Architectural Building Blocks for Plug-and-Play System Design

Shangzhu Wang, George S. Avrunin, Lori A. Clarke
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003

{shangzhu,avrunin,clarke}@cs.umass.edu

ABSTRACT

One of the distinguishing features of distributed systems is the importance of the interaction mechanisms that are used to define how the sequential components interact with each other. Given the complexity of the behavior that is being described and the large design space of various alternatives, choosing appropriate interaction mechanisms is difficult. In this paper, we propose a component-based specification approach that allows designers to experiment with alternative interaction semantics. Our approach is also integrated with design-time verification to provide feedback about the correctness of the overall system design. In this approach, connectors representing specific interaction semantics are composed from reusable building blocks. Standard communication interfaces for components are defined to reduce the impact of changing interactions on components' computations. The increased reusability of both components and connectors also allows savings at model-construction time for verification. We illustrate our approach by showing how the specification and verification of a small example could be done using the set of building blocks we have defined for message passing. We also show that this set of building blocks can be extended to describe a variety of semantics for other interaction mechanisms such as publish/subscribe and remote procedure call.

1. INTRODUCTION

One of the distinguishing features of distributed systems is the importance of the interaction mechanisms that are used to define how the sequential components interact with each other. Consequently, software architecture description languages typically separate the computational *components* of the system from the *connectors*, which describe the interactions among those components (e.g., [2,28,33,36]). Interaction mechanisms represent some of the most complex aspects of a system. It is the interaction mechanisms that primarily capture the non-determinism, interleavings, synchronization, and interprocess communication among components. These are all issues that can be particularly difficult to fully comprehend in terms of their impact on the overall system behavior.

As a result, it is often very difficult to design a distributed system with the desired component interactions. The large design space from which developers must select the appropriate interaction mechanisms adds to the difficulty. Choices range from shared-memory mechanisms, such as monitors and mutual exclusion locks, to distributed-memory mechanisms, such as message passing and event-based notification. Even for a single interaction mechanism type, there are usually many variations on how it could be structured.

Because of this complexity, design-time verification of distributed systems is particularly important. One would like to be able to propose a design, use verification to determine which important behavioral properties are not satisfied, and then modify and reevaluate the system design repeatedly until a satisfactory design is found. With component-based design, existing components are often used and glued together with connectors. In this mode of design, one would expect that the interaction mechanisms represented by the connectors would need to be reconsidered and fine-tuned several times during this design and design-time verification process, whereas the high-level design of the components would remain more stable. If using a finite-state verifier, such as Spin [25], SMV [27], LTSA [29], INCA [10], or FLAVERS [13], a model of each component and connector could be created separately and then the composite system model could be formed and used as the basis for verification.

With current design approaches, a major obstacle to the realization of this vision of component-based design is that the semantics of the interactions are deeply intertwined with the semantics of the components' computations. Changes in interactions usually require nontrivial changes in the components. As a result, it is often difficult and costly to modify the interactions without looking into the details of the components. Similarly, there is little model reuse during design-time verification.

In this paper, we propose a component-based approach that allows designers to experiment with alternative interaction semantics in a "plug-and-play" manner, while using the design-time verification to receive feedback about the correctness of the overall system design. The main contributions of our approach include the following:

- We provide a small set of *standard interfaces* by which components can communicate with each other through different connectors. The standard interfaces allow designers to change the semantics of interactions without having to make significant changes to the components.

- We separate connectors into *ports and channels* to represent different aspects of the semantics of connectors. This decomposition of connectors allows us to support a library of parameterizable and reusable building blocks that can be used to describe a variety of interaction mechanisms.

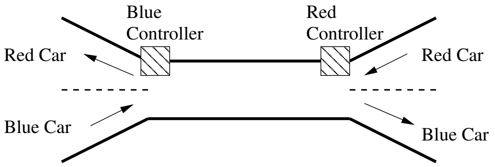


Figure 1: A single-lane bridge with two controllers

- The combined use of standard component interfaces and the reusable building blocks for connectors allows designers to be able to explore the design space and experiment with alternative choices of interaction semantics more easily.

- Our approach also facilitates design-time verification. With the increased reusability of components and connectors, one can expect that our approach will create savings in model-construction time during verification.

This paper presents the basic concepts as well as some preliminary results from an evaluation of our approach. Section 2 introduces an example that highlights some of the problems we are trying to address. Section 3 gives an overview of our "plug-and-play" approach. In Section 4, we show how the general approach can be applied to message passing. Specifically, a set of building blocks are defined based on the investigation of various message-passing semantics. In Section 5, using a small example, we illustrate how designers may experiment with alternative interaction semantics and achieve correct designs more easily using our approach. We also show that a small set of building blocks can be used to describe a variety of interaction semantics. Section 6 gives some preliminary results on extending this approach to other families of interaction mechanisms, namely publish/subscribe and remote procedure calls. Section 7 describes related work, followed by our conclusions and discussions of future work in Section 8.

2. AN ILLUSTRATIVE EXAMPLE

As an example, consider a bridge that is only wide enough to let through a single lane of traffic at a time [29]. An appropriate traffic control system is necessary to prevent crashes on the bridge. For this example, we assume that traffic control is provided by two controllers, one at each end of the bridge. Communication is allowed between controllers as well as between cars and controllers. To make the discussion easier to follow, we refer to cars entering the bridge from one end as the blue cars and refer to that end's controller as the blue controller; similarly the cars and controller on the other end are referred to as the red cars and the red controller, respectively, as shown in Figure 1.

We first introduce a very simple, and impractical, version of this system, called "exactly-N-cars-per-turn", that allows N cars to cross, starting with the blue end, before releasing control to the other end. Specifically, when it is the blue controller's turn, the blue controller counts exactly N blue cars entering the bridge and the red controller counts exactly N blue cars exiting the bridge. The two controllers then switch roles, and the red controller counts N red cars entering the bridge and the blue controller counts N red cars exiting the bridge. The above process then repeats. (We discuss some more realistic ways of controlling traffic in Section 5.)

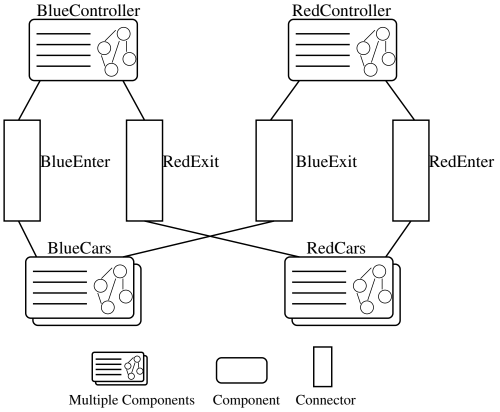
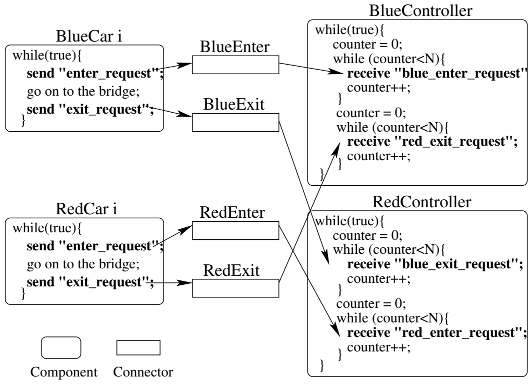


Figure 2: Architectural design of the single-lane bridge example



```
while(true){
   send "enter_request";
   go on to the bridge;
   send "exit_request";
}
```

```
while(true){
   send "enter_request";
   go on to the bridge;
   send "exit_request";
}
```

```
while(true){
   counter = 0;
   while (counter<N){
      receive "blue_enter_request"
      counter++;
   }
   counter = 0;
   while (counter<N){
      receive "red_exit_request";
      counter++;
   }
}
```

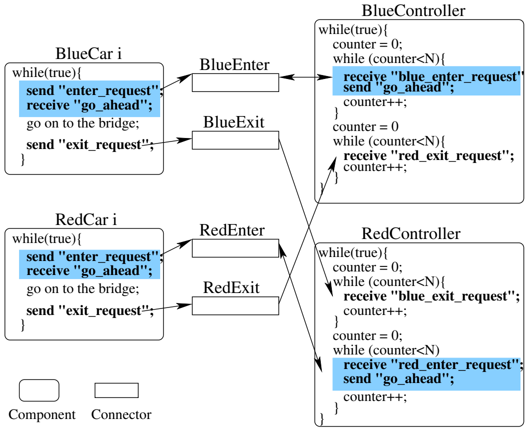
```
while(true){
   counter = 0;
   while (counter<N){
      receive "blue_exit_request";
      counter++;
   }
   counter = 0;
   while (counter<N){
      receive "red_enter_request";
      counter++;
   }
}
```

Figure 3: Some pseudocode design details for the single-lane bridge example

Figure 2 shows an informal design of the system. There are four different kinds of components: `BlueController`, `RedController`, one or more `BlueCar` components, and one or more `RedCar` components. Components communicate with each other through connectors: a `BlueEnter` connector between `BlueCar` components and the `BlueController` component, a `BlueExit` connector between the `BlueCar` components and `RedController` component, and similarly a `RedEnter` connector and a `RedExit` connector.

