Report Number: WUCS-95-17

1995-01-01

Formal Specification of a Dynamically Configurable Distributed System

Ram Sethuraman and Kenneth J. Goldman

The Programmers' Playground is a programming environment that supports end-user construction of distributed multimedia applications. The system implements a new programming model that is based, in part, upon ideas from the formal I/O automaton model of Lynch and Tuttle. Important features of The Programmers' Playground are a separation of communication and computation and graphical support for dynamic reconfiguration. This paper provides a formal specification of the Playground programming model and runtime system in terms of the I/O automaton model on which it is based. Exploiting the compositionality properties of the I/O automaton model, the formal specification is describd as... **Read complete abstract on page 2.**

Formal Specification of a Dynamically Configurable Distributed System

Ram Sethuraman and Kenneth J. Goldman

Complete Abstract:

The Programmers' Playground is a programming environment that supports end-user construction of distributed multimedia applications. The system implements a new programming model that is based, in part, upon ideas from the formal I/O automaton model of Lynch and Tuttle. Important features of The Programmers' Playground are a separation of communication and computation and graphical support for dynamic reconfiguration. This paper provides a formal specification of the Playground programming model and runtime system in terms of the I/O automaton model on which it is based. Exploiting the compositionality properties of the I/O automaton model, the formal specification is describd as a composition of several modules. A behavioral specification of each module is presented, followed by an I/O automaton that implements each specification. We present the specification in two stages, a centralized specification that captures the allowable behaviors, and then a detailed distributed implementation.

Formal Specification of a Dynamically Configurable Distributed System

Ram Sethuraman and Kenneth J. Goldman

WUCS-95-17

July 1995
(revised November 3, 1995)

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899

Formal Specification of a Dynamically Configurable Distributed System

Ram Sethuraman Kenneth J. Goldman*

15 July 1995
Revised 3 November 1995

Abstract

Abstract: *The Programmers' Playground* is a programming environment that supports end-user construction of distributed multimedia applications. The system implements a new programming model that is based, in part, upon ideas from the formal I/O automaton model of Lynch and Tuttle. Important features of The Programmers' Playground are a separation of communication and computation and graphical support for dynamic reconfiguration.

This paper provides a formal specification of the Playground programming model and run-time system in terms of the I/O automaton model on which it is based. Exploiting the compositionality properties of the I/O automaton model, the formal specification is described as a composition of several modules. A behavioral specification of each module is presented, followed by an I/O automaton that implements each specification. We present the specification in two stages, a centralized specification that captures the allowable behaviors, and then a detailed distributed implementation.

1 Introduction

The Programmers' Playground[5] is a software library and run-time system designed to support end-user construction of distributed multimedia applications. Design goals include the separation of communication from computation, dynamic reconfiguration, and the uniform treatment of discrete and continuous data types. The current version of Playground supports applications written in C++ on top of the Solaris operating system. Communication among modules is dynamically configured by end-users using graphics tools. Physical migration[4, 15] and end-user construction of direct-manipulation user interfaces for distributed applications[11] are also supported. An undergraduate course in distributed application paradigms has been taught using Playground as the programming environment. Also, Playground has been used to implement a video-conferencing application on top of an ATM network [2] built at Washington University.

The system is based on a new high-level approach to interprocess communication. Functional components of a concurrent system are written as encapsulated application programs that act upon local data structures, some of which may be published for external use. Relationships among these applications are specified by logical connections among their published data structures. Whenever an application updates published data, communication takes place implicitly according to the configuration of logical connections. The Playground programming model is based on ideas from the formal *I/O automaton model* of Lynch and Tuttle [10], a natural model of distributed systems that

*This research was supported in part by the National Science Foundation under grant CCR–94–12711.

provides compositionality properties and a clear separation of input and output actions. However, there are important semantic differences between the Playground semantics and the I/O automaton model, particularly in terms of asynchrony and in the handling of input actions.

In this paper, we come full circle to provide a formal description of the Playground run-time system in terms of the I/O automaton model. Providing a formal semantics for a system provides the obvious advantages of a clear understanding of the semantics and the ability to reason carefully about the properties of the system. In addition, it provides a vehicle with which one can explore the impact of potential new features on the overall semantics of the system.

We begin by modeling a centralized Playground run-time system automaton that is composed with a set of *application automata*. The application automata model the computational components whose communication structure is dynamically configured by programmers and end-users. The Playground run-time system automaton acts as an intermediary between the application automata and is responsible for transporting data among the various application automata according to the (dynamically changing) configuration. The centralized run-time system captures the set of allowable behaviors that we expect from any distributed implementation.

In the course of this discussion, certain key properties of user-configurable systems emerge. For example, the fact that reconfiguration requests are imposed externally rather than by the modules themselves requires a careful understanding of the interval in which two modules may be considered "connected."

Building on the centralized system specification, we formally model the distributed implementation of the run-time system as the composition of several components. Exploiting the compositionality properties of the I/O automaton model, we separately specify the allowable behaviors of each of the components. Safety and liveness properties are formally stated, as well as restrictions on how the external environment must interact with each component. To complete the presentation of each module, complete I/O automata implementing these specifications are provided.

The remainder of the paper is organized as follows. Section 2 reviews basic concepts of the I/O automaton model as used in this paper. Section 3 summarizes Playground's programming model, and provides a comparison of the programming model and the formal I/O automaton model, highlighting important similarities and differences. In Section 4, we model the centralized Playground run-time system. This is followed in Section 5 by a formal description of the distributed implementation. In Section 6, we outline a proof showing that our distributed implementation presented in Section 5 solves the Playground specification provided in Section 4. The paper concludes with a discussion of the utility of this formal framework for the study of additional communication and synchronization features that are envisioned for the Playground programming model and run-time system.

2 The I/O Automaton Model

This section provides a brief overview of the I/O automaton model [10]. The model is both the basis for the Playground programming model and the formal framework we will use to describe the Playground semantics and implementation.

An *I/O automaton* is a state machine with a *signature* consisting of a set of *input actions* and a set of *locally controlled actions* (divided into *output actions* and *internal actions*). Locally controlled actions are under the control of the automaton, while input actions may occur at any time. Automata may be composed such that when an output action of one automaton occurs, all automata having a same-named action as an input action make a state transition simultaneously. A *behavior* of an I/O automaton is a sequence of input and output actions that may occur in an

execution of that automaton.

The I/O Automaton model provides a framework in which one can write precise statements of the problems to be solved by modules in concurrent systems, careful algorithm descriptions and correctness proofs. In addition, the model can be used for carrying out complexity analysis and for proving impossibility results. The I/O automaton model is significantly different from CCS [12] and CSP [6] 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 I/O-systems [7, 8, 9]. The following introduction to the model is adapted from [10], which explains the model in more detail, presents examples, and includes comparisons to other models. Readers already familiar with the I/O automaton model may skip this section without loss of continuity.

2.1 I/O Automata

I/O automata are best suited for modeling systems in which the components operate asynchronously. Each system component is modeled as an I/O automaton, which 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 $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$ with the property that for every state s' and input action π there is a transition (s', π, s) in $steps(A)$, and
- an equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

We refer to the actions in the signature of A as the actions of A, denoted $acts(A)$ The equivalence relation $part(A)$ will be used in the definition of fair computation. Each class of the partition may be thought of as a separate process. We refer to an element (s', π, s) of $steps(A)$ as a *step* of A. If (s', π, s) is a step of A, then π 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 the automaton is unable to block its input.

An *execution* of A is a finite sequence $s_0, \pi_1, s_1, \ldots, \pi_n, s_n$ 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 α, denoted $sched(\alpha)$, is the subsequence of α consisting of the actions appearing in α. The *behavior* of an execution or schedule α of A, denoted $beh(\alpha)$, is the subsequence of α consisting of *external* actions. The same 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 modeling a complex system by composing a collection[1] of automata modeling the simpler system components. When we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π, all automata having π as an action perform π simultaneously (automata not having π as an action do nothing).

Since we require that at most one system component controls the performance of any given action, we must place some compatibility restrictions on the collections of automata that may be composed. Namely, their sets of output actions must be disjoint, and no automaton may have an internal action that is also an action of another automaton.

We can think of each state of a composition as a state vector indexed by the names of the component automata. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\ldots$ of A, let $\alpha|A_i$ (read "α projected on A_i") be the sequence obtained by deleting $\pi_j\vec{s_j}$ when $\pi_j \notin acts(A_i)$ and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

When an execution of a system is projected onto any component, the result is an execution of that component. The same is true for schedules and behaviors. Other important compositionality results for I/O automata may be found in [10].

2.3 Fairness

We may think of each class in the partition of locally-controlled actions of an automaton as a separate "control thread" that is responsible for the actions in that class. Of all the executions of an I/O automaton, we are primarily interested in the "fair" executions — those that permit each of these "threads" to have infinitely many chances to perform output or internal actions. Each equivalence class of a component is an equivalence class of the composition. Hence, that composition retains the essential control structure of the system's primitive components.

A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of *part*(A):

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

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

2.4 Problem Specification

A problem to be solved by an I/O automaton is formalized as a set of behaviors. An automaton is said to *solve* a problem P provided that its set of fair 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 $sig(H)$, and a set $scheds(H)$ of *schedules*. Each schedule in $scheds(H)$ 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 schedules $scheds(H)$

[1]The collection must be countable, but here we deal only with finite collections.

is the set of sequences β of actions of H such that for every schedule module H' in the composition, $\beta|H'$ is a schedule of H'.

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. A set of sequences P is said to be *prefix-closed* if $\beta \in$ P whenever both β is a prefix of α and $\alpha \in$ P. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that *finbehs*(M) is prefix-closed.

Let Φ be a set of actions and let M be a prefix-closed module, such that $\Phi \cap int(M) = \emptyset$. Let P be a nonempty, prefix-closed set containing sequences of actions from Φ. We say that M *preserves* P if $\beta\pi|\Phi \in$ P whenever $\beta|\Phi \in$ P, $\pi \in out(M)$, and $\beta\pi|M \in$ *finbehs*(M). 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.

In [10], it is shown that if each component of a composition preserves a property, then the composition preserves the property. A definition for the composition of schedules is also provided, and it is shown that the schedules of a composition of components are the same as the composition of the schedules of the components.

3 The Playground Programming Model

The Programmers' Playground is designed to support end-user construction of distributed multimedia applications. The system is based on a connection-oriented model of interprocess communication in which independent applications interact with an abstract environment. This section provides a brief informal overview of The Programmers' Playground in sufficient detail for understanding the more formal system descriptions that follow. For the purposes of this paper, the Playground programming model is taken as "given." Information on related programming models and details about the Programmers' Playground goals, design, implementation may be found elsewhere [3, 5].

Playground provides a model of interprocess communication in which each *application* in a system has a *presentation* that consists of data structures that may be externally observed and/or manipulated by its environment. A *distributed application* consists of a collection of independent applications and a *configuration* of *logical connections* among the data structures in the application presentations. Whenever published data structures are updated, communication occurs implicitly according to the logical connections.

Playground communication is *declarative*, rather than *imperative*. One declares direct high-level logical connections among the state components of individual applications, as opposed to directing communication within the control flow of the application. Once the high-level relationships between state components are declared, if a particular state change in one application should be reflected in the state of another application, then this can be recognized by the system and the necessary communication can be handled implicitly. Thus, output is essentially a byproduct of computation, and input is handled passively, treated as a modifier (or an instigator) of computation.

This declarative approach simplifies application programming by cleanly separating computation from communication. Playground applications do not make explicit requests to establish or effect communication, but instead are concerned only with the details of the local computation. Communication is declared separately as high-level relationships among the state components of different applications.

