Verification Support for Plug-and-Play Architectural 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

In software architecture, components are intended to represent the computational units of the system and connectors are intended to represent the interactions between those units. Choosing the semantics of these interactions is a key part of the design process, but the wide range of alternatives from which to choose and the complexity of the behavior affected by the choices makes it difficult to get them right.

We propose an approach in which connectors with particular semantics are constructed from a library of pre-defined building blocks and changes in the semantics of a connector can be accomplished by replacing some of its building blocks with others. In our approach, a small set of standard interfaces allows components to communicate with each other through a wide variety of connectors, so the impact on components for even substantial changes in the semantics of the connectors is minimized.

In this paper, we focus on the way this approach supports design-time verification to provide feedback about the correctness of the design. By enhancing the re-use of models of both components and connectors, this approach has the potential to significantly reduce the cost of verification as a design evolves.

1. INTRODUCTION

In software architecture, connectors are intended to represent the specific semantics of how components interact with each other. They capture some of the most important yet subtle aspects of a system, such as non-determinism, interleavings of computations, synchronization, and inter-component communication. These are all concerns that can be particularly difficult to fully comprehend in terms of their impact on the overall system behavior.

The large design space of available interaction mechanisms and their variations only adds to this difficulty. Choosing appropriate interaction semantics for a connector often involves not only a choice from commonly used interaction mechanisms, such as remote procedure call, message passing, and publish/subscribe, but also decisions about such details as the particular type and size of a message buffer or whether a communication should be synchronous or asynchronous. Thus, during architectural design, it is important that designers be able to select the specific interaction semantics they think should be employed and then to get feedback about the appropriateness of those choices based on their impact on the overall system behavior.

In particular, one would like to be able to propose a design, and use design-time verification to determine whether some important properties of the system are satisfied. Violations of these properties often reflect system behavior that was not anticipated by the designer due to the complexity of the interactions between components. When such a violation is found during design-time verification, changes have to be made to correct the specific interaction semantics that are causing the violation, and verification needs to be re-applied to confirm that the changes fix the problem. Given the wide variety of interaction mechanisms and the complexity of their semantics, it is expected that a system designer would have to go through a number of iterations modifying their decisions on interactions and re-verifying the overall system before a satisfactory design is achieved.

One major obstacle to the realization of this vision of design and verification is that the semantics of the interactions are often intertwined with the semantics of the components' computations. For example, a change from an asynchronous communication to a synchronous one often requires making changes to the components so that a callback can be placed to explicitly notify the sender of the receipt of messages. In general, experimenting with alternative choices of interaction semantics tends to be very difficult and inefficient when changes made in the interactions often require non-trivial changes in the components' computations.

This problem also complicates design-time verification. When using finite-state verification techniques, for instance, it is necessary to build a model of the system that represents the computation of each component and the interactions between them. With semantics of interactions intertwined with semantics of computations, any changes made to the interactions will often result in not only the re-construction of the connector models but also the component models. When repeated changes and verification of a design are necessary, the lack of reusability of the component and connector models could increase the cost of the design-time verification significantly.

Our approach tries to address these difficulties by providing system designers with an efficient way of experimenting with alternative design choices for connectors, and the ability to evaluate these design decisions based on the correctness of the overall system design using finite-state verification. In our approach, with just a small set of standard interfaces, components can communicate with each other through different connectors that express a wide range of interaction semantics. These standard interfaces allow designers to change the semantics of connectors without having to make significant changes to the components. In addition, connectors are decomposed into building blocks that capture different aspects of the semantics of connectors. This makes it possible to support a library of pre-defined building blocks from which a wide variety of connectors can be constructed.

With the standard component interfaces and the reusable building blocks for connectors, designers can easily experiment with alternative choices of interaction semantics. Designers may construct connectors with specific semantics by combining a subset of the building blocks from the library. They can subsequently make changes to the connectors by selectively adding, removing or replacing one or more of the building blocks. With the standard component interfaces, such changes in the connectors often require no changes in the components.

Our approach also facilitates design-time verification. In particular, designers may wish to get feedback about the correctness of the overall system design while experimenting with alternative design choices of interaction semantics. With our approach, reusable models can be created for the connector building blocks, and since changes in the connectors do not often require changes in the components, component models can also be reused for most of the time. Therefore, our approach creates savings in model-construction time during design-time verification.