A more detailed description of the "exactly-N-cars-per-turn" version of the single-lane bridge problem is shown in Figure 3 where component interactions are described using a message passing interaction mechanism. In this figure, a car sends an `enter_request` message to the controller at the end of the bridge it wants to enter and then proceeds onto the bridge. When it exits the bridge, it notifies the controller at the exit end by sending an `exit_request` message. Controllers receive `enter_request` and `exit_request` messages, update their counters, and decide when to switch turns. Since there are multiple cars that communicate with each controller, messages are buffered in the connectors between car components and controller components.

Astute readers will notice that, according to the description in Figure 3, cars from different directions can be on the bridge at the same time, which could result in a crash. This is due to an erroneous design in the component inter-



```
while(true){
  send "enter_request";
  receive "go_ahead";
  go on to the bridge;
  send "exit_request";
}
```

```
while(true){
  counter = 0;
  while (counter<N){
    receive "blue_enter_request"
    send "go_ahead";
    counter++;
  }
  counter = 0
  while (counter<N){
    receive "red_exit_request";
    counter++;
  }
}
```

```
while(true){
  send "enter_request";
  receive "go_ahead";
  go on to the bridge;
  send "exit_request";
}
```

```
while(true){
  counter = 0;
  while (counter<N){
    receive "blue_exit_request";
    counter++;
  }
  counter = 0;
  while (counter<N)
    receive "red_enter_request";
    send "go_ahead";
    counter++;
  }
}
```

Figure 4: Modified design of the single-lane bridge

actions. With this design, a car sends an `enter_request` message and immediately goes onto the bridge without confirming that its request has been accepted by the controller. At this point of time however, the controller may still be waiting for and handling exit requests from the cars from the other direction and the enter request message from this car may still be in the buffer to be retrieved and handled. Therefore, a car can go on to the bridge while there are still cars traveling in the opposite direction. Obviously, what is needed here is synchronous communication between a car and its controller rather than asynchronous communication.

One way to fix this problem is to have the controller send a `go_ahead` message after receiving each enter request to authorize the car to enter the bridge. After sending the enter request, the car would wait for this acknowledgement before entering the the bridge, as shown in Figure 4 (the highlighted areas indicate the changes). These changes, involving both the car components and the controller components, effectively make the communication between them synchronous and solve the problem caused by the asynchronous communication. Notice, however, that the synchronization semantics of the interaction between the components is expressed in the computations of those components, rather than in a connector.

Although our example is quite simple, we can see that the semantics of the interaction mechanisms are not specified independently from other aspects of the system, but instead are spread among the connectors and the components. This is a trivial example, but it is easy to envision how the intertwined semantics of the connectors and components makes it more challenging to discover and correct errors in the design of more complex systems. Therefore, we prefer an approach that allows us to modify connectors and components more independently of each other.

3. THE "PLUG-AND-PLAY" APPROACH

As illustrated in the example above, changing from asynchronous message passing to synchronous message passing requires significant changes in the components, not just the connectors. In our approach, we introduce ports as part of the connectors to capture those different synchronization semantics. Components employ simple, standard interfaces that allow them to access different synchronization semantics by plugging into the appropriate type of port. Other

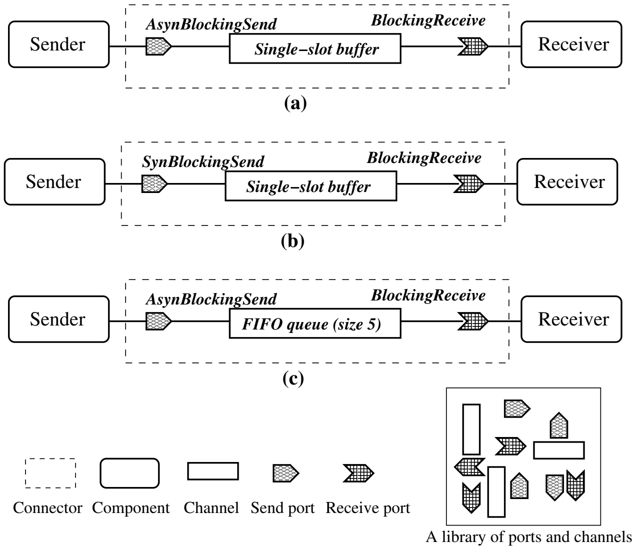


Figure 5: Constructing message passing connectors

semantics such as the buffering of the intermediate communication media are captured in channels, a separate part of the connectors. Components do not communicate with each other or with channels directly without going through ports.

To allow users to specify a wide range of interaction semantics, we support a library of parameterizable and reusable building blocks. One may select a subset of appropriate ports and channels from the library to construct a connector with the specific semantics. Figure 5(a) shows an example of how one might specify an asynchronous message passing communication between a pair of sender and receiver components.

The connector is composed of an asynchronous blocking send port, a blocking receive port, and a channel that buffers one message. Through this connector, the sender component sends a message without waiting for an acknowledgement from the receiver but blocks until the message is stored in the channel. The receiver component blocks until a message can be received. By replacing the asynchronous send port with a synchronous one from the library, the new connector in Figure 5(b) allows the sender to block not only until the message is stored in the channel but also until it has been delivered to the receiver. Similarly, channels can also be easily replaced. For example, a FIFO queue channel that holds up to 5 messages can be selected from the library to replace the single-slot buffer, when messages need to be buffered (as shown in Figure 5(c)). The replacement of channels can be done independently of the replacement of ports. The detailed semantics of these building blocks and the standard component interfaces are described in Section 4.2.

This kind of "plug-and-play" practice can be very efficient for designers experimenting with alternative interaction semantics. We have also found that our approach helps reduce the effort for repeated model construction when designers use design-time verification to check their design choices. In Section 4.3, we discuss techniques to facilitate design-time verification.

4. THE MESSAGE PASSING MECHANISM

To be more concrete, we describe how this approach could support a family of message passing semantics. Before describing the building blocks, standard interfaces, and model reuse that results during design-time verification, we first discuss some of the semantics of message passing that will need to be supported by the library of building blocks.

4.1 Dimensions

Although the fundamental message passing interactions are the two operations send and receive, there are a surprising number of variations in their semantics. Moreover, the implementation of the message passing infrastructure may also vary in terms of how messages are stored in the buffer, how messages are delivered, and what information is relayed to senders and receivers. Many languages, such as CSP [24], Occam [12], and Linda [7] incorporate message passing facilities. There are also message passing libraries such as MPI [37] and PVM [19]. Here we introduce a few dimensions that describe some of the most important aspects and common variations of the message passing semantics.

1. **Synchronous send ∼ Asynchronous send**

   With a *synchronous send*, a sender sends a message and blocks until it is notified that the message has been received by the receiver. With an *asynchronous send*, a sender sends a message and continues regardless of whether the message has been received by the receiver.

2. **Blocking send ∼ Nonblocking send ∼ Checking send**

   With a *blocking send*, a sender blocks until it is confirmed that the message has been accepted by the channel. With a *nonblocking send*, a sender sends a message to the channel and continues immediately regardless of whether the message can be accepted by the channel. With a *checking send*, a sender first checks with the channel. If it is notified that the channel currently cannot accept the message (e.g., when the buffer is full), it continues. Otherwise, it blocks until the message is stored in the channel.

3. **Blocking receive ∼ Nonblocking receive**

   With a *blocking receive*, a receiver blocks until a desired message is received successfully. With a *nonblocking receive*, a receiver may continue when no desired message can be received.

4. **Selective receive ∼ Nonselective receive**

   With *selective receive*, a receiver specifies the specific kind of messages it is interested in using certain selection criteria, such as the type of the messages or the sender of the messages. With *nonselective receive*, a receiver accepts any messages available in the buffer.

5. **Copy receive ∼ Remove receive**

   With *copy receive*, a copy of the message is delivered to the receiver, and the original message remains in the buffer. With *remove receive*, when a message is delivered to the receiver, it is removed from the buffer.