With implicit communication, it is the configuration (and not directives from within the program) that determine whether or not communication will take place. This means that choices

can be made at configuration time about whether applications will communicate and about how they will communicate (point-to-point, multicast, etc.) without modification of the applications themselves.

The Playground programming model is based on three fundamental concepts: data, control, and connections, presented here in terms of The Programmers' Playground implementation.

3.1 Data

The components of an application's state may be kept private or they may be *published* in a dynamically changing *presentation* so that other applications may access the data. Playground provides a library of publishable data types, including *base types* (integer, real, boolean, and string), *tuples* with named fields, and various *aggregates*. The types may be nested arbitrarily. Each presentation entry has a *public name* and *access privileges*. Recently, two *media types* (audio and video) have been added.

The environment: Each Playground application interacts with an abstract *environment* through its presentation. The environment consists of a collection of other applications that are unknown to this application but that may read and modify the data items in its presentation (as permitted by the access privileges).

Behaviors and specifications: A *behavior* of an application is a sequence of values held by the data items in its presentation. It is the view that the environment has of the application, and (symmetrically) the view that the application has of its environment. A Playground application can be described in terms of a *behavioral specification* including: the data items in the presentation, the behaviors that may be exhibited by the application, and any assumptions made about the allowable behaviors of the environment. Dividing the presentation into input (write-only) data items and output (read-only) data items can simplify the task of constructing a behavioral specification and such a division can be enforced using access protection. Behavioral specifications are similar to the *schedule module* specifications in the I/O automaton model, except that I/O automaton behaviors are sequences of actions, while our behaviors are sequences of state changes at the presentation.

3.2 Control

The *control* portion of an application defines how its state changes over time and in response to its environment. Insulated from the structure of its environment, a Playground application interacts entirely through the local data structures published in its presentation. An application may autonomously modify its local state, and it may react to "miraculous" changes in its local state cause by the environment. This suggests a natural division of the control into two parts: *active control* and *reactive control*. Playground applications may have a mixture of both active and reactive control.

Active control: The active control carries out the ongoing computation of the application. For example, in a discrete event simulation, the active control would be the iterative computation that simulates each event. External updates of simulation parameters could affect the course of future iterations, but would not require any special activity at the time of each change. Input simply steers the active computation without requiring immediate attention on arrival. Active control is analogous to the locally controlled actions of an I/O automaton.

Reactive control: The reactive control carries out activities in direct and immediate response to input from the environment. An application with primarily reactive control simply reacts to each input from the environment, updating its local state and presentation as dictated by that input change. For example, a data visualization application could be constructed so that each time

some data element changes, the visualization is updated to reflect the change. In the above discrete event simulation, one might add reactive control to check the consistency of simulation parameters that are modified by the environment. Reactive control is analogous to the input actions of an I/O automaton.

Specifying control: The active control component of a Playground application is defined by the "mainline" portion of the module. Reactive control is specified by associating a *reaction function* with a presentation data item. This function defines the action to be carried out when that data item is updated by the environment. As a simple example, one might associate with data item x an enqueue operation for some local queue q. With each external update to x, the new value of x would be enqueued into q for later processing by the application.

Atomicity: In order to preserve atomicity of reactive control and consistent semantics for active control, active and reactive control are not arbitrarily interleaved. The run-time system ensures that each reaction function runs as an atomic step as far as the active control can tell. Similarly, the active control may specify atomic steps during which the run-time system prevents both external updates to the presentation and execution of reactive control.

3.3 Connections

Relationships between data items in the presentations of different applications are declared with *logical connections* between those data items. These connections define the communication pattern of the system. Connections are established by a special Playground application, called the *connection manager*, that enforces type compatibility across connections and guards against access protection violations by establishing only authorized connections.

Connections are declared separately from applications so that one can design each application with a local orientation and later connect them together in various ways. Connections are designed to accommodate both discrete data (such as sets of integers) and continuous data (such as audio and video) in a single high-level mechanism, with differences in low-level communication requirements handled automatically by the run-time system according to data type information.

Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections. A given data item may be involved in multiple connections of both kinds.

Simple connections: A simple connection relates two data items of the same type, and may be either *unidirectional* or *bidirectional.* The semantics of a unidirectional connection from integer x in application A to integer y in application B is that whenever A updates the value of x, item y in application B is correspondingly updated. If the connection is bidirectional, then an update of y's value by application B would also result in a corresponding update to x in A. Since no locking schemes have been implemented, updates to data items on the ends of a bidirectional link may cross without seeing each other. Arbitrary *fan-out* and *fan-in* are permitted so that multiple simple connections may emanate from or converge to a given data item. If x in the above example is also connected to integer z in application C, then whenever x is updated, so are both y and z.

Element-to-aggregate connections: A Playground aggregate is an organized homogeneous collection of elements, such as a set of integers or an array of tuples. The *element type* of an aggregate is the data type of its elements. For example, if s is a set of integers, the element type of s is integer. An element-to-aggregate connection results when a connection is formed between a data item of type T and an aggregate data item with element type T. If an element-to-aggregate connection is created between each client's type T data structure and a server's *set(T)* data structure, then the server will see a set of client data structures, and each client may interact with the server through its individual element. In this paper, we do not model the element-to-aggregate connections formally since they are actually implemented by the run-time system as a

collection of simple connections, one for each element.

3.4 Playground and the I/O automaton model

The Playground programming model is partly based on the I/O automaton model, so it is natural that we use the I/O automaton model to provide a formal semantics for the Playground programming model and its implementation. However, due to several considerations (primarily due to implementation, expressive power, and scalability), there are some important differences between the Playground programming model and the I/O automaton model on which it is based. Therefore, before proceeding with the formal description, we highlight the similarities and differences between the formal I/O automaton model and the Playground programming model.

We may think of each application in The Programmers' Playground as an I/O automaton whose action signature corresponds to the set of presentation entries in the application in the following way. Each presentation entry x that is "readable" by the environment would have an associated set of output actions named $x_{out}(v)$, where v is a value in the domain of the variable x. Similarly, each presentation entry x that is "writable" by the environment would have an associated set of input actions named $x_{in}(v)$. Presentation entries that are both "readable" and "writable" would have both sets of actions (input and output). Thus, the behavior of the automaton would correspond to the set of changes occurring at the presentation that forms the boundary between the application and the environment. Output actions would correspond to state changes initiated by the application, and input actions would correspond to state changes initiated by the environment. In this way, behavioral specifications of a Playground application are essentially the same as I/O automaton schedule modules.

When collections of I/O automata are composed to form larger systems, communication occurs through shared actions of the same name. However, to avoid potential name clashes, Playground uses a configuration of logical connections to define the relationships among presentation entries in different applications. Thus, instead of defining communication according to like-named presentation entries, Playground defines communication according to a separately declared configuration of logical connections. A logical connection from x in one application to y in another application, establishes a relationship from output at x to input at y.

Although the above relationships suggest strong similarities between the formal I/O automaton model and the Playground programming model, there are three important semantic differences. The first distinction is that while shared actions in an I/O automaton composition occur simultaneously and atomically, there is necessarily some physical delay between the output at one Playground application and the corresponding input at other Playground applications. Therefore, one can think of each automaton representing a Playground application as interacting with another automaton that represents the Playground run-time system. Thus, while the output of one application is shared atomically with the run-time system, the corresponding receive occurs later as an output from the system that is shared with the receiving application. This explicit introduction of the message delivery system as a separate automaton is, in fact, is the way most distributed systems are modeled using the I/O automaton model, and it is the technique we will be using to describe the formal semantics of Playground.

The second important semantic distinction is that while the sharing of actions in a system of I/O automaton is *statically defined* by the signatures of the composed automata, the configuration of a Playground system is *dynamically changing* under external control. As users change the logical configuration of a running Playground system, the communication pattern among the applications changes accordingly. To describe this dynamically changing communication pattern in I/O automaton model, we let the implementation of our explicit run-time system contain state

information about the dynamically changing logical connection structure. With this configuration information, the system can direct output from each application to the appropriate destination applications. Thus, although the set of actions shared between each application and the run-time system is static, the actual events that occur in the execution depend upon the logical configuration known to the run-time system.

The third distinction concerns a subtle but important difference between the input actions of I/O automata and input in the Playground programming model. Input actions of an I/O automaton are always enabled and can execute at any time. In Playground, however, there are two exceptions to this in order to accomodate atomic steps and reaction functions. Both are handled automatically by the run-time system, and we will formally model both of them in the same way.

When a Playground application is inside of an atomic step, multiple presentation entries may be written. At the conclusion of the atomic step, the updated presentation entries are each output before any input is processed. One could model this as a single, more complex, action that updates several presentation entries and is shared with the run-time system. Another possibility is to consider only a restricted set of executions in which the run-time system respects special output actions from the each application marking the beginning and end of its atomic steps. The run-time system buffers messages for the application and forwards them as input only when the application signals it is is "ready," between atomic steps.

Because we want reaction functions to appear atomic as far as the active control can tell, no input may occur while a reaction function is executing. By itself, this is not a departure from the I/O automaton model since input actions also occur atomically. However, because reaction functions are allowed to update presentation entries as part of their execution, output may result. In contrast to this, recall that the I/O automaton model has a strict separation of input and output actions; no action may represent both input and output.[2] One can avoid this semantic difference by writing Playground applications in which reaction functions update only local variables and do not update the presentation. However, we have found that allowing reaction functions to update the presentation provides significant expressive power for applications programmers. Therefore, rather than rule out the possibility of reaction functions performing output, we model each reaction function as an input action followed by a (possibly empty) series of output actions. Just as in the case of atomic steps, the run-time system buffers input from the environment during reaction functions and a "ready" action is used to signal the end of each reaction function.

4 Centralized Specification

A centralized view of the Playground system is shown in Figure 1. The logical connections among the applications are dynamically configured by the users through interaction with the system. Of course, users also interact with the applications, but this interaction is outside the scope of the run-time system, so we do not model it. We model each application as an I/O automaton whose actions correspond to publishing and unpublishing data in the application presentation, writing to published data and experiencing external updates to published data. The run time system delivers messages to their ultimate destination according to the configuration of the logical connections.

Before presenting the formal specification of the system, we define a few useful terms. The first two definitions will be used to define safety and liveness conditions for the I/O automata that model the various components of the Playground system.

[2]Otherwise, the fact that shared actions occur simultaneously and atomically in the I/O automaton model would result in the possibility of specifying infinite work to be done in finite time. However, this is not a danger in the Playground model because communication is asynchronous.

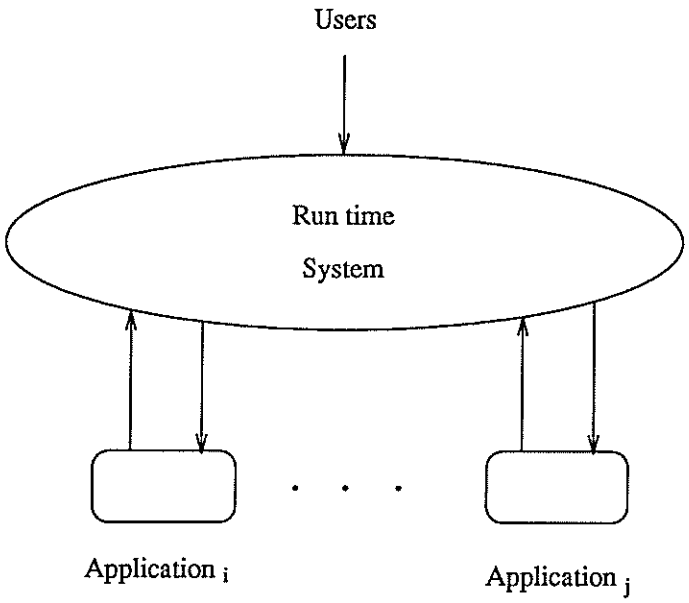

Figure 1: A Centralized Playground System

An automaton A is a *transducer* for the action sets Π_1 and Π_2, denoted $\Pi_1 \overset{A}{\leadsto} \Pi_2$, iff

1. the argument types of all the actions in Π_1 and Π_2 are the same, and
2. if β is a prefix of a schedule of A, the sequence of argument values in $\beta|\Pi_2$ is a prefix of the sequence of argument values in $\beta|\Pi_1$.