Section 2 shows how this approach can be realized to support the plug-and-play design of a family of message passing semantics. In Section 3, we discuss how this plug-and-play design approach facilitates design-time verification. Section 4 describes the related work, and Section 5 discusses the status of our work and some future directions.

2. PLUG-AND-PLAY WITH MESSAGE PASSING

Message passing is one of the most commonly used interaction mechanisms for distributed systems. Many languages, such as CSP [10], Occam [5], and Linda [4] incorporate message passing facilities. There are also message passing libraries such as MPI [16] and PVM [7]. Although the fundamental message passing semantics come from two basic operations, send and receive messages, there are a surprising number of variations in their semantics. For example, a message may be sent synchronously or asynchronously; a component that receives messages may block or continue when a requested message is not available. Other message passing semantics such as how messages are stored in the buffer, how they are delivered, and what information is relayed to the sender component and the receiver component may also vary.

Based on a study of the most commonly used message passing semantics, we have defined a set of building blocks for the construction of message passing connectors. This set of building blocks consists of different kinds of *send ports*, *receive ports*, and *channels* that together can be used to express a wide variety of message passing semantics. Figure 1 gives a few examples of the message passing building blocks we have defined.

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 1: A set of message passing building blocks

```
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 2: Standard component interfaces

As we can see from the description of the building blocks, channels are essentially message buffers that capture semantics such as the buffering and delivery of messages. A send port is a mediator between a sender component and a channel, and captures such semantics as whether a message should be sent synchronously or asynchronously or whether the sender should block when the message buffer is full. Different send ports provide different semantics by forwarding and interleaving the messages between the sender component and the channel in different ways. A similar notion applies to receive ports. To construct a message passing connector with specific semantics, we simply select the appropriate channel we are going to use to store and deliver messages, and then select the appropriate ports that components may send messages to and receive messages from.

Figure 2 shows the standard interfaces for components to send and receive messages. In Figure 2(a), a sender component waits for a *SendStatus* message from the connector after sending a message. This interface is designed to work with connectors that implement different semantics for sending messages. For example, in the case of asynchronous message passing, the connector should immediately return the *SendStatus* message to the sender component, while for syn-

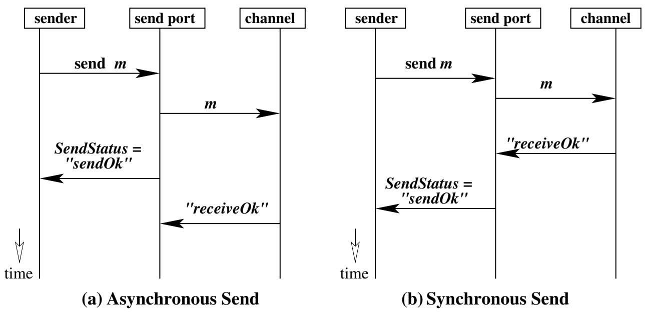

Figure 3: Example scenarios of message passing interactions (using send ports)

chronous message passing, the connector should not return the *SendStatus* message until the sender's message has been delivered. Such a difference is captured in a send port between a component and a channel. Using a notation similar to Message Sequence Charts, Figure 3 illustrates how a send port controls the interleaving of the messages between the component and the channel to give different interaction semantics. Notice that the same protocol is used between the sender component and the send port, and between the send port and the channel for both synchronous and asynchronous message passing. Switching between asynchronous message passing and synchronous message passing can be simply achieved by substituting the send port that is used with the other kind.

Similarly, in Figure 2(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). By always having the receive port send back an explicit status message to the receiver component, the same interface can be used for both blocking and nonblocking semantics. A blocking receive port does not send a success status message to the component until a message has been successfully received from the channel and can be delivered to the component. A nonblocking receive port sends a failure status message immediately to the component when there is no message currently available in the channel, allowing the receiver component to continue its execution.

With the standard interfaces, changes in such design decisions as the specific semantics for sending and receiving messages, or the behavior of the message buffers can be accomplished by simply replacing the send or receive ports, or the channels that are employed in the connector. The components do not have to be involved in these changes. This way, designers can easily experiment with alternative choices of different message passing semantics.

In this section, we have shown how we support the plug-and-play design for a family of message passing semantics. However, our approach is not restricted to message passing. In fact, we have extended the message passing building blocks to support other kinds of interaction mechanisms such as the publish/subscribe and RPC(remote procedure call). More details of this work can be found in [19].

3. VERIFICATION SUPPORT FOR THE PLUG-AND-PLAY DESIGN APPROACH

In this section, we describe how our plug-and-play design approach facilitates design-time verification. We first briefly introduce the modeling language and model checker we use to verify the designs. We then give a detailed discussion about how reusable models for the message passing building blocks are defined, how they can be composed to form different connectors, and how the connector models are composed with component models through components' standard interfaces. Finally, through a small example, we illustrate how these pre-defined building blocks can be used in the design and verification of a message passing system. With this example, we show that our plug-and-play approach not only makes it easy to change the message passing semantics represented by the connectors but also allows the re-use of models for verification.

3.1 Creating and Composing Building Block Models

For an initial evaluation of our approach, we use the SPIN [11] model checker to verify the system designs created using our plug-and-play approach. We define the formal models of the message passing building blocks in Promela, the input language of SPIN. These models are defined in such a way that they can be readily composed with other parts of the model of any system that uses these building blocks.

In Promela, communicating components are defined as processes using the keyword `proctype`. Communications between processes take place through channels that provide either buffered or synchronous (when the channel size is 0) message passing. A Promela channel can be declared using the keyword `chan`, along with the size of the buffer and the data type for each field of the messages that can be accepted by the channel. The following Promela code shows an example of a typical channel declaration and the basic operations for sending and receiving messages.