A number of other dimensions should be considered, such as the ordering of messages being stored and delivered, what happens when a message buffer becomes full, or whether a buffer is reliable or lossy. Since they are not the focus of this paper, we leave out the details here.

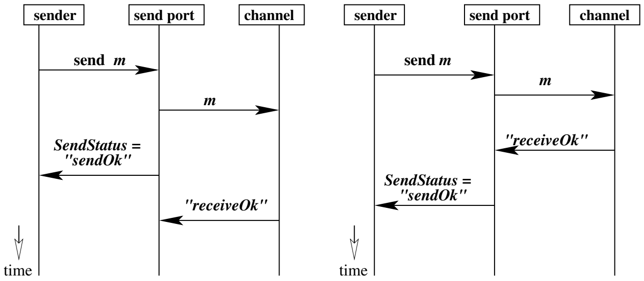
```
Message m,  SendStatus;
Component{
      ⋮
   send m;
   receive SendStatus;
      ⋮
}
```

(a) Component–send port protocol

```
Message m,  RecvRequest,  RecvStatus;
Component{
      ⋮
   send   RecvRequest;
   receive  RecvStatus;
   receive   m;
      ⋮
}
```

(b) Component–receive port protocol

Figure 6: Standard component interfaces



(a) Asynchronous Send

(b) Synchronous Send

Figure 7: Example scenarios of message passing interactions (using send ports)

4.2 Component Interfaces and Building Blocks

As mentioned in Section 3, we find it useful to decompose connectors into ports and channels that capture different parts of the message passing semantics. We construct a typical message passing connector from a channel to buffer messages, a send port to mediate between the sender component and the channel, and a receive port to mediate between the receiver component and the channel. To understand how these building blocks support the practice of "plug-and-play" design, we first introduce the standard interfaces between components and ports and illustrate how ports capture the synchronization semantics.

Figure 6(a) shows the standard interface that can be used by any component that wants to send a message through a connector to another component. We require the component to wait to receive a *SendStatus* message from the connector after sending a message. This interface is designed to work with connectors with different interaction semantics. For example, in the case of asynchronous message passing, the connector should return the *SendStatus* message to the sending component immediately, while for synchronous message passing, the connector should not return the *SendStatus* until the sender's message has been delivered.

Similarly, in Figure 6(b), a component that wishes to receive a message first sends a receive request to the port and waits for feedback (the *RecvStatus* message) on whether the requested message has been successfully retrieved. It then waits for a message from the receive port, either a real message (when the receive is successful) or an empty message (when the receive has failed).[1] It is up to the receiving component to check if a valid message has been received.

Using a notation similar to Message Sequence Charts, Fig-

[1] The empty message is used as a stub so that the interface remains the same in case of failure.

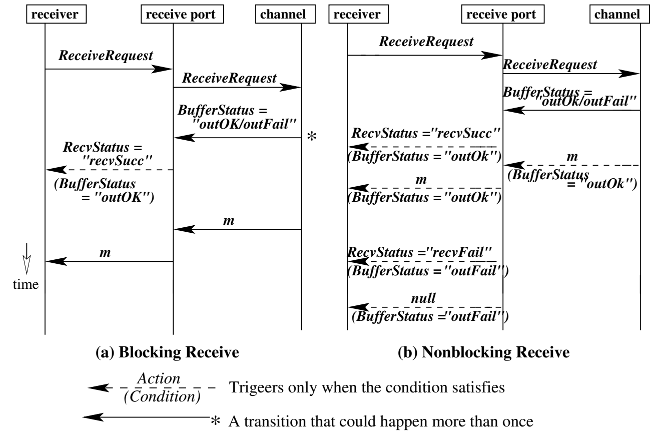


Figure 8: Example scenarios of message passing interactions (using receive ports)

ure 7 and Figure 8 illustrate how different ports may work with the same interfaces to implement different semantics. In the two parts of Figure 7, we see the same protocol being used between the sending component and the send port, and between the send port and the channel. (The figure only shows the cases in which the communication occurs without problems.) The component sends a message m and the send port returns the *SendStatus* message *sendOk*. Similarly, the send port passes the message m to the channel and, after delivering the message to a receive port, the channel sends the acknowledgment *receiveOk* to the send port. It is the send port that controls the relaying and interleaving of the internal events, and thus whether the message passing is synchronous or asynchronous.

In Figure 7(a), the asynchronous send port returns the *sendOk* message to the sending component without waiting for the channel to deliver the message and simply discards the *receiveOk* message from the channel when it arrives. The synchronous send port in Figure 7(b) waits to receive the *receiveOk* message from the channel before sending *sendOk* to the sending component, which is therefore blocked until after the message m is received. Neither the sending component nor the channel needs to know whether the connector is implementing synchronous or asynchronous message passing; the designer can swap one send port for the other to switch the semantics of the connector.

Similar considerations apply to the interface between a receiving component and a receive port. Figure 8(a) shows how a receive port implement the semantics of blocking receive. After forwarding the *ReceiveRequest* from the receiver to the channel, the port blocks until an *outOk* message is received from the channel indicating that the desired message is available. A *recvSucc* confirmation is then sent to the receiver followed the retrieved message. To implement the semantics of nonblocking receive (Figure 8(b)), a receive port may immediately return when the desired message is not available (*outFail*) by sending a *recvFail* message followed by an empty message to the receiving component.

In a fashion similar to that illustrated above, we are able to define a number of send and receive ports that can be used to implement a wide range of different message passing semantics, all designed to work with the standard component interface. We also define some of the most commonly used channels. Figure 9 shows a few examples of those ports and channels. Notice that when receiving a message from the channel, a receive port could either notify the channel to keep the message in the buffer or to remove it after the message is delivered to the port, which creates two versions (*copy/remove*) for each kind of receive port.

Send Port	**Asynchronous Nonblocking**	Waits for a message from the sender and sends a confirmation back immediately; the message may or may not be accepted and handled by the channel.
	Asynchronous Blocking	Waits for a message from the sender and sends a confirmation back AFTER the message has been accepted by the channel.
	Asynchronous Checking	Waits for a message from the sender and forwards it to the channel. If the message cannot be accepted by the channel, it returns and sends a notification to the sender. Otherwise, it blocks until the message is accepted and sends a confirmation back to the sender.
	Synchronous Blocking	Waits for a message from the sender and sends a confirmation back AFTER it is notified by the channel that the message has been received by the receiver.
	Synchronous Checking	Similar to "asynchronous checking send" except that when the message can be accepted by the channel, it blocks until the message is received by the receiver and then sends a confirmation back to the sender.
Receive Port	**Blocking (copy/remove)**	Waits for a "receive request" from the receiver and forwards it to the channel. It blocks until a desired message is retrieved from the channel and sends a confirmation to the receiver.
	Nonblocking (copy/remove)	Similar to "blocking receive" except that it returns immediately if no desired message can be retrieved currently. It then sends a notification along with an empty message to the receiver.
Channel	**1–slot buffer**	A buffer of size 1.
	FIFO queue	A FIFO queue of size N.
	Priority queue	A priority queue of size N.

Figure 9: Examples of message passing building blocks

The semantics of *selective/nonselective*, for example, can be specified as optional tags in the receive request messages sent by the receive components and acted upon by the channel. The way we define these message passing building blocks is preliminary and the set of building blocks is not yet meant to cover every aspect of message passing semantics. As we can see in the example in Section 5, however, these building blocks are useful in practice and can in fact express a wide range of message passing semantics.

The same interface and protocols can be used when we apply this approach to some other interaction mechanisms as shown in Section 6. In general, any outgoing communication will be mapped to sending a message and incoming communication will be mapped to receiving a message. In addition, as we show in Section 6, the library of message passing building blocks can also be used to describe a range of important semantics for other interaction mechanisms such as publish/subscribe and remote procedure calls. In fact, one of our long-term goals is to support a shared library of building blocks that are rich enough to describe a large variety of heterogeneous interaction mechanisms.

4.3 Formal Models and Verification

In addition to providing a convenient and efficient way of specifying and experimenting with various interaction semantics, this approach supports design-time verification for checking important properties of the system. With this approach, predefined and reusable formal models are created for every building block. Formal models of the selected building blocks are composed at verification time with formal models of components to form a system model that is then checked against the specified properties. Note that the designer is responsible for providing the models of the components and specifying the properties to be checked.

Through verification, users may find unexpected behaviors or errors in their system design. If the problems are caused by the interaction mechanisms, changes can be made by simply adjusting the building blocks of the connectors without having to modify the components. When this occurs, there is no need to recreate the component models. Moreover, predefined models for the building blocks can be used in most cases for the modified interaction mechanisms, also reducing the cost of model construction for verification.