Moreover, an automaton A is a *live transducer* for the action sets Π_1 and Π_2, denoted $\Pi_1 \overset{A}{\mapsto} \Pi_2$, iff

1. $\Pi_1 \overset{A}{\leadsto} \Pi_2$, and
2. if β is a schedule of A, then the sequence of argument values in $\beta \mid \Pi_2$ is the sequence of argument values in $\beta|\Pi_1$.

When describing sets Π_1 and Π_2 for use in the above definitions, we will use an action name without arguments as a shorthand for the set of all actions with that name.

For convenience in discussing systems of applications with presentations, we let $\mathcal{P}$ be a set of presentation variable names and $\mathcal{I}$ be a totally ordered set of application names. We let the letters *x, y and z* denote arbitrary elements of $\mathcal{P}$ and the letters *i, j* and *k* denote arbitrary elements of $\mathcal{I}$. Thus x_i denotes the presentation item x of application A_i. Also, for each element of $x \in \mathcal{P}$ we let $type(x)$ denote the set of possible values held by the variable named x.

Finally, since queues are a common abstraction in message passing systems, we define four operations on queues. If q is an arbitrary queue, then

1. $enq(q, t)$ appends item t to q,
2. $t \leftarrow deq(q)$ removes the item at the head of q, or $\perp$ if q is empty, and assigns it to t,
3. $head(q)$ is the value of the head of q, or $\perp$ if q is empty, and
4. $extract(t, q)$ removes the first occurrence t from q, or returns $\perp$ if there is no such item t in q.

4.1 Applications

A Playground application is modeled as an I/O automaton with actions to publish and unpublish presentation data items and to write and read published variables. For flexibility, we intentionally allow the behavior of an application to be arbitrary, except for a few simple restrictions. We specify only the schedule module of the application, as we are not concerned with the internal state. An application schedule module A_i has the following action signature:

Input Actions:
$Update_i(x, v)$ where $v \in type(x)$

Output Actions:
$Publish_i(x)$
$UnPublish_i(x)$
$Write_i(x, v)$ where $v \in type(x)$
$Ready_i$

For an arbitrary sequence α of actions, we say that x_i is *locally published after* α iff there exists a $Publish_i(x)$ action in α and no later $Unpublish_i(x)$ action occurs in α. We say that α is *application well-formed* iff for every $UnPublish_i(x)$ action in α, there exists a unique preceding $Publish_i(x)$ action.

We can now define the set of allowable behaviors of an application. Let α be an arbitrary sequence of actions in $sig(A_i)$. Then $\alpha \in behs(A_i)$ iff

1. A_i preserves application well-formedness in α, and
2. if α is application well-formed, then
   (a) (Safety) For all prefixes β of α, if β ends with $Write_i(x, v)$, then x_i is locally published after β.
   (b) (Liveness) At least one $Ready_i$ occurs in α. Also, if an $Update_i(x, v)$ or $Write_i(x, v)$ occurs in α, then a $Ready_i$ occurs later in α.

The safety property specifies that updates can be sent out only for published variables. The liveness condition specifies that the application should allow data to arrive infinitely often. Therefore, the only liveness constraint we specify on a Playground application is that it perform a $Ready_i$ action when it is prepared to accept incoming updates. If $Update_i(x, v)$ and $Write_i(x, v)$ actions occur infinitely often, then so must $Ready_i$ actions. Multiple writes can occur before a ready action, but an application cannot indefinitely block updates from the environment.

The reason for the $Ready_i$ action is to model atomic steps and reactive control as implemented in Playground. Since reaction functions run atomically and may write to presentation variables, we use the $Ready_i$ action to signal the end of both reaction functions and atomic steps. Note, however, that this interaction is handled automatically by the Playground library used to construct Playground applications. Playground programmers do not send explicit $Ready_i$ signals to the veneer. Instead, the same effect is achieved in the library by allowing updates to arrive whenever the active control accesses (reads or writes) the presentation, and preventing updates from occurring during atomic steps and reaction functions.

4.2 The Playground Schedule Module (P)

The Playground run time system manages the configuration of user applications. It transports data according to the dynamically changing configuration of the applications, accepts requests from the applications to publish and unpublish presentation data items, and processes configuration requests by the user.

The following schedule module specifies the safety and liveness properties of the Playground system. In Section 5, we provide a distributed implementation of this schedule module.

The Playground schedule module P has the following action signature:

Input Actions:
- $Publish_i(x)$
- $Unpublish_i(x)$
- $Req\_Connect(x_i, y_j)$
- $Req\_Disconnect(x_i, y_j)$
- $Write_i(x, v)$ where $v \in type(x)$
- $Ready_i$

Output Actions:
- $Confirm\_Connect(x_i, y_j)$
- $Confirm\_Disconnect(x_i, y_j)$
- $Update_i(x, v)$ where $v \in type(x)$
- $Published(x_i)$
- $Unpublished(x_i)$

The $Publish_i(x)$ and $Unpublish_i(x)$ actions accept publish/unpublish requests from application A_i. We distinguish multiple publishes of a data item. If a data item is unpublished and republished, then we model it here with a different variable name. $Req\_Connect(x_i, y_j)$ and $Req\_Disconnect(x_i, y_j)$ actions accept requests to create and delete logical connections between published data items, respectively. The system informs its environment that the connection has been made successfully by the $Confirm\_Connect(x_i, y_j)$ output action. The system confirms that a connection has been broken by the $Confirm\_Disconnect(x_i, y_j)$ action. Updates to Playground data items are accepted by the $Write_i(x, v)$ action. If the Playground data item has been published and is connected to some data item, then the system outputs this update to the destination j by the $Update_j(x, v)$ output action. Recall that the application must perform a $Ready_i$ action whenever it becomes ready to accept an update to its published data items.

For convenience in reasoning about executions, we define a tagging scheme for the message system actions. Let α be a sequence of actions of the message system. Let α_T be an arbitrary sequence of tags, where one tag is associated with each event in α. We say that the labeling α_T is *consistent* iff

1. no two $Write_i$ events have the same tag,
2. if a $Write_i(x, v)$ event and a $Update_j(y, w)$ event have the same tag then
   a) $v = w$, and
   b) the $Write_i$ event occurs before the $Update_j$ event in α.

Two events with the same tag are considered *corresponding* events. Note that these tags do not exist in the implementation. They are omnisciently assigned in this discussion for the sole purpose of reasoning about action sequences.

The following terms will be used to define the set of allowable behaviors of the Playground system.

For an arbitrary sequence α of actions in $sig(P)$, we say that x_i is *published after* α iff there exists a $Published(x_i)$ action in α and no later $Unpublished(x_i)$ action occurs in α. Let $\alpha = \beta\pi\gamma$. If π is a $Write_i(x, v)$ or $Update_i(x, v)$ event and γ contains a $Ready_i$ event, then we say that α is *application-live.*

Let α be an arbitrary sequence of actions $\in sig(P)$. We say that α is *system well-formed* iff it is application well-formed and, for sequences β and γ such that $\alpha = \beta\pi\gamma$,

1. if π is a $Req\_Connect(x_i, y_j)$ or $Req\_Disconnect(x_i, y_j)$ event then x_i and y_j are published after β,

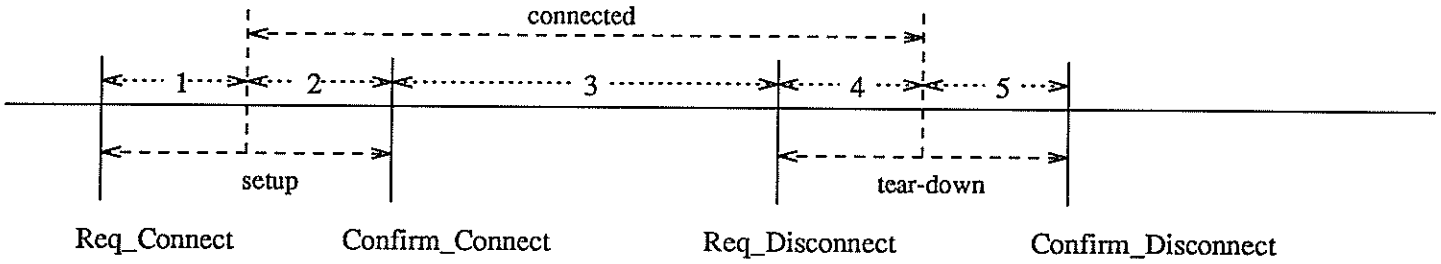

Figure 2: Sample execution showing setup, tear-down and connected intervals

Consider an execution sequence as shown in Figure 2. The execution subsequence between a $Req\_Connect(x_i, y_j)$ and $Confirm\_Connect(x_i, y_j)$ event is the connection *setup* interval. The execution subsequence between a $Req\_Disconnect(x_i, y_j)$ and $Confirm\_Disconnect(x_i, y_j)$ event is the connection *tear-down* interval. The *connected interval* for (x_i, y_j) is the period in which updates to x_i are propagated to y_j. The start of the connected interval occurs within the setup interval while the end of the connected interval occurs within the tear-down interval. Therefore, the user is guaranteed that the interval (3) between the $Confirm\_Connect(x_i, y_j)$ and the $Req\_Disconnect(x_i, y_j)$ events is fully contained by the connected interval. The connected interval also ends as soon as either of the endpoints of the link is unpublished.

If an update to the presentation data item x_i occurs before the $Request\_Connect(x_i, y_j)$ event, then the update will not be sent to y_j. If y_j is unpublished, then once an update for y_j is dropped, all subsequent updates to it are also dropped. This implies that not only are updates delivered in order, but that the sequence of updates cannot have gaps within a given connected interval.

We now define the set of allowable behaviors of the system. Let α be an arbitrary sequence of actions $\in sig(P)$. Then $\alpha \in behs(P)$ iff

1. P preserves well-formedness in α.

2. (Safety) If α is well-formed, then there exists a consistent labeling such that

(a) $Req\_Connect(x_i, y_j) \overset{P}{\rightsquigarrow} Confirm\_Connect(x_i, y_j)$,
$Req\_Disconnect(x_i, y_j) \overset{P}{\rightsquigarrow} Confirm\_Disconnect(x_i, y_j)$, and

(b) if there exists an $Update_j$ event in α, then there exists a corresponding $Write_i$ event in α.

(c) $Publish_i(x) \overset{P}{\rightsquigarrow} Published(x_i)$,
$Unpublish_i(x) \overset{P}{\rightsquigarrow} Unpublished(x_i)$,

(d) If a $Write_i(x, v_1)$ precedes $Write_i(x, v_2)$ in α and both the events occur in a connected interval of (x_i, y_j) and the corresponding $Update_j(y, v_2)$ event occurs in α, then it is preceded by the corresponding $Update_j(y, v_1)$ occurs earlier in α.

3. (Liveness) If α is well-formed and application-live, then there exists a consistent labeling such that

(a) $Publish_i(x) \overset{P}{\mapsto} Published(x_i)$,
$Unpublish_i(x) \overset{P}{\mapsto} Unpublished(x_i)$,

(b) if a $Req\_Connect(x_i, y_j)$ occurs in α, then either
   i. a later $Confirm\_Connect(x_i, y_j)$ event occurs, or
   ii. a later $Unpublished(x_i)$ or $Unpublished(y_j)$ event occurs.

