separation that is not only convenient for designing systems, but also provides a possible foundation for the study of system coordination in general.

Acknowledgements

I thank Nancy Lynch, Bill Weihl, Alan Fekete, and Mark Tuttle for helpful advice and encouragement in the early phases of this project, Christopher Colby for his effort in constructing a graphical user interface for the first prototype of Spectrum, Christopher Davis for his recent work in constructing a parser for the new Spectrum system, and Lincoln Smith and Karl Stiefvater for their work on the simulator and configuration mechanism. I also thank Joel Bartlett for his advice and assistance in using his Scheme-to-C compiler. I am grateful to all of the users of the first Spectrum prototype: Mini Gupta, John Leo, Stephen Ponzio, and the students in Nancy Lynch’s graduate course in Distributed Algorithms taught at M.I.T. in the Fall of 1990. I also thank Terry Idol, Paul McCartney, and Celia Vergara for their careful reading of earlier drafts.

A Language Grammar

The following is a partial grammar for the Spectrum language. Syntactic sugar, such as infix operators for common type operations and value constructors for aggregate types, is omitted.

decl ::= TYPE <type-name> type

type ::= integer | boolean | real | automaton_id | character |
    string | set(type) | multiset(type) | sequence(type) |
    tuple(comma-fields) | mapping(type,type) |
    graph(type,type) | array(type) | vertex(type) |
    edge(type) | type-name

comma-fields ::= <field-name>:type [, comma-fields]
fIELDS ::= <field-name>:type [fields]

43
autdef ::= AUTOMATON <aut-name> [(comma-fields)] header
          [inputs] [classes]

header ::= [state] [shared] [unconstrained] [maintain] [invariant]
          [initially]

state ::= STATE fields
shared ::= SHARED fields
unconstrained ::= UNCONSTRAINED fields
maintain ::= MAINTAIN statements
invariant ::= INARIANT predicates
initially ::= INIT statements

classes ::= CLASS [(comma-fields)]
          [STEP TIME (expression, expression )]
          locals [classes]

class-params ::= identifier:expression [[,...],class-params]
locals ::= output [locals] | internal [locals]
output ::= OUTPUT act-name [(comma-fields)] precond select effect
internal ::= INTERNAL act-name [(comma-fields)] precond select effect

inputs ::= input [inputs] | creation [inputs]
input ::= INPUT act-name [(comma-fields)] where effect
creation ::= CREATION act-name [(comma-fields)] where effect

where ::= WHERE predicate
precond ::= PRE predicate
select ::= SEL statements
effect ::= EFF statements

func ::= FUNCTION <func-name> [(comma-fields) [MUTATES field-names]]
       [RETURNS type] [vars] [body] [return]

vars ::= DECLARE fields
body ::= statements
return ::= RETURN expression

conditional ::= IF predicate THEN stmt [ELSE stmt]
loop ::= FORALL <identifier> IN expression [UNTIL predicate]
        DO stmt
        * the expression following IN must evaluate to a set *

test ::= <func-name>() [expressions] | constant | identifier
expressions ::= expression [,expressions]

predicate ::= expression
* a predicate must evaluate to a boolean *
predicates ::= predicate [,predicates]

statement ::= <func-name> ( [expressions] ) | conditional | loop
statements ::= statement [statements]
stmt ::= statement | BEGIN statements END

References