In the example described in this paper, we used SPIN [25] as our back-end verifier to check properties of our design. Formal models of all the building blocks listed in Figure 9, as well as the system components, are described in Promela, the input language of SPIN. In particular, the formal models of the predefined ports and channels are made parameterizable and instantiatable so that they can be reused in different applications simply by plugging in different parameters. Specifically, we use the default message passing operations ("?" and "!") in Promela to implement the communications among components, ports and channels. Each port is a Promela `proctype` that takes two Promela native channels as parameters for communications with the component and the channel that are connected to this port. For the purpose of this paper, we have coded models in a way that reflects our goal of reusable and parameterizable building blocks. For a particular choice of interaction mechanisms, it might well be possible to implement connectors more directly using features of the Promela language. The full description of the Promela models for the building blocks is given in [43].

Notice that by using SPIN and Promela to support design-time verification, we are only showing one possible way to combine our design approach and verification. Our approach is not tied to particular formalisms or verification techniques. In fact, we have defined the same set of building blocks in the process algebra FSP and used LTSA [29] to verify the system designs. It is reasonable to expect, however, that when using different formalisms and verification techniques, specialized optimizations will need to be developed.

5. THE SINGLE-LANE BRIDGE PROBLEM REVISITED

In this section, we return to the single-lane bridge problem introduced in Section 2 to illustrate the use of the building blocks described above to facilitate iterative exploration and verification of designs. The architecture of the exactly-N-cars-per-turn system is shown in Figure 10. All the cars of a given color share a single connector to the controller of that color and a single connector to the controller of the other color. For the initial design, the developer chose an interaction mechanism with asynchronous blocking sends, blocking receives, and a reliable FIFO queue for each of these connectors. As indicated in the figure, this involves choosing the appropriate send and receive ports and channels.

Of course, we want our bridge system to satisfy the property that cars traveling in opposite directions are never allowed on the bridge at the same time. Constructing Promela models of the components and using the models of the building blocks from the library, as described in the previous section, we can use SPIN to determine whether the system satisfies the property. In this case, of course, SPIN produces a counterexample in which a blue car sends an `enter_request` message and enters the bridge, followed by a red car sending an `enter_request` message and entering the bridge. The problem is caused by the fact that the cars do not need to wait for the messages to be received by the controller before entering the bridge, as previously mentioned.

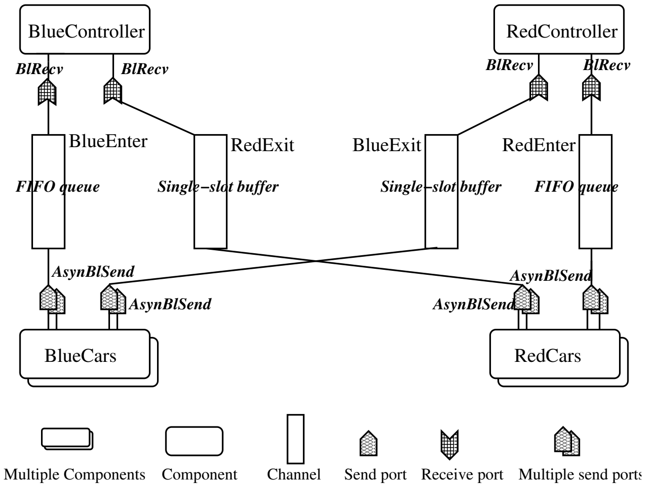


Figure 10: The architecture design of the "exactly-N-cars-per-turn" single-lane bridge problem

The designer might then try to solve the problem by changing the interaction mechanism to make the `enter_request` messages be sent synchronously, so that a car cannot enter the bridge before the controller has received the message. The only change required to achieve this is to replace the asynchronous blocking send ports with synchronous blocking send ports. Replacing the Promela code for these ports with the appropriate code from the library, we can redo the verification. In this case, SPIN reports that the property holds, and the system never allows cars traveling in opposite directions on the bridge at the same time.

Note that other designs can also satisfy the property. For instance, since the exact order in which the `exit_request` messages are received does not really matter and these messages are handled by a connector with blocking send ports (so that the sending component will wait until the channel can accept the message), we could replace the FIFO queues in the connectors carrying the `exit_request` messages with single-slot buffers. Again, we need only replace the Promela code for these channels and run SPIN again to see that this system would also keep cars from opposite directions from colliding on the bridge.

Of course, not all modifications to a system require only simple changes in the interaction mechanisms. Suppose that, in order to improve traffic flow, the designer wishes to modify the bridge system so that if cars on one side are waiting to cross and no cars from the other side are on the bridge or waiting to cross, the controllers will allow the waiting cars to cross no matter whose turn it might be. If cars from both sides are waiting, though, no more than N cars from one side can cross before cars from the other side get a turn.

To construct this "at-most-N-cars-if-waiting" system, we must add some communication between the controllers and

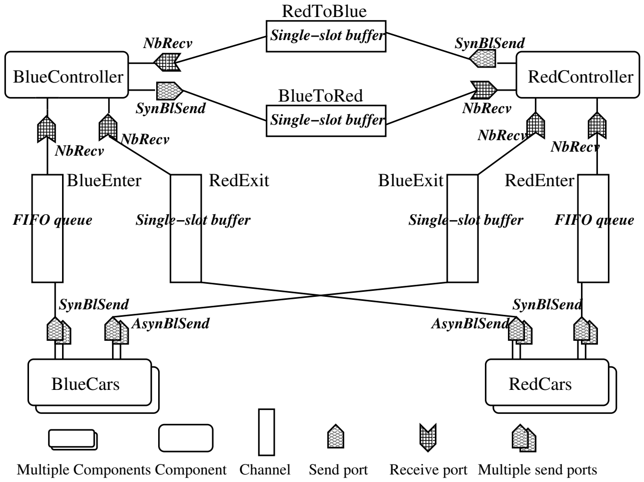


Figure 11: The architecture design of the "at-most-N-cars-if-waiting" single-lane bridge

modify the controller components to use it. Since this version of the system has additional functionality, it is not unreasonable to have to change the components to support this functionality. Still, however, we would like to limit the impact of these changes and reuse models of the components and connectors as much as possible.

Figure 11 shows a possible architecture for the modified system, with two new connectors between the components, one for the blue controller to notify the red controller that no blue cars are waiting and one for the red controller to notify the blue controller that no red cars are waiting. In this case the designer chose an interaction mechanism with synchronous blocking send, nonblocking receive, and a reliable single-slot buffer. Since the controllers will poll for messages from cars and from the other controller, we must also change the interaction mechanisms for the communications between cars and controller to have nonblocking receive semantics. To verify that this new system still prevents crashes of cars traveling in opposite directions on the bridge, the component models need to be modified to reflect the new communications. Models of the new connectors, however, can be constructed from the library models of building blocks.

Going further, the designer may then decide to modify the bridge system so that when (on-duty) emergency vehicles approach the bridge no new vehicles are allowed onto the bridge from the opposite side and the emergency vehicles are allowed to cross the bridge as soon as possible. Again there is new functionality so the components must be modified to provide this functionality. One way to design this version of the system is to modify the controller components so that when there are emergency vehicles waiting on the side of the controller that does not have the turn, that controller sends an `emergency_stop` signal to the other controller. The controller that holds the turn should stop accepting `enter_requests` messages from its side after receiving an `emergency_stop` signal if there are no emergency vehicles waiting on its side. This new design can use the existing connectors between the controllers, but the controller component logic will need to be modified to reflect the new functionality. To allow a controller to check if there is an emergency vehicle waiting on its side, a nonblocking copy receive port is used to poll the `enter_request` channel without actually removing a request from the channel. Furthermore, the FIFO queues used for the channels between the vehicles and the controllers must be changed to priority queues so that the emergency vehicles are allowed to cross the bridge ahead of the cars. Due to space limitations, we leave out the details of the implementation of these priority queues. For this version of the system, the designer wants to verify the property that emergency vehicles are allowed to cross the bridge as soon as possible as well as the property that no vehicles traveling in opposite directions are on the bridge at the same time. After creating new models of the controller components, models of the new connectors can be constructed from the library models of building blocks and both properties verified using Spin.

This example has illustrated a series of system designs that attempt to fix problems as well as add new functionality. Using our plug-and-play approach, the impact of each change was kept relatively local. Components had to be modified when only new functionality was added. When connectors were changed, they could be easily composed from the building blocks. For each version, components and connector models were reused, making verification much easier. Note, all the library building blocks and versions of the components are described in [43].