(c) if a $Req\_Disconnect(x_i, y_j)$ occurs in α, then either
   i. a later $Confirm\_Disconnect(x_i, y_j)$ event occurs, or
   ii. a later $Unpublished(x_i)$ or $Unpublished(y_j)$ event occurs.

(d) if a $Write_i(x, v)$ event occurs in the connected interval for (x_i, y_j), then either
   i. a later corresponding $Update_j(y, v)$ occurs, or
   ii. a later $Unpublished(y_j)$ event occurs.

The safety conditions state that a $Confirm\_Connect(x_i, y_j)$ event cannot occur unless a $Req\_Connect(x_i, y_j)$ event has occurred, and that a $Confirm\_Disconnect(x_i, y_j)$ event cannot occur unless a $Req\_Disconnect(x_i, y_j)$ event has occurred. Similarly, a $Published(x_i)$ event is preceded by a $Publish_i(x)$ event, while an $Unpublished(x_i)$ event is preceded by a $Unpublish_i(x)$ event. An $Update_j$ event cannot occur unless the corresponding $Write_i$ event has already occurred. Also, we guarantee FIFO ordering between application pairs. Updates from one application to the other are received in the order sent.

The liveness conditions state a $Publish_i(x)$ event is eventually followed by a $Published(x_i)$ event, while an $Unpublish_i(x)$ event is eventually followed by an $Unpublished(x_i)$ event. Also, a $Req\_Connect(x_i, y_j)$ event is eventually followed by a $Confirm\_Connect(x_i, y_j)$ event unless either x_i or y_j is unpublished. Similarly, a $Req\_Disconnect(x_i, y_j)$ is eventually followed by a $Confirm\_Disconnect(x_i, y_j)$ event unless either x_i or y_j is unpublished. Also if a $Write_i(x, v)$ event occurs while x_i are y_j are connected, then eventually an $Update_j(y, v)$ occurs unless y_j is unpublished.

Semantics of message delivery: Referring to Figure 2 we consider each interval marked on the figure as a separate case.
Case 1: If an update to the presentation data item x_i occurs before the $Request\_Connect(x_i, y_j)$ event then it will not be sent to y_j.

Case 2: If the update occurs after the $Request\_Connect(x_i, y_j)$ event but before the $Confirm\_Connect(x_i, y_j)$ event then we cannot tell externally whether the update occurred in

interval 1 or interval 2. If an earlier update in this interval was delivered, then we know that the link has been created and that subsequent updates will also be delivered. If it occurred during interval 1 then the update will not be transmitted to y_j. On the other hand, if the update occurred during interval 2, then it will be sent to y_j.

Case 3: If the update occurs during interval 3 (after the *Confirm_Connect*(x_i, y_j) event and before the *Request_Disconnect*(x_i, y_j) event) then it will definitely be sent to y_j.

Case 4: If the update occurs after the *Request_Disconnect*(x_i, y_j) and before *Confirm_Disconnect*(x_i, y_j) event then it may or may not be sent to y_j. Externally, we have no way of knowing whether the update occurred during interval 4 (will be sent) or during interval 5 (the update will not be sent).

Case 5: If the update occurs after the *Confirm_Disconnect*(x_i, y_j) event then it will definitely not be sent to y_j.

5 Distributed Implementation

The Playground system is designed as a software library, run-time system, and programming environment that insulates the applications programmer from the operating system and the network. The current version supports applications written in C++ on top of the Solaris operating system with sockets as the underlying communication mechanism.

An overview of the distributed implementation is shown in Figure 3. The Playground system has been implemented in a distributed manner by a collection of veneers, one per application, communicating among each other via the network. In the implementation, the veneer is compiled into the application. Here, we model it as a separate automaton since it is distinct from the part of the application written by the application programmer. The veneer embodies a library of publishable data types and a communication protocol subsystem. As shown in Figure 4, all communication into or out of an application goes through the veneer. The veneer acts as an intermediary between the message system and the application, hiding the message formatting and transmission details from the modules. The veneer handles encoding data prior to transmission, decoding data, atomic steps, and registration and invocation of reactive control.

The veneers also communicate among themselves and with a special module called the connection manager. The connection manager has the presentation information of every application (only type information, not data) and the configuration of the applications. The presentation information is sent to the connection manager by the veneers associated with each application. Requests by users to establish and drop logical connections are handled by the connection manager, which also checks access rights and type compatibility. The connection manager informs the endpoint veneers of a new logical connection to set up the connection. The connection manager is not a communication bottleneck because once it has set up a connection, communication on that connection is handled by the individual veneers.

We model each veneer, the connection manager and the network as separate I/O automata. These automata are composed to form the Playground system. The veneers communicate among themselves and with the connection manager by passing messages. A *message*

is a $(MesgType, Payload)$ pair. The message types are Publish($\mathcal{P}$), Unpublish($\mathcal{U}$), AddLink($\mathcal{A}$), RemoveLink($\mathcal{R}$), Handshake($\mathcal{H}$), Goodbye($\mathcal{G}$), ConfirmConnect($\mathcal{CC}$), ConfirmDisconnect($\mathcal{CD}$), ClearLink($\mathcal{CL}$) and Data($\mathcal{D}$). For brevity, we will use the term *link* to refer to a logical connection.

For message types $\mathcal{P}$ and $\mathcal{U}$, the payload consists of a variable name x_i. For message types $\mathcal{A}$, $\mathcal{R}$, $\mathcal{H}$, $\mathcal{G}$, $\mathcal{CC}$ and $\mathcal{CD}$, the payload consists of a pair (x_i, y_j) of variable names. Lastly, for data messages, ($\mathcal{D}$) the payload is of the form (x, v), where $v \in type(x)$. For message m, we use *type(m)* to refer to the message type and *payload(m)* to refer to the message payload.

We generalize the notions of transducer and live transducer defined in Section 4. In this section we overload the notations $\rightsquigarrow$ and $\mapsto$ by allowing the form of the action arguments to be specified. For example, we say $Write_i(x, v) \overset{A}{\rightsquigarrow} SendMesg_i(j, (\mathcal{D}, (x_i, v)))$ to mean that A is a transducer defined as before that issues output actions of the form $SendMesg_i(j, (\mathcal{D}, (x_i, v)))$ in response to inputs of the form $Write_i(x, v)$. In other words, we allow a syntactic relationship between the input and output action arguments to be specified, rather than require that the arguments be identical.

5.1 The Veneer

5.1.1 The Veneer Schedule Module (V_i)

The veneer schedule module has the following action signature:

Input Actions:
- $Publish_i(x)$
- $Unpublish_i(x)$
- $Write_i(x, v)$ where $v \in type(x)$
- $Ready_i$
- $ReceiveMesg_i(M)$

Internal Actions:
- $ProcessDataMesg_i$
- $DiscardInput_i(x, v)$
- $DiscardOutput_i(M)$ where $type(M) = \mathcal{D}$

Output Actions:
- $Update_i(x, v)$ where $v \in type(x)$
- $SendMesg_i(cm, M)$
- $SendMesg_i(j, M)$

The veneer accepts requests to publish/unpublish Playground data items by the $Publish_i$ and $Unpublish_i$ actions respectively. Updates made by applications to Playground data items are accepted by the $Write_i$ action. External updates to Playground data items at the local application occur via the $Update_i$ action. The application must perform the $Ready_i$ action whenever it is ready to accept updates to its published data items. The veneer communicates with other veneers via the network automaton by using the $SendMesg_i$ and $ReceiveMesg_i$ actions. The internal action $DiscardInput_i$ is responsible for removing from the InBox updates to those variables that have been unpublished in the period between the time that the update arrived and the time that it gets to the head of the InBox. The action $DiscardOutput_i$ removes updates from the OutBox if there exists no outgoing link. The link could be deleted by the time the update gets to the head of the OutBox. The other internal actions prepare data and information messages that are sent to peer applications.

Let α be a sequence of actions in $sig(V)$. If β is a subsequence of α beginning with a $SendMesg_i(cm,(\mathcal{CC},(x_i,y_j)))$ or $SendMesg_j(cm,(\mathcal{CC},(x_i,y_j)))$ action and no $SendMesg_i(cm,(\mathcal{CD},(x_i,y_j)))$ or $SendMesg_j(cm,(\mathcal{CD},(x_i,y_j)))$ occurs in β, then we say that β is a *connected interval* for (x_i,y_j).

We now define the set of allowable behaviors of the veneer. Let α be an arbitrary sequence of actions in $sig(V)$. Then $\alpha \in behs(V)$ iff

1. V_i preserves application well-formedness in α,
2. (Safety) if α is application well-formed, then
   (a) $Publish_i(x) \overset{V_i}{\rightsquigarrow} SendMesg_i(cm,(\mathcal{P},x_i))$, $Unpublish_i(x) \overset{V_i}{\rightsquigarrow} SendMesg_i(cm,(\mathcal{U},x_i))$,
   (b) $Write_i(x,v) \overset{V_i}{\rightsquigarrow} SendMesg_i(j,(\mathcal{D},(y_j,v)))$ if $Write_i(x,v)$ occurs while x_i and y_j are connected, $ReceiveMesg_i(\mathcal{D},(x_i,v)) \overset{V_i}{\rightsquigarrow} Update_i(x,v)$,
   (c) $ReceiveMesg_i(\mathcal{A},(x_i,y_j)) \overset{V_i}{\rightsquigarrow} SendMesg_i(j,(\mathcal{H},(x_i,y_j)))$,
   $ReceiveMesg_i(\mathcal{R},(x_i,y_j)) \overset{V_i}{\rightsquigarrow} SendMesg_i(j,(\mathcal{G},(x_i,y_j)))$,
   (d) $ReceiveMesg_i(\mathcal{H},(x_i,y_j)) \overset{V_i}{\rightsquigarrow} SendMesg_i(cm,(\mathcal{CC},(x_i,y_j)))$,
   $ReceiveMesg_i(\mathcal{G},(x_i,y_j)) \overset{V_i}{\rightsquigarrow} SendMesg_i(cm,(\mathcal{CD},(x_i,y_j)))$,
3. (Liveness) if α is application well-formed and application-live, then
   (a) $Publish_i(x) \overset{V_i}{\mapsto} SendMesg_i(cm,(\mathcal{P},(x_i,y_j)))$, $Unpublish_i(x) \overset{V_i}{\mapsto} SendMesg_i(cm,(\mathcal{U},(x_i,y_j)))$,
   (b) $ReceiveMesg_i(\mathcal{A},(x_i,y_j)) \overset{V_i}{\mapsto} SendMesg_i(j,(\mathcal{H},(x_i,y_j)))$,
   $ReceiveMesg_i(\mathcal{R},(x_i,y_j)) \overset{V_i}{\mapsto} SendMesg_i(j,(\mathcal{G},(x_i,y_j)))$,
   (c) for $j < i$, then
   $ReceiveMesg_i(\mathcal{H},(x_i,y_j)) \overset{V_i}{\mapsto} SendMesg_i(cm,(\mathcal{CC},(x_i,y_j)))$,
   $ReceiveMesg_i(\mathcal{G},(x_i,y_j)) \overset{V_i}{\mapsto} SendMesg_i(cm,(\mathcal{CD},(x_i,y_j)))$,
   (d) $Write_i(x,v) \overset{V_i}{\mapsto} SendMesg_i(j,(\mathcal{D},(y_j,v)))$, if the *Write* action occurs in a connected interval for (x_i,y_j).
   (e) if a $ReceiveMesg_i(\mathcal{D},(x_i,v)$ event occurs in α, then eventually either an $Update_i(x,v)$ event or an $Unpublish_i(x)$ event occurs.