```
/* a channel of size 3 that takes messages of type short */
chan myChannel = [3] of {short};

/* sends a message of value 3 to myChannel */
myChannel!3;

/* receives a message from myChannel
 * and stores it in variable myMsg */
myChannel?myMsg;

/* receives a message from myChannel with a value
 * that matches the constant 3 */
myChannel?3;
```

With the send operation "!", the message is appended at the end of the channel, assuming space is available in the channel when the message is sent. With the receive operation "?", the first message in the channel is retrieved. When constants are used in one of the fields after "?", only messages with values that match the constants can be retrieved. The receiving process is blocked when the value of the first message in the channel does not match the constant specified. There are a number of variations on the send and receive operations supported by Promela. For example, with the receive operation "??", the first matching message in the channel will be retrieved. The receiver process does not block as long as there is at least one matching message in the channel.

```
typedef SynChan{
  chan signal = [0] of {InternalMsg};
  chan data = [0] of {DataMsg}
}
proctype SynBlSendPort(SynChan componentChan;
                       SynChan channelChan){
      DataMsg m;
      do
      :: componentChan.data?m;
         m.sender_id = _pid;
         do
         :: channelChan.data!m;
            if
            :: channelChan.signal?IN_OK,_;
               break;
            :: channelChan.signal?IN_FAIL,_;
            fi;
        od;
        channelChan.signal?RECV_OK,eval(_pid);
        componentChan.signal!SEND_SUCC,-1;
     od;
}
```

Figure 4: Promela model for a synchronous blocking send port