6. OTHER INTERACTION MECHANISMS

Although our experience with other interaction mechanisms is not as extensive as with message passing, we have begun to explore the application of our approach to other interaction mechanisms such as publish/subscribe. In fact, based on case studies about available publish/subscribe systems and event models such as CORBA Event Services [1], Java Event Models, JEDI [11], SIENA [8], etc., we have found that the set of building blocks we have defined for message passing can be easily extended to describe a range of important semantics for publish/subscribe.

In publish/subscribe systems, the fundamental communications between components and connectors are the announcement of events by components, the delivery of events to components, and the subscription or unsubscription by which components indicate their interest in particular events. It is straightforward to map these communications to sending and receiving messages; therefore they can be described using available message passing building blocks. In message passing, it is almost always the case that the sender initiates the communication by pushing messages to the connector and the receiver pulls messages from the connector. Unlike message passing, however, most publish/subscribe systems support one or more combinations of *push/pull* on both the publisher side and the subscriber side. To describe these semantics, we define new kinds of send and receive ports for publish/subscribe (Figure 12).

With the *pull send ports*, communication is initiated by the port forwarding a *pull request* from the channel to the sender. A message (or event, but we use them interchangeably here) is then subsequently handed to the port by the sender. The port blocks either until the message is received by the channel (*asynchronous blocking*) or until it is received by a receiver (*synchronous blocking*)[2]. With the *push receive*

[2] Although this type of strict synchronization is only used from time to time in publish/subscribe systems, it can be

		Message Passing	Publish/Subscribe
Send Port (push)	Asynchronous Nonblocking	✓	✓
	Asynchronous Blocking	✓	✓
	Asynchronous Checking	✓	✓
	Synchronous Blocking	✓	✓
	Synchronous Checking	✓	✓
Send Port (pull)	Synchronous Blocking		✓
	Asynchronous Blocking		✓
Receive Port (pull)	Blocking	✓	✓
	Nonblocking	✓	✓
Receive Port (push)	Synchronous Blocking		✓
	Synchronous Nonblocking		✓
	Asynchronous Blocking		✓
	Asynchronous Nonblocking		✓

Figure 12: A shared library of send and receive ports

port, communication is initiated by the channel pushing a message to the port which then forwards the message to the receiver. The receiver may block until a a message is available in the port (*blocking*) or return and check back later (*nonblocking*). When a message is delivered to a component, a confirmation may be issued by the port and sent to the channel (*synchronous*).

The newly introduced ports are designed such that components interacting through publish/subscribe can employ the same interface as when they interact through message passing. So again, it is possible for designers to change their choices of interaction mechanisms without changing the design of components. Notice that here we are only describing the synchronization semantics between components and connectors and have not described the semantics of the communication media inside a connector. In publish/subscribe systems, the communication media are often much more complicated than in message passing. Publish/subscribe connectors might involve a router, a registrar, and an assortment of message filtering capabilities. It is certainly desirable and possible to provide parameterizable building blocks for the different aspects of the communication media. Garlan et.al. have defined a set of building blocks based on the dimensions of the publish/subscribe middleware [18]. Other work on defining semantics and comparing variations for publish/subscribe systems include [5, 14–16]. We are working to define additional building blocks to express these other aspects of the semantics of publish/subscribe systems. In addition, using combinations of different send and receive ports we have defined for message passing and publish/subscribe, we have been able to describe several versions of remote procedure calls (RPC) such as the one-way RPC, synchronous RPC, deferred synchronous RPC, and asynchronous RPC [40].

7. RELATED WORK

The limitations and frustrations of component-based development are well known (e.g., [17, 26]). Previous work such as [2, 4, 20, 28, 33, 36] has proposed treating connectors as first-class entities in component-based development, although [20] in particular, has put the focus at a lower level of abstraction (programming level) than what we are interested in.

The idea of specifying complex connectors and modeling them for verification is, of course, not new. The Wright architecture description language [2], for example, used the CSP process algebra to describe arbitrary connectors, and the Architectural Interaction Diagrams (AIDs) of Ray and Cleaveland [34] use process algebra methods to construct connectors hierarchically. Constraint automata based approaches have also been proposed to specify and analyze the semantics of connectors composed from a set of primitive channels [3, 32]. In approaches like these, the burden is on the designer to construct a model of a connector with the right semantics from powerful, but low-level, primitives. Our approach is aimed more at providing a library of building blocks from which connectors representing widely used interaction mechanisms can be easily constructed, offering "ready-to-use" pieces that hide from the user most of the details of how these pieces are actually constructed and modeled. As we noted above, however, the actual formal models of our building blocks used for verification could be built using any suitable formalisms with verification support, including CSP or AIDs.

Our use of ports to allow a standard component interface and facilitate the substitution of connectors with different semantics is closely related to the connector wrappers of [38], although that work is aimed more at adapting existing connectors and our emphasis is on building up new connectors that can be easily exchanged for one another. Our notion of ports is very similar to what is described in [35]. The difference is that in our approach, ports are part of connectors and provide more complex semantics.

The term *building blocks* has been often used in different contexts. For example, in [42], building blocks are referred to as parts of software used to build a system. The building blocks in our approach are design-level elements used to construct connectors representing interactions.

Our approach differs from previous work on architectural evolution (e.g., [30, 41]) in our focus on supporting the exploration of different interaction mechanisms at the design stage and our emphasis on modeling and verification. Our goals are to support the design of systems that may involve heterogeneous interaction mechanisms between different components and to allow the designer to easily experiment with different mechanisms validating the suitability of different combinations with finite-state verification tools.

Our work on the semantics of interaction mechanisms is closely related to work on categorizing connectors (e.g. [23, 31]). In particular, our analysis of the dimensions for message passing semantics is similar in spirit to the analysis of publish/subscribe systems in [18]. In terms of applying verification to one particular interaction mechanism, as we did with message passing, there has been extensive work on modeling and verifying publish/subscribe systems(e.g. [6, 21, 44]) However, this work has not attempted to introduce explicit design-level building blocks to allow the construction of connectors with different semantics as we did. And our approach is intended to support many kinds of mechanisms, rather than being restricted to a single type.

A number of middleware frameworks support component-

easily described using the available ports.

based development, although each typically allows a somewhat limited range of interaction mechanisms and no direct support is provided for verification. Some work, such as the Cadena system [9], has been directed at providing verification support for systems built on standard middleware. A number of approaches have also been proposed for assembling existing components into applications, including mediators [39], active interfaces [22], and various techniques for wrapping components. Our interest here is more in the choice of interaction mechanisms between components and less on the adaptation of existing components to interact with each other.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose a compositional specification approach that helps designers more easily experiment with different interaction mechanisms between components. By decomposing the connectors into ports and channels, and using ports as mediators between components and channels, we are able to keep the interface of the components simple and standardized so that changes to the interaction mechanisms can be made with little or no modification to the components. The decomposition also allows us to build a library of ports and channels as reusable building blocks to construct connectors with different semantics. Our approach is also integrated with finite-state verification techniques, facilitating design-time verification and the early detection of design errors. Using our approach, designers may experiment with their choice of design for various interaction semantics by simply dragging, plugging in, or replacing building blocks and frequently using verification to check their design choices. While this process may repeat, our approach allows considerable reuse of the models of components and connectors. Consequently, we also save on model-construction time while doing the verification.

Our long-term goal is to provide a framework that integrates compositional specification and design-time verification and supports a rich library of building blocks from which one can build different interaction mechanisms. An architecture description language and a GUI design environment may be developed to allow designers to easily use building blocks to specify and verify a system architecture. We intend to explore the semantics of other commonly used interaction mechanisms and to construct additional building blocks to express those semantics. It may also be useful to allow users to define their own building blocks, requiring a more systematic way of defining and modeling the building blocks. There are a number of interesting issues related to design-time verification. For instance, a variety of optimizations could be developed to reduce the formal system models that are composed of the building blocks and models of the components; these depend, of course, on the particular modeling formalism and verification tools being applied. We need to explore these optimizations and learn when they can be profitably applied. Finally, more extensive case studies need to be done to evaluate the effectiveness of our approach.

9. ACKNOWLEDGEMENTS

We are grateful to Prashant Shenoy for helpful conversations about this work.

This material is based upon work supported by the National Science Foundation under awards CCF-0427071 and CCR-0205575 and by the U.S. Department of Defense/Army Research Office under award DAA-D19-01-1-0564 and award DAAD19-03-1-0133. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U. S. Department of Defense/Army Research Office.