Informally, the safety conditions say that if a publish or unpublish type message is sent, then the publish/unpublish action must have occurred. Also, if a data message is sent out, then a write event must have occurred. If an update event occurs, then a data message must have been received. The last two safety conditions pertain to the handshake mechanism for making and deleting connections. If a handshake message is sent out, then an addlink message must have been received. Similarly, if a goodbye message is sent out, then a removelink message must have been received. If a confirm connect message is sent out, then a handshake must have been received. If a confirm disconnect message is sent out, then a goodbye message must have already been received.

The liveness conditions say that if a publish or unpublish event occurs, then eventually a publish/unpublish message is sent to the connection manager. Receipt of an addlink message is eventually followed by the sending of a handshake message. Receipt of a droplink message is eventually followed by the sending of a goodbye message. Receipt of a handshake message

eventually leads to a confirm connect message being sent to the connection manager by the node with the higher id. If a write action occurs when two variables are connected then eventually the data message will be sent. If a data message is received at an application, then, provided that the application is live, eventually the update event occurs or the variable is unpublished.

5.1.2 The Veneer Automaton

The state of the veneer automaton consists of the following components. *InBox* and *OutBox* are queues that store messages. These messages can be data messages or requests to publish/unpublish variables. The message format has been described earlier. *PSet* is the set of published variables and *LSet* is the set of logical connections to and from the application. *OutLinkSet* is the set of outgoing links from a variable. *IncompleteLinkSet* is the set of links being created for which the handshake process is still unfinished. The *send_mode* flag is set when the veneer is ready to send updates for a variable on all links from that variable. The *ready* flag controls communication between the veneer and the application. Initially, all the above sets and queues are empty and the flags are set to FALSE.

The action signature is the same as that of the schedule module. The transition relation is shown in Figure 5. The *Publish* and *Unpublish* actions accept publish/unpublish requests from the application. They simply queue a message onto the *OutBox*. The publish or unpublish event is actually processed only when this message gets to the head of the *OutBox*. This ensures that all updates that occur before a publish or unpublish event are processed before the publish/unpublish event is processed. The veneer accepts messages from the network by the *ReceiveMesg* action and sends messages to the network by the $SendMesg_i(j, M)$ and the $SendMesg_i(cm, M)$ actions. The messages can be of the types described earlier.

Messages from the connection manager to add or delete a link initiate a handshake or tear-down protocol respectively. The connection manager is sent a message of type $\mathcal{CC}$ or $\mathcal{CD}$ as the case may be when the handshake terminates. The veneer uses a handshake process to create a link when it receives a message from the connection manager. The connection manager sends addlink messages to both endpoints of the link . The veneer with the higher id is the one to initiate the handshake. It enqueues a $(\mathcal{H}, (x_i, y_j))$ message onto the *OutBox* when it receives a $(\mathcal{A}, (x_i, y_j))$ message. It also adds (x_i, y_j) to *IncompleteLinkSet*. The veneer with the lower id waits for the handshake message to arrive from the other endpoint. It adds (x_i, y_j) to *IncompleteLinkSet* when it receives a handshake or addlink message, whichever comes first. When this veneer has received both handshake and addlink messages, it adds the link to its *LSet* and sends a handshake message back to the other endpoint to indicate that it has made the link. The veneer which initiated the handshake waits until a handshake message returns, and then adds the link to its *LSet*. It then enqueues a $(\mathcal{CC}, (x_i, y_j))$ message onto the it OutBox. At the time that a link is added to *LSet*, it is removed from *IncompleteLinkSet*.

The veneer uses a tear-down process to delete a link. This is to ensure that all messages in transit on the link and those already in the *InBox* are delivered before the link is deleted. The veneer enqueues a goodbye message $((\mathcal{H}, (x_i, y_j)))$ onto the *OutBox* when it receives a $(\mathcal{R}, (x_i, y_j))$ message from the connection manager. When this message gets to the head of the *OutBox*, the veneer removes (x_i, y_j) from *LSet* and sends the message to $application_j$. Note that the connection manager sends a droplink message only to the upstream side of the link. When the veneer receives a goodbye message, it removes the link from its linkset and enqueues a $(\mathcal{CD}, (x_i, y_j))$ message onto the *OutBox*. Since the network guarantees FIFO delivery of messages, when the veneer receives a goodbye message, all messages ahead of it must have been delivered.

Updates to Playground data items are accepted by the $Write_i(x, v)$ action which enqueues

the update onto the *OutBox* if the concerned variable is published and there exists at least one outward link from that variable. When the data message gets to the head of the *OutBox*, the veneer prepares to send the update to all veneers that are connected to this variable. The internal action *ProcessDataMesg* initializes the *OutLinkSet* with the set of outgoing connections from the variable. It also sets *send_mode* to TRUE. This puts the veneer in a state in which it can transmit a message onto all the links in the *OutLinkSet*. The action $SendMesg_i(j,(\mathcal{D},(y_j,v)))$ is enabled when *send_mode* is TRUE, $(x_i,y_j) \in$ *OutLinkSet* and $(\mathcal{D},(x_i,v))$ is at the head of the *OutBox*. It sends the message out and removes (x_i,y_j) from *OutLinkSet*. When *OutLinkSet* becomes empty, the message at the head of the OutBox is dequeued and *send_mode* is set to FALSE. It may happen that a message may reach the head of the *OutBox* and there may be no outgoing link from that variable. This happens when a goodbye message was in the *OutBox* ahead of the data update. In such a situation, the $DiscardInput_i$) action dequeues the message from the *OutBox*.

A data message is queued onto the *InBox*. An update is sent to the application by the $Update_i$) action. The $Update_i(x,v)$ action is enabled when (x_i,v) is at the head of the *InBox* and the ready flag is TRUE. It provides the update to the application and dequeues the data message from the *InBox*. The $Ready_i$ action simply sets the ready flag to TRUE. If x_i is $\notin$ *PSet* and (x_i,v) is at the head of the InBox then the $DiscardInput_i$ action is enabled. This action simply drops an update as the presentation item it was destined for does not exist any longer.

If the message received is of type $\mathcal{CL}$, then the veneer removes the connection (which may still not have been completed) without performing the handshake. The connection manager sends such a message when one of the endpoints of the connection is unpublished. In this case, the tear-down protocol is not carried out and no acknowledgment is sent to the connection manager once the link is deleted.

The set of locally controlled actions is partitioned into 2 equivalence classes. The set of $DiscardInput_i$ and $Update_i$ actions are in the same class while all the rest are in the other equivalence class.

We assume that all queues are of infinite length. Obviously, this is not achievable in practice. Since the veneer is compiled with the application, the application would block if any of the queues are full. The system has been implemented in such a way that it will not block if the queues become full. If the *OutBox* becomes empty, then the veneer drains the *OutBox* even it is in the middle of enqueueing a message onto the *OutBox*. If the veneer is reading from the network and the *InBox* becomes full, then the veneer hands control to the part that interfaces with the application to supply updates and tries to empty the *InBox*.

5.2 The Connection Manager

The connection manager is itself a Playground application. It acts as a repository for presentation and link information. Information on the presentations of all the applications and the logical connections between them is stored here. The presentation information is sent to the connection manager by the veneers associated with each module. In Playground, a graphical front-end is provided to display this information for a user and to allow direct manipulation of the configuration. The connection manager also acts as an "introduction service," managing the set-up of logical connections. The connection manager checks requests to add and delete links for type validity and protection. If the connection can be created, it informs the concerned veneers which then run a handshake protocol to set up the link. The node which initiates the handshake informs the connection manager when the connection has been created. Deleting connections is handled similarly. Once a connection has been created, the veneers can communicate directly over the network. The data does not go through the connection manager. We first describe the connection

manager schedule module, then present an automaton that implements this specification. Note that we do not model protection information here.

5.2.1 The Connection Manager Schedule Module (CM)

The connection manager has the following action signature:

Input Actions:
$Req\_Connect(x_i, y_j)$
$Req\_Disconnect(x_i, y_j)$
$ReceiveMesg_{cm}(M)$

Internal Actions:
$ProcessDeferredRequest(x_i, y_j)$

Output Actions:
$SendMesg_{cm}(i, M)$
$Confirm\_Connect(x_i, y_j)$
$Confirm\_Disconnect(x_i, y_j)$
$Published(x_i)$
$Unpublished(x_i)$

The connection manager communicates with the network by the $SendMesg_{cm}$ and $ReceiveMesg_{cm}$ actions. Requests to add and delete logical connections are made by the user to the connection manager by the *Req_Connect* and *Req_Disconnect* actions. The connection manager confirms that a connection has been made or deleted by the *Confirm_Connect* and *Confirm_Disconnect* actions respectively. The connection manager delays requests to add or delete a connection until a prior request for that connection is confirmed, so that at most one connection request for a given pair of variables is in progress. However, multiple connection requests for different variables may proceed simultaneously, including those with a common endpoint. The internal action *ProcessDeferredRequest* processes a deferred request for the connection (x_i, y_j) when the current request for that connection is confirmed. The connection manager informs user that a data item has been published by the *Published* action. It informs the user of unpublish events by the *Unpublished* action. For convenience in reasoning about executions of the connection manager, we use the action name $publish(x_i)$ as a shorthand for the action $ReceiveMesg_{cm}(P, x_i)$. Similarly, we use $unpublish(x_i)$ as a shorthand for the action $ReceiveMesg_{cm}(U, x_i)$.

Consider an arbitrary sequence of actions $\alpha = \alpha_1\alpha_2$ from $sig(CM)$. We say that x_i and y_j are *connected after* α iff

1. x_i and y_j are published after α,
2. $Confirm\_Connect(x_i, y_j)$ is the last event in α_1,
3. no $Req\_Disconnect(x_i, y_j)$ occurs in α_2, and
4. no $unpublish(x_i)$ or $unpublish(y_j)$ event occurs in α_2.

Let α be an arbitrary sequence of actions from *sig(CM)*. We say that α is *connection manager well-formed* iff

1. for every $Unpublished(x_i)$ in α, there exists a unique, preceding $Published(x_i)$ action,

2. for all prefixes β of α, if β ends with a $Req\_Connect(x_i, y_j)$ or $Req\_Disconnect(x_i, y_j)$ event then x_i and y_j are published after β, and

3. if the last action of β is a $Req\_Disconnect(x_i, y_j)$ event, then $\beta = \beta_1\beta_2$ such that the last event of β_1 is a $Req\_Connect(x_i, y_j)$ event and no $Req\_Connect(x_i, y_j)$, $Req\_Disconnect(x_i, y_j)$, $Unpublished(x_i)$ or $Unpublished(y_j)$ event occurs in β_2.

The tagging scheme is extended for reasoning about behaviors of the connection manager. Let α be a sequence of actions of the connection manager. Let α_T be an arbitrary sequence of tags, where one tag is associated with each event in α.

We say that the labeling α_T is *connection manager consistent* iff

1. if $SendMesg_{cm}(j, M)$ and $SendMesg_{cm}(k, N)$ have the same tag, then $j \neq k$ or $M \neq N$

2. if a $SendMesg_{cm}(j, (\mathcal{A}, (x_i, y_j)))$ event and a $Req\_Connect(x_i, y_j)$ event have the same tag then, the $Req\_Connect(x_i, y_j)$ event occurs before the $SendMesg_{cm}(j, (\mathcal{A}, (x_i, y_j)))$ event in α.

3. if a $SendMesg_{cm}(j, (\mathcal{R}, (x_i, y_j)))$ event and a $Req\_Disconnect(x_i, y_j)$ event have the same tag then, the $Req\_Disconnect(x_i, y_j)$ event occurs before the $SendMesg_{cm}(j, (\mathcal{R}, (x_i, y_j)))$ event in α.