```
proctype AsynNbSendPort(SynChan componentChan;
                        SynChan channelChan){
        DataMsg m;
        do
        :: channelChan.signal?_,eval(_pid);
        :: componentChan.data?m;
           componentChan.signal!SEND_SUCC, -1;
           m.sender_id = _pid;
           channelChan.data!m
        od
}
```

Figure 5: Promela model for an asynchronous non-blocking send port

It is important to notice the difference between the Promela channels and the channels used as building blocks for connectors in our design approach. Promela channels are used for sending and receive messages between Promela processes. Promela channels can only support the simplest kinds of message buffers such as a FIFO queue. On the other hand, our channels are architecture-level building blocks for connectors that can essentially capture arbitrary interaction semantics among components, and therefore are not necessarily message buffers. For example, a channel in a publish/subscribe connector may represent an event pool where delivery of events is based on subscription. Even when our channels are used as building blocks for message passing connectors, they can be much more complicated than simple message buffers. Such a channel may be able to handle messages based on their priorities, notify components of the current buffer status, or deliver messages to a group of interested components. In the following discussion, we will always refer to channels supported in Promela as *Promela channels* to distinguish them from the architecture-level channels in our approach.

For the purpose of this approach, all the ports, channels, and components in the design are modeled as communicating processes in Promela. We use Promela channels to model all the internal communications between components and ports, and between ports and channels.

Figure 3.1 shows the Promela model for a synchronous blocking send port. The port is modeled as a Promela process (`proctype`) that takes two parameters of type `SynChan`. `componentChan` is a set of two Promela channels, `data` and `signal`, for the communication between the send port and the component. `componentChan.signal` is used for communicating internal signals such as the status of the buffer and the status of message delivery; `componentChan.data` is used for communicating messages that actually contain application-specific data. The two Promela channels in `channelChan` are used in a similar way for the communication between the send port and our architecture-level channel.

As we have described in Section 2, a send port is a mediator between a sender component and a channel that intercepts their communication of data messages and message delivery status signals. In this model, the send port receives a data message `m` from the component (`componentChan.data?m`), forwards it to the channel (`channelChan.data!m`), and then waits for a signal back from the channel that indicates whether the message can be properly stored in its buffer. Such a signal could either be `IN_OK` or `IN_FAIL`. Since this is a synchronous blocking send port, it keeps sending message m to the channel until the message is successfully stored in the channel, and an `IN_OK` signal is received (this can be seen from the `do...od` loop). It then waits for a `RECV_OK` signal from the channel that indicates the successful delivery of the message m. Finally, after receiving both `IN_OK` and `RECV_OK` signals from the channel, the synchronous blocking send port sends the send status message (`SEND_SUCC`) back to the sender component. Notice that one port can only be used by one component, but multiple ports may be talking to a single channel. Therefore, a send port has to check if a delivery confirmation `RECV_OK` from the channel is addressed to itself by making sure the tag matches with its own `_pid` (`eval(_pid)` gives the constant value of `eval(_pid)`). Since the `IN_OK` and `IN_FAIL` signals are issued immediately after the port tries to send message m to the channel, they will always be properly addressed to the send port.

As one may have guessed, the definition of an asynchronous blocking send port is similar to its synchronous counterpart except that an asynchronous send port immediately sends `SEND_SUCC` to the component after receiving `IN_OK` from the channel. Similarly, for a nonblocking send port, `SEND_SUCC` may be sent to the component before the message has been stored in the buffer by the channel. Figure 3.1 shows the Promela model for an asynchronous nonblocking send port. This port receives a message m from the component and immediately returns a `SEND_SUCC` status signal to the sender component, regardless whether message m will be successfully stored in the channel or eventually received by the a receiver component. In fact, the port ignores any signals sent from the channel using a wildcard receive `channelChan.signal?_,eval(_pid)` (in Promela, `_` can be matched with anything).

Figure 6 illustrates what a component model may look like to send messages through a send port. In this model, the component sends its message to the send port and immediately waits for a status signal back. Depending on the specific semantics of the send port the component is sending messages through, the status signal may be returned at different stages of message delivery and may indicate either a failure or success. But no matter what kind of send ports the component is communicating with, the same interface can be used. This allows the model of the port to be

```
proctype aSendComponent(SynChan sendPortChan){
     DataMsg myMsg;
        ...
     sendPortChan.data!myMsg;
     /* sendStatus could be SEND_OK or SEND_FAIL */
     sendPortChan.signal?sendStatus,_;
        ...
}
```

Figure 6: A sender component