10. REFERENCES

[1] OMG, Notification service specification c1.0.1, 2002.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Softw. Eng. and Methodol.*, pages 140–165, 1997.

[3] F. Arbab, C. Baier, J. J. M. M. Rutten, and M. Sirjani. Modeling component connectors in reo by constraint automata: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 97:25–46, 2004.

[4] D. Bálek and F. Plášil. Software connectors and their role in component deployment. In *Proc. Third Intl. Working Conf. on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, 2001.

[5] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise. A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, 5(4), 1996.

[6] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. 11th ACM Symp. on Found. of Softw. Eng.*, Finland, Sept. 2003.

[7] Carriero, N., and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–58, Apr 1989.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, Aug. 2001.

[9] A. Childs, J. Greenwald, V. P. Ranganath, X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, P. Shanti, and G. Singh. Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems. In *Proc. of Fund. Approaches to Softw. Eng., 7th Intl. Conf.*, pages 160–164, Mar. 2004.

[10] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, Jan. 1995.

[11] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. 20th Intl. Conf. on Softw. Eng.*, pages 261–270, Kyoto, Japan, 1998.

[12] M. Day. Occam. *SIGPLAN Notices*, 18(4):69–79, Apr 1983.

[13] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. on Softw. Eng. and Methodol.*, 13(4):359–430, 2004.

[14] W. Emmerich and N. Kaveh. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *Proc. 24th Intl. Conf. on Softw. Eng,*, pages 691–692, 2002.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[16] R. S. S. Filho, C. R. B. de Souza, and D. F. Redmiles. The design of a configurable, extensible and dynamic notification service. In *Proc. 2nd Intl. Workshop on Distributed Event-based Systems*, pages 1–8, San Diego, California, 2003.

[17] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proc. 17th Intl. Conf. on Softw. Eng.*, pages 179–185, Seattle, Washington, Apr. 1995.

[18] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proc. 10th Intl. SPIN Workshop on Model Checking of Softw.*, Portland, Oregon, 2003.

[19] Geist, A., A. Beguelin, J. Dongarra, W. Wiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[20] T. Gensler and W. Lowe. Correct composition of distributed systems. In *Tech. of Object-Oriented Languages and Systems*, 1999.

[21] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. 9th European Softw. Eng. Conf. / 11th ACM SIGSOFT Intl. Symp. on Found. of Softw. Eng.*, pages 257–266, Helsinki, Finland, 2003.

[22] G. Heineman. Adaption of software components. In *2nd Intl. Workshop on Component-Based Softw. Eng. / the 21st Intl. Conf. on Softw. Eng.*, Los Angeles, CA, June 1999.

[23] D. Hirsch, S. Uchitel, and D. Yankelevich. Towards a periodic table of connectors. In *Proc. Third Intl. Conf. on Coordination Languages and Models*, page 418, London, UK, 1999.

[24] Hoare and C.A.R. *Communicating Sequential Processes*. Englewood Cliffs, NJ:Prentice-Hall Intl., 1985.

[25] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, Boston, 2004.

[26] P. Inverardi and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.

[27] K.L.McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem.* Kluwer Academic, 1993.

[28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. 5th European Softw. Eng. Conf.*, pages 137–153, Sitges, Spain, Sept. 1995.

[29] J. Magee and J. Kramer. *Concurrency State Models and Java Programs.* John Wiley and Sons, 1999.

[30] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proc. 21st Intl. Conf. on Soft. Eng.*, pages 44–53, Los Angeles, May 1999.

[31] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd Intl. Conf. on Softw. Eng.*, pages 178–187, Limerick, Ireland, 2000.

[32] N. R. Mehta, N. Medvidovic, M. Sirjani, and F. Arbab. Modeling behavior in compositions of software architectural primitives. In *19th IEEE Intl. Conf. on Automated Softw. Eng.*, pages 371–374, 2004.

[33] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[34] A. Ray and R. Cleaveland. Architectural interaction diagrams: AIDs for system modeling. In *Proc. 25th Intl. Conf. on Softw. Eng.*, pages 396–406, 2003.

[35] B. Selic. Using UML for modeling complex real-time systems. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, Montreal, Canada, June 1998.

[36] M. Shaw and D. Garlan. *Softw. Architecture:Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[37] Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[38] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proc. 2003 Intl. Conf. on Softw. Eng.*, Portland, Oregon, 2003.

[39] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, 1992.

[40] A. S. Tanenbaum and M. van Steen. *Distributed Systems. Principles and Paradigms.* Prentice Hall, 2002.

[41] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In P. Inverardi, editor, *Proc. 8th European Softw. Eng. Conf./9th Symp. on the Found. of Softw. Eng.*, pages 1–10, Vienna, Sept. 2001.

[42] F. J. van der Linden and J. K. Mller. Creating architectures with building blocks. *IEEE Softw.*, 12(6):51–60, 1995.

[43] S. Wang, G. S. Avrunin, and L. A. Clarke. Architectural building blocks for plug-and-play system design. Technical Report UM-CS-2005-16, Dept. of Comp. Sci., Univ. of Massachusetts, 2005.

[44] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *Proc. Specification and Verification of Component-Based Systems*, pages 35–41, Helsinki, Finland, 2003.