4. no two *Req_Connect* or *Req_Disconnect* events have the same tag.

Two events with the same tag are considered *corresponding* events.

We define the set of allowable behaviors of the connection manager. Let α be an arbitrary sequence of actions $\in sig(CM)$. Then $\alpha \in behs(CM)$ iff

1. CM preserves well-formedness in α,

2. (Safety) if α is connection manager well-formed, then there exists a connection manager consistent labeling α_T such that

   (a) $Req\_Connect(x_i, y_j) \overset{CM}{\rightsquigarrow} Confirm\_Connect(x_i, y_j)$,
   $Req\_Disconnect(x_i, y_j) \overset{CM}{\rightsquigarrow} Confirm\_Disconnect(x_i, y_j)$,

   (b) For every $SendMesg_{cm}(j, (\mathcal{A}, (x_i, y_j)))$ or $SendMesg_{cm}(i, (\mathcal{A}, (x_i, y_j)))$ action in α, there exists a corresponding $Req\_Connect(x_i, y_j)$ action that precedes it in α. Similarly, for every $SendMesg_{cm}(i, (\mathcal{R}, (x_i, y_j)))$ action in α, there exists a corresponding $Req\_Disconnect(x_i, y_j)$ action that precedes the send event in α.

   (c) $ReceiveMesg_{cm}(\mathcal{CC}, (x_i, y_j)) \overset{CM}{\rightsquigarrow} Confirm\_Connect(x_i, y_j)$,
   $ReceiveMesg_{cm}(\mathcal{CD}, (x_i, y_j)) \overset{CM}{\rightsquigarrow} Confirm\_Disconnect(x_i, y_j)$,

   (d) $ReceiveMesg_{cm}(\mathcal{P}, x_i) \overset{CM}{\rightsquigarrow} Published(x_i)$,
   $ReceiveMesg_{cm}(\mathcal{U}, x_i) \overset{CM}{\rightsquigarrow} Unpublished(x_i)$,

3. (Liveness) if α is connection manager well-formed, then there exists a connection manager consistent labeling α_T such that

   (a) $Req\_Connect(x_i, y_j) \overset{CM}{\mapsto} Confirm\_Connect(x_i, y_j)$,
   $Req\_Disconnect(x_i, y_j) \overset{CM}{\mapsto} Confirm\_Disconnect(x_i, y_j)$,

(b) For each $Req\_Connect(x_i, y_j)$ action in α, a $SendMesg_{cm}(i, (\mathcal{A}, (x_i, y_j)))$ action occurs later in α as does a $SendMesg_{cm}(j, (\mathcal{A}, (x_i, y_j)))$ action. Similarly, for each $Req\_Disconnect(x_i, y_j)$ action in α, a $SendMesg_{cm}(i, (\mathcal{R}, (x_i, y_j)))$ action occurs later in α action, where $type(M) = \mathcal{R}$.

(c) $ReceiveMesg_{cm}(\mathcal{CC}, (x_i, y_j)) \overset{CM}{\mapsto} Confirm\_Connect(x_i, y_j)$,
$ReceiveMesg_{cm}(\mathcal{CD}, (x_i, y_j)) \overset{CM}{\mapsto} Confirm\_Disconnect(x_i, y_j)$,

(d) $ReceiveMesg_{cm}(\mathcal{P}, x_i) \overset{CM}{\mapsto} Published(x_i)$,
$ReceiveMesg_{cm}(\mathcal{U}, x_i) \overset{CM}{\mapsto} Unpublished(x_i)$,

The safety conditions state that for a confirm connect (confirm disconnect) event to occur, the corresponding request connect (request disconnect) event must have occurred and a confirm connect (confirm disconnect) message must have been received earlier in the execution. Also, if an add link or drop link request message is sent,then the corresponding request must have occurred prior to the send. For a published (unpublished) event to occur, a publish (unpublish) message must have been received earlier.

The liveness conditions state that if a request connect (request disconnect) event or a confirm connect (confirm disconnect) message is received, then the confirm connect (confirm disconnect) event will eventually occur. Also if a request connect occurs, then eventually an addlink messages will be sent to both endpoints. If a request disconnect event occurs then a drop link message will be sent to the upstream veneer. If a published (unpublished) message is received, then a publish (unpublish) event will eventually occur in the execution.

5.2.2 The Connection Manager Automaton

The state of the connection manager consists of the following data structures. *PSet* is the set of published variables, *LSet* is the set of logical connections, and *Requests* is a queue which stores requests to add/remove logical connections. The requests are stored as ordered pairs of destinations and messages. For example, a request to add a link between the Playground data items x_i and y_j is stored as $(i, (A, (x_i, y_j)))$. Requests still being processed are stored in the *UnconfirmedRequests* set. In addition. there are two queues: *DeferredRequests* and *ConfirmedRequests*. Confirmed requests are queued onto the *ConfirmedRequests* queue. Any request to create or delete a link which is still being processed is queued onto the *DeferredRequests* queue to be processed later. Requests stored in the above two queues and the *UnconfirmedRequests* set are of the type $(\mathcal{A}, (x_i, y_j))$ for create connection requests and are of the type $(\mathcal{R}, (x_i, y_j))$ for delete requests. There is a *PresentationUpdateQueue* that stores incoming publish/unpublish messages. Initially, all the queues and sets are empty. Publish/unpublish messages from veneers are stored in the *PresentationUpdateQueue*.

The transition relation for the connection manager automaton is shown in Figure 8. When the $Req\_Connect(x_i, y_j)$ action occurs, the connection manager checks to see if there is a request for connection (x_i, y_j) in the unconfirmed set or the confirmed queue. If so, it queues the request onto the *DeferredRequests* queue. If not, it puts the request into the *UnconfirmedRequests* set and also queues add link messages (of type $\mathcal{A}$) onto the *Requests* queue. The requests are duly sent out by the $SendMesg_{cm}(i, M)$ action. When the connection manager receives an acknowledgment from a veneer informing it that both endpoints have created the link, it removes the connection from *UnconfirmedRequests* and queues it onto the *ConfirmedRequests* queue. The $Confirm\_Connect(x_i, y_j)$ action then removes the connection from *ConfirmedRequests* and inserts it

into the *LSet*. A *Req_Disconnect*(x_i, y_j) action is handled analogously. The only difference is that the drop link message is sent only to the upstream end of the link, not to both endpoints.

The connection manager receives messages from the veneers informing it of changes in their presentations. The publish/unpublish messages are enqueued onto the *PresentationUpdateQueue*. then When the message gets to the head of the queue, the The *Published* action is enabled when the publish message gets to the head of the queue. This action adds the published variable to the *PSet*. When the unpublish action reaches the head of *PresentationUpdateQueue*, the *Unpublished* action is enabled. This action removes the variable from the *PSet*. All links in the *LSet* to and from that variable are removed. The endpoints are sent messages of type $\mathcal{CL}$ to make them to delete the link. Any request involving the unpublished variable that has not yet been confirmed is removed from the queue or set it is in. For all add link requests that are being processed, clear link messages are sent to the concerned veneers.

All *SendMesg*$_{cm}(i, M)$ actions are in the same equivalence class. The *ProcessDeferredRequest*(x_i, y_j) actions are in a separate equivalence class. The *Confirm_Connect*(x_i, y_j) and *Confirm_Disconnect*(x_i, y_j) actions are in the same equivalence class. The *Published*(x_i) and *Unpublished*(x_i) actions are in the same equivalence class.

5.2.3 The Network Schedule Module N

The network transfers messages among veneers and the connection manager. Our model is that of a lossless network. We only specify a schedule module for the network as we are not interested in the mechanics of the transfer, and are concerned solely with the behavior of the network. The network delivers messages in pairwise FIFO order. The network schedule module N has the following action signature:

Input Actions:
SendMesg$_i(j, M)$ where j is the destination of the message M and $i \in \mathrm{I} \cup \{cm\}$
Output Actions:
ReceiveMesg$_i(M)$ $i \in \mathrm{I} \cup \{cm\}$

The network accepts a message by the *SendMesg*$_i(j, M)$ action and delivers the message M to the recipient j by the *ReceiveMesg*$_i(M)$ action.

Let α be an arbitrary sequence of actions in $sig(N)$. Then $\alpha \in behs(N)$ iff, there exists a protocol consistent labeling α_T such that

1. (Safety) For every *ReceiveMesg*$_i(M)$ action in α, there exists a corresponding *SendMesg*$_i(j, M)$ action preceding it α.

2. (Safety) If there exists a *SendMesg*$_i(j, M)$ action with tag a, followed by *SendMesg*$_i(j, M')$ action by the same application with tag b, then the corresponding *ReceiveMesg*$_j(M)$ action with tag a precedes the *ReceiveMesg*$_j(M)$ action with tag b.

3. (Liveness) For each *SendMesg*$_i(j, M)$ action in α there exists a corresponding *ReceiveMesg*$_j(M)$ action.

The first safety condition states that the network cannot create messages. The second safety condition states that messages between every pair of applications will be delivered in FIFO order. The liveness condition states that the network must deliver messages eventually.

6 Proof Sketch

We compose the veneers, connection manager automaton and any network automaton that implements the network schedule module and hide the message sending and receive actions to obtain the implementation automaton I. We present a proof sketch that our implementation(I) solves the centralized Playground schedule module presented in Section 4.2. We must show that $fairbehs(I) \subseteq fairbehs(P)$. Let β be a fair behavior of I. It is clear that the components of the distributed implementation P preserve system well-formedness, as the only actions in the well-formedness condition are input actions performed by the user. Thus β is well-formed. Let us prove that $fairbehs(I)$ satisfy the safety properties of $fairbehs(P)$.

Lemma 1: (Safety) Let α be an execution in $fairbehs(P)$. If there exists an $Update_j$ event in α, then there exists a corresponding $Write_i$ event in α.

We denote the $Write_i(M)$ event with tag t by $Write_i(M_t)$. The network guarantees that a sequence of messages sent by application A_i to A_j is delivered to in order without any messages being lost. A message accepted by the network and not yet delivered to its destination is considered to be "in transit." The *in transit sequence* for the link x_i, y_j is the sequence of messages in transit from application A_i to A_j. Consider the sequence S of messages formed by concatenating $OutBox_i$, the in transit sequence for the link x_i, y_j and $InBox_i$. The number of messages M_t in the sequence S denoted by $M_t(S)$ is less than or equal to 1 in all reachable states.

This can be proved by induction on the length of the execution by considering the effect each individual action of the compositon has on the assertion. We provide an operational reasoning here to give the intuition behind the conclusion that the above safety property is satisfied. One of the preconditions for the $Update_i(M_t)$ action is that (M_t) is at the head of $InBox_i$. From the proof of the above assertion, we know that some $Write_i(x, v)$ event must have occurred to set this condition.

Lemma 2: (Safety) Let α be an execution in $fairbehs(P)$. If $Write_i(x, v_1)$ precedes $Write_i(x, v_2)$ in α and both the events occur in the connected interval of (x_i, y_j). If the $Update_j(y, v_2)$ event occurs in α, then it is preceded by the corresponding $Update_j(y, v_1)$.

Recall that if a data item is published twice, the two names are considered distinct. One of the preconditions for the $Update_j$ event is that y_j is in the *Pset*. Once y_j is unpublished, the condition $y_j \in Pset$ becomes false and stays false forever. Thus if y_j is unpublished before $Update_j(y, v_1)$ occurs, then all updates still in transit or in $InBox_j$ will be discarded. The network preserves the message ordering sequence. Thus, updates are queued in the InBox in the order that they were sent. If the update events occur, they occur in the same order as the write events.