APPENDIX

A. EXAMPLE BUILDING BLOCKS IN PROMELA

```
#define BUFFER_SIZE 3
/* internal communication signals */
mtype = {SEND_SUCC, SEND_FAIL, IN_OK, IN_FAIL};
mtype = {OUT_OK, OUT_FAIL, RECV_OK, RECV_SUCC, RECV_FAIL};
/* messages from components */
typedef DataMsg{
byte data;              // application-specific data
        byte sender_id;         // filled by the send port
        byte selectiveData;     // matching criteria
        bool selective;         // selective/nonselective receive
        bool remove             // remove/copy receive
 };
/* internal messages from ports and channels */
typedef InternalMsg{
mtype signal;
        byte port_pid  // used to route different internal signals
                       // using -1 if the id does not matter
};
/* two synchronous channels for internal communication */
typedef SynChan{
  chan c = [0] of {InternalMsg};
  chan d = [0] of {DataMsg}
}
mtype sendStatus;   // SEND_SUCC, SEND_FAIL
mtype bufferStatus; // IN_OK, IN_FAIL
mtype recvStatus;   // RECV_SUCC, RECV_FAIL
/****************** send ports ***************************/
proctype AsynNbSendPort(SynChan component; SynChan channel){
   DataMsg m;
   end: do
        :: channel.c?_,eval(_pid); // ignore irrelevant signals
        :: atomic{
              component.d?m;
              component.c!SEND_SUCC,-1;
              m.sender_id = _pid;
              channel.d!m
           }
        od
}
proctype AsynBlSendPort(SynChan component; SynChan channel){
      DataMsg m;
      end: do
           :: channel.c?_,eval(_pid);
           :: atomic{
                 component.d?m;
                 m.sender_id = _pid;
                 do
                 :: channel.d!m;
                    if
                    :: channel.c?IN_OK,_;
                       break
                    :: channel.c?IN_FAIL,_
                    fi
                 od;
                 component.c!SEND_SUCC,-1
              }
           od
}
proctype SynBlSendPort(SynChan component; SynChan channel){
      DataMsg m;
      end: do
           :: atomic{
             component.d?m;
              m.sender_id = _pid;
              do
              :: channel.d!m;
                 if
                 :: channel.c?IN_OK,_;
                    break
                 :: channel.c?IN_FAIL,_
                 fi
              od;
              channel.c?RECV_OK,eval(_pid);
              component.c!SEND_SUCC,-1
              }
           od
}
proctype AsynCheckingSendPort(SynChan component; SynChan channel){
      DataMsg m;

      end: do
           :: channel.c?_,eval(_pid);
           :: atomic{
                 component.d?m;
                 m.sender_id = _pid;
                 channel.d!m;
                 if
                 :: channel.c?IN_OK,_;
                    component.c!SEND_SUCC,-1
                 :: channel.c?IN_FAIL,_;   // won't try again
                    component.c!SEND_FAIL,-1
                 fi
              }
           od
}
/****************** receive port **************************/
proctype BlRecvPort(SynChan component; SynChan channel){
      DataMsg recvRequest,m;
      end: do
           :: atomic{
                 component.d?recvRequest;
                 do
                 :: channel.d!recvRequest;
                    if
                    :: channel.c?OUT_OK,_;
                       channel.d?m;
                       break
                    :: channel.c?OUT_FAIL,_  // if fail, try again
                    fi
                 od;
                 component.c!RECV_SUCC,-1;
                 component.d!m    //should always be a valid message
              }
           od
}
proctype NbRecvPort(SynChan component; SynChan channel){
      DataMsg recvRequest,m;
      end: do
           :: atomic{
                 component.d?recvRequest;
                 channel.d!recvRequest;
                 if
                 :: channel.c?OUT_OK,_;
                    channel.d?m;
                    component.c!RECV_SUCC,-1
                 :: channel.c?OUT_FAIL,_;
                    component.c!RECV_FAIL,-1
                 fi;
                 component.d!m  // may or may not be a valid message
              }
           od
}
/************************* channel ********************************/
/* channels have only one incoming and one outgoing synchans (one-way).
 * On each synchan, there might be multiple send or receive ports listerning.
 */
proctype single_slot_buffer (SynChan sender; SynChan receiver){
    DataMsg recvRequest, m, buffer;
    bool buffer_empty = 1;
    do
    :: receiver.d?recvRequest;     /* handle receive request */
       if
       :: (!buffer_empty && !recvRequest.selective)
           || (!buffer_empty && recvRequest.selective
               && buffer.selectiveData == recvRequest.selectiveData) ->
             receiver.c!OUT_OK,-1;
             receiver.d!buffer;
             sender.c!RECV_OK,buffer.sender_id;
             if
             :: recvRequest.remove ->
                  buffer_empty = 1
             :: else
             fi
       :: else ->
             receiver.c!OUT_FAIL,-1
       fi
   :: sender.d?m;  /* handle send request */
       if
       :: buffer_empty ->
            sender.c!IN_OK,-1;
            buffer.data = m.data;
            buffer.sender_id = m.sender_id;
            buffer.selectiveData = m.selectiveData;
            buffer.selective = m.selective;
            buffer.remove = m.remove;
            buffer_empty = 0
       :: else ->
            sender.c!IN_FAIL,-1
       fi
    od
}
proctype FIFO_queue(SynChan sender; SynChan receiver){
    chan buffer = [BUFFER_SIZE] of { DataMsg };
    DataMsg m, recvRequest;
    do
    :: receiver.d?recvRequest;     /* handle receive */
       if
       :: !recvRequest.selective ->
          /* find the first message in the buffer */
          if
          :: buffer?[m];
             if
             :: recvRequest.remove ->
                  buffer?m
             :: else ->
                  buffer?<m>
             fi;
             receiver.c!OUT_OK,-1;
             receiver.d!m;
             sender.c!RECV_OK,m.sender_id
          :: else ->
             receiver.c!OUT_FAIL,-1
          fi
       :: recvRequest.selective ->
          /* find the first matching message in the buffer */
          if
          :: buffer??[m.data,m.sender_id,
                      eval(recvRequest.selectiveData),
                      m.selective, m.remove];
             if
             :: recvRequest.remove ->
                buffer??m.data,m.sender_id,
                        eval(recvRequest.selectiveData),
                        m.selective, m.remove
             :: else ->
                buffer??<m.data,m.sender_id,
                         eval(recvRequest.selectiveData),
                         m.selective, m.remove>
               fi;
               receiver.c!OUT_OK,-1;
               receiver.d!m;
               sender.c!RECV_OK,m.sender_id
            :: else ->
               receiver.c!OUT_FAIL,-1
           fi
       fi
    :: sender.d?m;
       if
       :: full(buffer) ->
          sender.c!IN_FAIL, -1
       :: nfull(buffer) ->
          buffer!m;
          sender.c!IN_OK,-1
       fi
    od
}
/* A priority queue that only handles two priorities  */
proctype priority_queue(SynChan sender; SynChan receiver){
    chan buffer = [BUFFER_SIZE] of { DataMsg };
    DataMsg m, recvRequest;
    do
    :: receiver.d?recvRequest;     /* handle receive */
       if
       :: !recvRequest.selective ->
          if
          /* find the message with higher priority */
          :: buffer??[m.data,m.sender_id, 1,
                      m.selective, m.remove];
             if
             :: recvRequest.remove ->
                buffer??m.data,m.sender_id, 1,
                        m.selective, m.remove;
             :: else ->
                buffer??<m.data,m.sender_id, 1,
                         m.selective, m.remove>;
             fi;
             receiver.c!OUT_OK,-1;
             receiver.d!m;
             sender.c!RECV_OK,m.sender_id
          :: else -> //retrieve a message with lower priority
             if
             :: buffer??[m.data,m.sender_id, 0,
                         m.selective, m.remove];
                if
                :: recvRequest.remove ->
                   buffer??m.data,m.sender_id, 0,
                           m.selective, m.remove;
                :: else ->
                   buffer??<m.data,m.sender_id, 0,
                            m.selective, m.remove>;
                fi;
                receiver.c!OUT_OK,-1;
                receiver.d!m;
                sender.c!RECV_OK,m.sender_id
             :: else ->
                receiver.c!OUT_FAIL,-1
             fi
          fi
       :: recvRequest.selective ->
          /* find the first matching message in the buffer */
          if
          :: buffer??[m.data,m.sender_id,
                      eval(recvRequest.selectiveData),
                      m.selective, m.remove];
             if
             :: recvRequest.remove ->
                buffer??m.data,m.sender_id,
                        eval(recvRequest.selectiveData),
                        m.selective, m.remove
             :: else ->
                buffer??<m.data,m.sender_id,
                         eval(recvRequest.selectiveData),
                         m.selective, m.remove>
             fi;
             receiver.c!OUT_OK,-1;
             receiver.d!m;
             sender.c!RECV_OK,m.sender_id
          :: else ->
             receiver.c!OUT_FAIL,-1
          fi
       fi
    :: sender.d?m;
       if
       :: full(buffer) ->
          sender.c!IN_FAIL, -1
       :: nfull(buffer) ->
          buffer!m;
          sender.c!IN_OK,-1
       fi
    od
}
```

B. THE BRIDGE EXAMPLE VERSION 1

```
/**************** "exactly-N-cars-per-turn" ***************/
#define EACH_TURN_MAX 2   //maximum number of cars allowed each turn
#define NUM_OF_CARS 2     //total number of cars from each direction
#define INITIAL_TURN 1    //the blue controller

proctype car(SynChan enter; SynChan exit){
/* contents of these messages do not matter in this case */
DataMsg enter_request;
DataMsg exit_request;

end:do
    ::  enter.d!enter_request; // connected to SynBlSend port
        enter.c?sendStatus,_;
        onbridge: skip;
        offbridge: skip;
        exit.d!exit_request; // connected to AsynBnSend port
        exit.c?sendStatus,_
    od
}
proctype controller(bool myColor; SynChan enter; SynChan exit){
   byte count = 0;
   bool myturn = false;
   DataMsg recvRequest, enter_request, exit_request;

   if
   :: (myColor == INITIAL_TURN) -> myturn = true
   :: else
   fi;
   recvRequest.selective = 0; //nonselective receive
   recvRequest.remove = 1;    //remove receive

 end:do
     :: myturn -> /* handle enter_request */
          atomic{
            enter.d!recvRequest; // connected to BlRecv port
            enter.c?recvStatus,_;
            enter.d?enter_request;

            count++;
            if
            :: (count == EACH_TURN_MAX) ->
                    myturn = false;
                    count = 0
            :: else
            fi
          }
     :: !myturn ->
           atomic{
             exit.d!recvRequest; // connected to BlRecv port
             exit.c?recvStatus,_;
             exit.d?exit_request;

             count++;
             if
             :: (count == EACH_TURN_MAX) ->
                     myturn = true;
                     count = 0
             :: else
             fi
           }
     od
}
/* initialize the system
 * compose different channels, ports and components
 * using default ‘‘promela channels’’ */
init{
        byte i=0;
  /** define ‘‘promela channels’’ for internal communication **/
        /* for communitation between components and ports */
        SynChan BlueCar_BlueEnter[NUM_OF_CARS];
        SynChan BlueCar_BlueExit[NUM_OF_CARS];
        SynChan RedCar_RedEnter[NUM_OF_CARS];
        SynChan RedCar_RedExit[NUM_OF_CARS];
        SynChan BlueController_BlueEnter;
        SynChan BlueController_RedExit;
        SynChan RedController_RedEnter;
        SynChan RedController_BlueExit;

        /* for communication between channels and ports */
        SynChan BlueEnter_BlueController;
        SynChan BlueExit_RedController;
        SynChan BlueEnter_BlueCar;
        SynChan BlueExit_BlueCar;
        SynChan RedEnter_RedController;
        SynChan RedExit_BlueController;
        SynChan RedEnter_RedCar;
        SynChan RedExit_RedCar;
  atomic{
   do
   /* start car components and corresponding send/receive ports */
   :: (i<NUM_OF_CARS)->
      run car(BlueCar_BlueEnter[i],BlueCar_BlueExit[i]);
      run car(RedCar_RedEnter[i],RedCar_RedExit[i]);
      run SynBlSendPort(BlueCar_BlueEnter[i],BlueEnter_BlueCar);
      run SynBlSendPort(RedCar_RedEnter[i],RedEnter_RedCar);
      run AsynBlSendPort(BlueCar_BlueExit[i],BlueExit_BlueCar);
      run AsynBlSendPort(RedCar_RedExit[i],RedExit_RedCar);
      i++
   :: else -> break
   od;
   /* start channels and corresponding send/receive ports */
   run FIFO_queue(BlueEnter_BlueCar,BlueEnter_BlueController);
   run FIFO_queue(RedEnter_RedCar,RedEnter_RedController);
   run single_slot_buffer(BlueExit_BlueCar,BlueExit_RedController);
   run controller(1,BlueController_BlueEnter,BlueController_RedExit);
   run controller(0,RedController_RedEnter,RedController_BlueExit);
   run NbRecvPort(BlueController_BlueEnter, BlueEnter_BlueController);
   run NbRecvPort(RedController_RedEnter, RedEnter_RedController);
   run NbRecvPort(BlueController_RedExit, RedExit_BlueController);
   run NbRecvPort(RedController_BlueExit, BlueExit_RedController);
  }
}
/* Property: cars from two directions should never */
 *           be on the bridge at the same time     */
/* car[1] blue, car[2] red       */

never {    /*  ! ( [] ( ! ( ( p ) && ( q ) ) ) )  */
T0_init:
        if
        :: ((car[1]@onbridge) && (car[2]@onbridge)) ->
           goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

C. THE BRIDGE EXAMPLE VERSION 2

```
/* "at-most-N-cars-per-turn-if-waiting" */
proctype controller(bool myColor; SynChan enter; SynChan exit;
                    SynChan toOther; SynChan fromOther){
   byte count = 0;
   bool myturn = 0;
   bool otherFinished = 0;
   byte numEntered = 0;
   DataMsg recvRequest, enter_request,
           exit_request, finish_notification_with_num;
   recvRequest.remove = 1;
   recvRequest.selective = 0;
   if
   :: (myColor == INITIAL_TURN) -> myturn = true
   :: else
   fi;
 end:do
     :: myturn ->
        enter.d!recvRequest;   //connected to NbRecv port
        enter.c?recvStatus,_;
        enter.d?enter_request;
        if
        :: (recvStatus == RECV_SUCC) ->
           count++;
           if
           :: (count == EACH_TURN_MAX) -> //finished, yields turn
               myturn = false;
               finish_notification_with_num.data = count;
               toOther.d!finish_notification_with_num;
               toOther.c?sendStatus,_; //connected to SynBlSend port
               count = 0
           :: else
           fi
        :: (recvStatus == RECV_FAIL) -> //finished, yields turn
            myturn = false;
            finish_notification_with_num.data = count;
            toOther.d!finish_notification_with_num;
            toOther.c?sendStatus,_; //connected to SynBlSend port
            count = 0
        fi
     :: !myturn ->
            if //check if already received ‘‘finish’’ signal
            :: (otherFinished) -> goto L1;
            :: else
            fi;
            /* check if a finish notification is available */
            fromOther.d!recvRequest;    // connected to NbRecv port
            fromOther.c?recvStatus,_;
            fromOther.d?finish_notification_with_num;
            if
            :: (recvStatus == RECV_SUCC) ->
               numEntered = finish_notification_with_num.data;
               otherFinished = true
            :: (recvStatus == RECV_FAIL) ->
            fi;
            /* handle exit requests */
       L1:  exit.d!recvRequest;
```

```
            exit.c?recvStatus,_;
            exit.d?exit_request;
            if
            :: (recvStatus == RECV_SUCC) ->
                 count++
            :: (recvStatus == RECV_FAIL) ->
            fi;
            if
            :: (otherFinished && count == numEntered) ->
               myturn = true;
               otherFinished = false;
               count = 0
            :: else
            fi;
    od
}
```

D. THE BRIDGE EXAMPLE VERSION 3

```
/* "at-most-N-cars-per-turn-if-waiting" with emergency vehicles */
#define NUM_OF_NM_CARS 1     //number of normal cars from each direction
#define NUM_OF_EM_CARS 1     //number of emergency cars from each direction
#define NUM_OF_CARS_TOTAL 2  //total number of cars from each direction

proctype car(SynChan enter; SynChan exit; byte emergency){
DataMsg enter_request;
DataMsg exit_request;

enter_request.selectiveData = emergency;
end:do
    ::  enter.d!enter_request;
enter.c?sendStatus,_;
        onbridge: skip;
        offbridge: skip;
        exit.d!exit_request;
        exit.c?sendStatus,_
    od
}
proctype controller(bool myColor; SynChan enter; SynChan exit;
                    SynChan toOther; SynChan fromOther){
   byte count = 0;
   bool myturn = 0;
   bool otherFinished = 0;
   byte numEntered = 0;
   DataMsg recvRequest, enter_request, exit_request,
           finish_notification_with_num, emergency_stop;

   if
   :: (myColor == INITIAL_TURN) -> myturn = true
   :: else
   fi;
   recvRequest.selective = 0;
 end: do
     :: myturn ->
        recvRequest.remove = 0; // nonblocking copy receive
        enter.d!recvRequest;
        enter.c?recvStatus,_;
        enter.d?enter_request;
        if
        :: (recvStatus == RECV_SUCC) ->
           if
           /* check if an emergency_stop signal is available only when */
           /* there is no emergency car waiting on my own side */
           :: enter_request.selectiveData == 0  ->
              /* there is only request from normal vehicles */
              recvRequest.remove = 1;
              fromOther.d!recvRequest;
              fromOther.c?recvStatus,_;
              fromOther.d?emergency_stop;
              if
              :: (recvStatus == RECV_SUCC)
                 || (count == EACH_TURN_MAX) ->
                 /* there are emergency cars waiting on the other side */
                 /* finish */
                 myturn = false;
                 finish_notification_with_num.data = count;
                 toOther.d!finish_notification_with_num;
                 toOther.c?sendStatus,_;
                 count = 0;
                 goto end
              :: else
              fi
           :: else
           fi;
           /* continue to handle my own enter_request */
           recvRequest.remove = 1;    //nonblocking remove receive
           enter.d!recvRequest;
           enter.c?recvStatus,_;
           enter.d?enter_request;
           count++
        :: (recvStatus == RECV_FAIL) -> // finish
           myturn = false;
           finish_notification_with_num.data = count;
           toOther.d!finish_notification_with_num;
           toOther.c?sendStatus,_;
           count = 0;
           /* flush the possible emergency_stop signal*/
           recvRequest.remove = 1;
           fromOther.d!recvRequest;
           fromOther.c?recvStatus,_;
           fromOther.d?emergency_stop
        fi
     :: !myturn ->
        if
        :: (otherFinished) -> goto L1;
        :: else
        fi;
        /* check if there are emergency vehicles waiting on my side */
        recvRequest.remove = 0;  //don’t remove
        enter.d!recvRequest;
        enter.c?recvStatus,_;
        enter.d?enter_request;
        if
        :: (recvStatus == RECV_SUCC
           && enter_request.selectiveData == 1) ->
           /* if there are emergency cars waiting */
            toOther.d!emergency_stop;
            toOther.c?sendStatus,_;
        :: (recvStatus == RECV_FAIL) ->
        fi;
        /* check if a finish notification is available */
        recvRequest.remove = 1;
        fromOther.d!recvRequest;     //nonblocking remove receive
        fromOther.c?recvStatus,_;
        fromOther.d?finish_notification_with_num;
        if
        :: (recvStatus == RECV_SUCC) ->
            numEntered = finish_notification_with_num.data;
            otherFinished = true
        :: (recvStatus == RECV_FAIL) ->
        fi;
        /* handle exit requests */
   L1:  recvRequest.remove = 1;
        exit.d!recvRequest;
        exit.c?recvStatus,_;
        exit.d?exit_request;
        if
        :: (recvStatus == RECV_SUCC) ->
           count++
        :: (recvStatus == RECV_FAIL) ->
        fi;
        if
        :: (otherFinished && count == numEntered) ->
           myturn = true;
           otherFinished = false;
           count = 0
        :: else
        fi
    od
}
```