Lemma 3: (Safety)

Let α be an execution in $fairbehs(P)$. $Req\_Connect(x_i, y_j) \overset{P}{\rightsquigarrow} Confirm\_Connect(x_i, y_j)$ in α.

The precondition of the $Confirm\_Connect(x_i, y_j)$ action is that $(\mathcal{A}, (x_i, y_j))$ is the at the head of the *ConfirmedRequests* queue. Such a request can be queued onto the *ConfirmedRequests* queue, only when a confirm connect message is received. This can be seen from the actions of the connection manager. For such a message to arrive, the veneers must have performed the handshake process in response to addlink message from the connection manager. These messages are enqueued onto the *Requests* queue only when a $Req\_Connect(x_i, y_j)$ action occurs.

Lemma 4: (Safety)

Let α be an execution in $fairbehs(P).Req\_Disconnect(x_i, y_j) \overset{P}{\rightsquigarrow} Confirm\_Disconnect(x_i, y_j)$ in α.

The proof is on the same lines as the previous one. Only the message type is different and the endpoint veneers conduct a tear-down process by sending goodbye messages instead of going through a handshake process.

We now have to prove that *fairbehs*(I) satisfy the liveness properties of *fairbehs*(P).

Lemma 5: (Liveness) Let α be an execution in *fairbehs*(P). If a $Write_i(x, v)$ event occurs in α in the connected interval for (x_i, y_j), then either

1. a later corresponding $Update_j(y, v)$ occurs, or
2. a later $Unpublish_j(y)$ event occurs.

Once a $Write_i(x, v)$ event occurs during the connected interval for (x_i, y_j), the update will be delivered to application A_j by the network and sender liveness properties. The fairness properties of the sender and receiver automata ensure that messages in the queues will eventually be delivered to their destinations. By the liveness of the application automaton A_j and the veneer automaton V_j, the update reaches the head of $InBox_j$. If $y_j \in Pset$, the $Update_j(y, v)$ action is enabled and remains continuously enabled until an $Unpublish_j(y_j)$ action occurs. If an unpublish event has occurred, then $y_j \notin Pset$. Further, this condition remains true for all subsequent states. In this case, the $Discard_j(y_j, v)$ action is enabled and remains continuously enabled. The above liveness property follows from the fairness of the veneer and application automata.

Lemma 6: (Liveness) Let α be an execution in *fairbehs*(P). If a $Req\_Connect(x_i, y_j)$ occurs in α, then either

1. a later $Confirm\_Connect(x_i, y_j)$ event occurs, or
2. a later $Unpublish_i(x)$ or $Unpublish_j(y)$ event occurs.

A $Req\_Connect(x_i, y_j)$ event causes $(i, (\mathcal{A}, (x_i, y_j)))$ and $(j, (\mathcal{A}, (x_i, y_j)))$ requests to be queued onto the *Requests* queue. Eventually, these requests are sent and received by the destination veneers. These veneers go through a handshake protocol. That the handshake process terminates is easily seen. The veneer with the higher id sends a handshake message to the veneer at the other end of the link. The veneer with the lower id sends a handshake message when it has received both the addlink and the handshake messages. The initiator of the handshake sends a confirm connect message to the connection manager when it gets back a handshake message. The confirm connect message arrives at the connection manager and $(\mathcal{A}, (x_i, y_j))$ is removed from the *UnconfirmedRequests* set and queued onto the *ConfirmedRequests* queue. Eventually, this request moves to the head of the queue and enables the precondition of the $Confirm\_Connect(x_i, y_j)$ action.

Lemma 7: (Liveness) Let α be an execution in *fairbehs*(P). If a $Req\_Disconnect(x_i, y_j)$ occurs in α, then either

1. a later $Confirm\_Disconnect(x_i, y_j)$ event occurs, or
2. a later $Unpublish_i(x)$ or $Unpublish_j(y)$ event occurs.

The proof is similar to that of Lemma 6.

Theorem: *fairbehs*(I) $\subseteq$ *fairbehs*(P). The proof follows immediately from the lemmas.

7 Conclusions and Future Work

We have provided a formal specification of the semantics of the Playground programming model and we have presented a formal description of a distributed implementation of that model. The system, as currently implemented, provides separation of computation from communication and dynamic reconfiguration.

The precise description of the semantics of the programming model provides a basis for comparison to the semantics of other models, as well as a foundation for understanding possible enhancements to the programming model. The formal I/O automaton implementation description provides an abstract formulation of the Playground run-time system, abstracting away details that are specific to the operating system and programming language.

We advocate the use of a formal semantics for real systems not only to provide a better understanding of the system for its users, but also as an important part of the design process. It provides a deeper understanding of the system and its implementation, and in some cases may lead to useful design changes in the system itself. In fact, our analysis has convinced us to change the design to enable us to reason more easily about the executions of the system. The changes include exchanging goodbye messages when a link is to be torn down, and having the connection manager buffer requests for a link until all prior requests for that link are confirmed.

We have presented a proof sketch to give the reader a sense that our distributed implementation does indeed solve the specification, without getting involved in all the details that an actual proof would entail. However, one possible direction for future work would be a complete assertional proof that that the formal I/O automaton description of the system implements the formal specification. Following this, one would want to conduct a detailed study of the actual C++ implementation to verify that it indeed implements the formal I/O automaton description. Ideally, this could be carried out on a component by component basis, verifying that each major software component of the run-time system implements its corresponding I/O automaton in the formal description.

The current implementation uses asynchronous message delivery with pairwise FIFO ordering. We plan to implement causal delivery of messages. Consider three applications A_i, A_j and A_k, where there exists a logical connection from A_j and A_k to A_i and also from A_j to A_k. If there is a causal relationship between messages generated by applications A_j and A_k, then messages at application A_i will be delivered in that order. The veneer and the application need not be aware that such an ordering scheme is in place. The protocols can piggyback sequencing information in every message. The receiving protocols could use this information to decide whether to accept the message, or to buffer it till all the messages preceding it in the causal sequence are received. Causal delivery of messages is discussed in [1, 14].

References

[1] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[2] Jerome R. Cox, Jr., Mike Gaddis, and Jonathan S. Turner. Project Zeus: Design of a broadband network and its application on a university campus. *IEEE Network*, pages 20–30, March 1993.

[3] Kenneth J. Goldman et al. http://www.cs.wustl.edu/cs/playground.

[4] Kenneth J. Goldman. Data interfaces as support for module migration. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994. Position paper, as invited panelist.

[5] Kenneth J. Goldman, Bala Swaminathan, Michael D. Anderson, T. Paul McCartney, and Ramachandran Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering.* to appear.

[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[7] Bengt Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the 4th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1985.

[8] Bengt Jonsson. Compositional specification and verification of distributed systems. Technical Report SICS/R-90/90010, Swedish Institute of Computer Science, October 1990.

[9] Bengt Jonsson. Simulations between specification of distributed systems. In *Proceedings of the 2nd International Conference on Concurrency Theory, LNCS 527*, pages 346–360. Springer–Verlag, August 1991.

[10] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3), 1989.

[11] T. Paul McCartney and Kenneth Goldman. Visual specification of interprocess and intraprocess communication. In *Proceedings of the 10th International Symposium on Visual Languages*, pages 80–87, October 1994.

[12] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[13] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, October 1989.

[14] Bala Swaminathan and Kenneth J. Goldman. An incremental distributed algorithm for computing biconnected components. In *Proceedings of the 8th International Workshop on Distributed Algorithms, WDAG '94.* Springer-Verlag, 1994.

[15] Bala Swaminathan and Kenneth J. Goldman. Dynamic reconfiguration with I/O abstraction. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Computing*, October 1995. To appear.

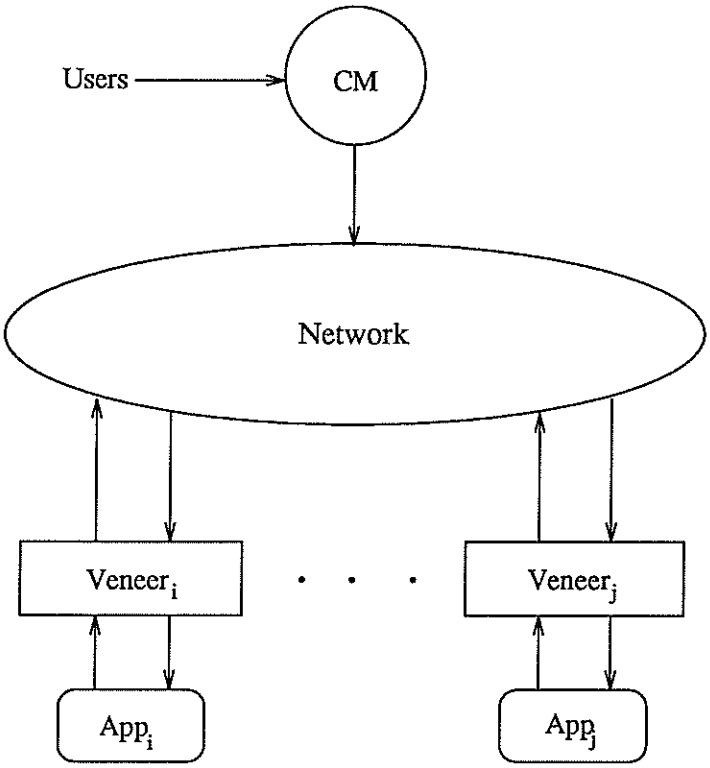

Figure 3: Distributed Implementation of Playground Message System

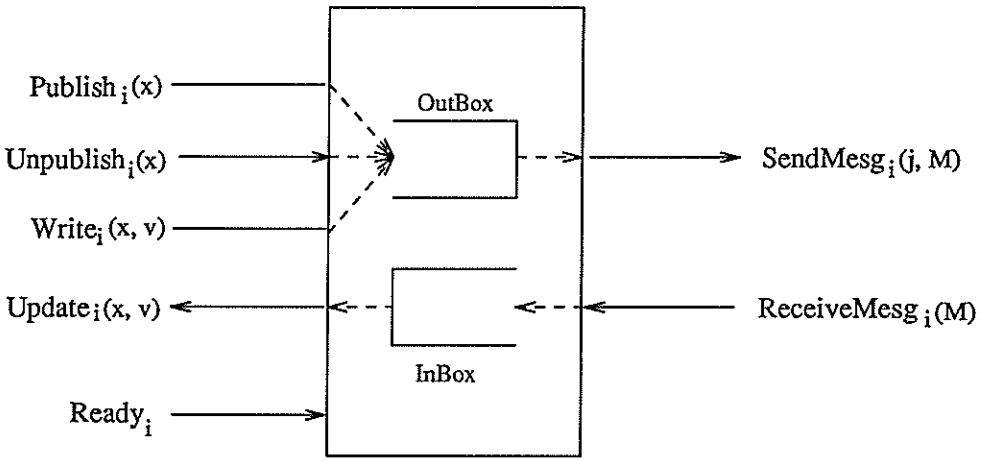

Figure 4: Veneer

Input Actions:

- $Publish_i(x)$
  Effect: $enq(OutBox, (\mathcal{P}, x_i))$
- $Unpublish_i(x)$
  Effect: $enq(OutBox, (\mathcal{U}, x_i))$
- $Write_i(x, v)$
  Effect: if $x_i \in PSet \wedge \exists\ (x_i, y_j) \in LSet$
  $enq(OutBox, (\mathcal{D}, (x_i, v)))$
- $Ready_i$
  Effect: $ready_i \leftarrow$ TRUE

Internal Actions:

- $ProcessDataMesg_i$
  Precondition: $head(OutBox) = (\mathcal{D}, (x_i, v)) \wedge send\_mode = FALSE$
  Effect: $\forall (x_i, y_j) \in LSet$
  $OutLinkSet \leftarrow OutLinkSet \cup \{(x_i, y_j)\}$
  $send\_mode \leftarrow TRUE$
- $DiscardInput_i(x, v)$
  Precondition: $head(InBox) = (x, v) \wedge x_i \notin PSet$
  Effect: $deq(InBox)$
- $DiscardOutput_i(\mathcal{D}, (y_j, v))$
  Precondition: $head(OutBox) = (\mathcal{D}, (x_i, v)) \wedge (x_i, y_j) \notin LSet$
  Effect: $deq(OutBox)$

Output Actions:

- $Update_i(x, v)$
  Precondition: $(x, v) = head(InBox) \wedge x_i \in PSet$
  Effect: $deq(InBox)$
- $SendMesg_i(cm, M)$
  Precondition: $M = head(OutBox) \wedge type(M) \in \{\mathcal{P}, \mathcal{U}, \mathcal{CC}, \mathcal{CD}\}$
  Effect: if $M = (\mathcal{P}, x_i)$
  $PSet \leftarrow PSet \cup \{x\}$
  if $M = (\mathcal{U}, x_i)$
  $PSet \leftarrow PSet - \{x\}$
  $deq(OutBox)$
- $SendMesg_i(j, (\mathcal{D}, (y_j, v)))$
  Precondition: $head(OutBox) = (\mathcal{D}, (x_i, v)) \wedge (x_i, y_j) \in OutLinkSet \wedge send\_mode = TRUE$
  Effect: $OutLinkSet \leftarrow OutLinkSet - \{(x_i, y_j)\}$
  if $OutLinkSet = \emptyset$
  $deq(OutBox)$
  $send\_mode \leftarrow FALSE$

Figure 5: Veneer Transition Relation (the message types Publish, Unpublish, AddLink, RemoveLink, Handshake, Goodbye and Data are abbreviated by the first letter of the type name)

Output Actions:

- $SendMesg_i(j, (\mathcal{H}, (y_j, v)))$
  Precondition: $head(OutBox) = (\mathcal{H}, (y_j, v))$
  Effect: $deq(OutBox)$
- $SendMesg_i(j, (\mathcal{G}, (y_j, v)))$
  Precondition: $head(OutBox) = (\mathcal{G}, (y_j, v))$
  Effect: $LSet \leftarrow LSet - (x_i, y_j)$
  $deq(OutBox)$

Input Actions:

- $ReceiveMesg_i(\mathcal{D}, (x_i, v))$
  Effect: $enq(InBox, (x, v))$
- $ReceiveMesg_i(\mathcal{A}, (x_i, y_j))$
  Effect: if $x_i \in PSet$
  if $i > j$
  $IncompleteLinkSet \leftarrow IncompleteLinkSet \cup \{(x_i, y_j)\}$
  $enq(OutBox, (\mathcal{H}, (x_i, y_j)))$
  else
  if $(x_i, y_j) \in IncompleteLinkSet$
  $LSet \leftarrow LSet \cup \{(x_i, y_j)\}$
  $IncompleteLinkSet \leftarrow IncompleteLinkSet - \{(x_i, y_j)\}$
  $enq(OutBox, (\mathcal{H}, (x_i, y_j)))$
  else
  $IncompleteLinkSet \leftarrow IncompleteLinkSet \cup \{(x_i, y_j)\}$
- $ReceiveMesg_i(\mathcal{H}, (x_i, y_j))$
  Effect: if $x_i \in PSet$
  if $i > j$
  $LSet \leftarrow LSet \cup \{(x_i, y_j)\}$
  $IncompleteLinkSet \leftarrow IncompleteLinkSet - \{(x_i, y_j)\}$
  $enq(OutBox, (\mathcal{CC}, (x_i, y_j)))$
  else
  if $(x_i, y_j) \in IncompleteLinkSet$
  $LSet \leftarrow LSet \cup \{(x_i, y_j)\}$
  $IncompleteLinkSet \leftarrow IncompleteLinkSet - \{(x_i, y_j)\}$
  $enq(OutBox, (\mathcal{H}, (x_i, y_j)))$.
  else
  $IncompleteLinkSet \leftarrow IncompleteLinkSet \cup \{(x_i, y_j)\}$
- $ReceiveMesg_i(\mathcal{R}, (x_i, y_j))$
  Effect: $enq(OutBox, (\mathcal{G}, (x_i, y_j)))$
- $ReceiveMesg_i(\mathcal{G}, (y_j, x_i))$
  Effect: $LSet \leftarrow LSet - (x_i, y_j)$
  $enq(OutBox, (\mathcal{CD}, (y_j, x_i)))$
- $ReceiveMesg_i(\mathcal{CL}, (x_i, y_j))$
  Effect: $LSet \leftarrow LSet - (x_i, y_j)$
  $IncompleteLinkSet \leftarrow IncompleteLinkSet - \{(x_i, y_j)\}$

Figure 6: Veneer Transition Relation (the message types Publish, Unpublish, AddLink, RemoveLink, ClearLink, Handshake, Goodbye and Data are abbreviated by the first letter of the type name)

We denote the fact that a request for a particular link is being processed by $processing\_request(x_i, y_j)$. This means that $(\mathcal{A}, (x_i, y_j)) \in UnconfirmedRequests \vee (\mathcal{R}, (x_i, y_j)) \in UnconfirmedRequests \vee (\mathcal{A}, (x_i, y_j)) \in ConfirmedRequests \vee (\mathcal{R}, (x_i, y_j)) \in ConfirmedRequests$.

Input Actions:

- $Req\_Connect(x_i, y_j)$
  Effect: if $x_i \in PSet \wedge y_j \in PSet \wedge (x_i, y_j) \notin LSet$
    if $processing\_request(x_i, y_j)$
      $enq(DeferredRequests, (\mathcal{A}, (x_i, y_j)))$
    else
      $UnconfirmedRequests \leftarrow UnconfirmedRequests \cup \{(\mathcal{A}, (x_i, y_j))\}$
      $enq(Requests, (i, \mathcal{A}, (x_i, y_j)))$
      $enq(Requests, (j, \mathcal{A}, (x_i, y_j)))$
- $Req\_Disconnect(x_i, y_j)$
  Effect: if $(x_i, y_j) \in LSet$
    if $processing\_request(x_i, y_j)$
      $enq(DeferredRequests, (\mathcal{R}, (x_i, y_j)))$
    else
      $UnconfirmedRequests \leftarrow UnconfirmedRequests \cup \{(\mathcal{R}, (x_i, y_j))\}$
      $enq(Requests, (i, \mathcal{R}, (x_i, y_j)))$
- $ReceiveMesg_{cm}(M)$
  Effect: if $M = (\mathcal{P}, x_i)$
    $enq(PresentationUpdateQueue, (\mathcal{P}, x_i))$
  if $M = (\mathcal{U}, x_i)$
    $enq(PresentationUpdateQueue, (\mathcal{U}, x_i))$
- $ReceiveMesg_{cm}(\mathcal{CC}, (x_i, y_j))$
  Effect: $UnconfirmedRequests \leftarrow UnconfirmedRequests - \{(\mathcal{A}, (x_i, y_j))\}$
  $enq(ConfirmedRequests, (\mathcal{A}, (x_i, y_j)))$
- $ReceiveMesg_{cm}(\mathcal{CD}, (x_i, y_j))$
  Effect: $UnconfirmedRequests \leftarrow UnconfirmedRequests - \{(\mathcal{R}, (x_i, y_j))\}$
  $enq(ConfirmedRequests, (\mathcal{R}, (x_i, y_j)))$

Internal Actions:

- $ProcessDeferredRequest(x_i, y_j)$
  Precondition: if not $processing\_request(x_i, y_j)$
    $extract(request, DeferredRequests)$
    if $request = (\mathcal{A}, (x_i, y_j)) \wedge (x_i, y_j) \notin LSet \wedge s_i, y_j \in PSet$
      $enq(Requests, (i, \mathcal{A}, (x_i, y_j)))$
      $enq(Requests, (j, \mathcal{A}, (x_i, y_j)))$
      $UnconfirmedRequests \leftarrow UnconfirmedRequests \cup \{(\mathcal{A}, (x_i, y_j))\}$
    if $request = (\mathcal{R}, (x_i, y_j)) \wedge (x_i, y_j) \in LSet \wedge s_i, y_j \in PSet$
      $enq(Requests, (i, \mathcal{R}, (x_i, y_j)))$
      $UnconfirmedRequests \leftarrow UnconfirmedRequests \cup \{(\mathcal{R}, (x_i, y_j))\}$

Figure 7: Connection Manager Transition Relation

Output Actions:

- $SendMesg_{cm}(j, M)$
  Precondition: $head(Requests) = (j, M)$
  Effect: $deq(Requests)$

- $Confirm\_Connect(x_i, y_j)$
  Precondition: $head(ConfirmedRequests) = (\mathcal{A}, (x_i, y_j))$
  Effect: $LSet \leftarrow LSet \cup \{(x_i, y_j)\}$
  $deq(ConfirmedRequests)$

- $Confirm\_Disconnect(x_i, y_j)$
  Precondition: $head(ConfirmedRequests) = (\mathcal{R}, (x_i, y_j))$
  Effect: $LSet \leftarrow LSet - \{(x_i, y_j)\}$
  $deq(ConfirmedRequests)$

- $Published(x_i)$
  Precondition: $head(PresentationUpdateQueue) = (\mathcal{P}, x_i)$
  Effect: $PSet \leftarrow PSet \cup \{x_i\}$
  $deq(PresentationUpdateQueue)$

- $Unpublished(x_i)$
  Precondition: $head(PresentationUpdateQueue) = (\mathcal{U}, x_i)$
  Effect: $PSet \leftarrow PSet - x_i$
  We refer to a link involving x_i as (v_j, w_k) where $x_i = v_j \vee x_i = w_k$

  $\forall (v_j, w_k) \in LSet$
  $enq(Requests, (\mathcal{CL}, (v_j, w_k)))$
  $enq(Requests, (\mathcal{CL}, (w_k, v_j)))$
  $LSet \leftarrow LSet - \{v_j, w_k\}$

  $\forall (\mathcal{A}, (v_j, w_k)) \in UnconfirmedRequests$
  $enq(Requests, (\mathcal{CL}, (v_j, w_k)))$
  $enq(Requests, (\mathcal{CL}, (w_k, v_j)))$
  $UnconfirmedRequests \leftarrow UnconfirmedRequests - \{(\mathcal{A}, (v_j, w_k))\}$
  $\forall (\mathcal{R}, (v_j, w_k)) \in UnconfirmedRequests$
  $UnconfirmedRequests \leftarrow UnconfirmedRequests - \{(\mathcal{R}, (v_j, w_k))\}$

  $\forall (\mathcal{A}, (v_j, w_k))$ in $ConfirmedRequests$
  $enq(Requests, (\mathcal{CL}, (v_j, w_k)))$
  $enq(Requests, (\mathcal{CL}, (w_k, v_j)))$
  $extract(\mathcal{A}, (v_j, w_k))$
  $\forall (\mathcal{R}, (v_j, w_k)) \in ConfirmedRequests$
  $extract(\mathcal{R}, (v_j, w_k))$

  $\forall (\mathcal{A}, (v_j, w_k))$ in $DeferredRequests$
  $extract(\mathcal{A}, (v_j, w_k))$
  $\forall (\mathcal{R}, (v_j, w_k)) \in DeferredRequests$
  $extract(\mathcal{R}, (v_j, w_k))$

Figure 8: Connection Manager Transition Relation