```
proctype aRecvComponent(SynChan recvPortChan){
     DataMsg myMsg;
        ...
     recvPortChan.data!recvRequest;
     /* recvStatus could be RECV_OK or RECV_FAIL */
     recvPortChan.signal?recvStatus,_;
     /* myMsg should not be used when recvStatus is RECV_FAIL */
     recvPortChan.data?myMsg;
        ...
}
```

Figure 7: A receiver component

changed or replaced without having to change the model of the component.

Similarly, Figure 7 shows the component interface for receiving a message. In this model, a receiver component sends a receive request to the receive port, and it tries to receive a status signal from the port, followed by a data message delivered by the channel. If `recvStatus` indicates `RECV_SUCC`, the message `myMsg` is the actual requested message delivered by the channel. If `recvStatus` indicates `RECV_FAIL`, the message `myMsg` is an empty message sent by the receive port as a stub, and therefore should not be used by the component.

Such an interface for receiving messages makes it possible to support both blocking and nonblocking semantics. Figure 8 shows the Promela model for a blocking receive port. The receive port starts by waiting for a `recvRequest` message from the component. When it arrives, it tries to send the request to the channel until the request is confirmed by the channel (indicated by the `OUT_OK` signal). After the port successfully retrieves a message m from the channel (`channelChan.data?m`), it then sends a `RECV_SUCC` confirmation to the receiver component followed by the message m delivered by the channel. A nonblocking receive port would send a `RECV_FAIL` signal immediately to the component when the receive request is rejected by the channel (indicated by signal `OUT_FAIL`). It then sends an empty message to the receiver component as a stub to accormodate the standard interface of the receiver component.

Note that other variations of receive ports may be supported. For example, a receive port (whether blocking or nonblocking) may ask the channel to keep the message (*copy receive*) that has been received in the buffer or to remove it (*remove receive*)[1]. A receive port may also support *selective receive* where a tag is used as the matching criteria to retrieve a message from a channel.

For message passing, channels are essentially buffers that store and deliver messages. There are a number of different properties of a message buffer that may affect the overall correctness of the system. For example, some channels may notify the sender component when its buffer is full so that the component may choose to send at a different moment; other channels block the sender until space is available in the buffer; a third kind of channels may simply drop messages that are sent after its buffer becomes full without notifying the sender. Of course channels may have buffers with different sizes and may implement different message delivery policies. We have defined the Promela models for a number of message passing channels that implement a variety of such semantics.

[1]In order for this to work, the channel has to provide such service

```
proctype BlRecvPort(SynChan componentChan;
                    SynChan channelChan){
      DataMsg recvRequest,m;
      do
      :: componentChan.data?recvRequest;
         do
         :: channelChan.data!recvRequest;
            if
            :: channelChan.signal?OUT_OK,_;
               channelChan.data?m;
               break;
            :: channelChan.signal?OUT_FAIL,_;
            fi;
         od;
         componentChan.signal!RECV_SUCC,-1;
         componentChan.data!m;
      od;
}
```

Figure 8: Promela model for a blocking receive port

Figure 9 shows our model for a *single-slot-buffer*, a message buffer that only holds one message. The process model of a message passing channel takes two parameters of type `SynChan`. `senderChan` is used for the communication with the send ports that components are using to send messages to the channel. `receiverChan` is used for the communication with the receive ports that components are using to receive messages from the channel. The channel accepts a receive request from a receive port or a message forwarded by a send port, and handles them according to the current status of its buffer. In this particular implementation, the channel notifies the send port with an `IN_FAIL` signal when its message buffer is full, and notifies the receive port with an `OUT_FAIL` signal when no requested message is currently available in the buffer. This channel model can be easily composed with a number of send and receive ports by matching the Promela channels `channelChan` used by the send ports and the `channelChan` used by the receive ports with the `senderChan` and `receiverChan` used by the channel, respectively.

Figure 9 only gives an example of a fixed-sized message buffer. It is possible to create a model for a channel that have a message buffer of an arbitrary size. In this case, the Promela process of the channel takes an additional parameter that specifies the size of the buffer. Semantics of how messages are stored and delivered also need to be implemented. In fact, in addition to the single-slot buffer, we have also defined the Promela models for a channel that stores and delivers messages in a FIFO order, and one that handles messages based on their priorities. The models for both types of channels can be instantiated with the size of the message buffer used in the channel. This allows a range of similar message passing channels to be defined by parameterizing the same model. **[SW: If we have space, we will show the mode for a FIFO queue as appendix.]**

As we have described above, building blocks are mod-

```
proctype single_slot_buffer (SynChan senderChan;
                             SynChan receiverChan){
    DataMsg recvRequest, m, buffer;
    bool buffer_empty = 1;
    do
    :: receiverChan.data?recvRequest;
       if
       :: (!buffer_empty && !recvRequest.selective)
          || (!buffer_empty && recvRequest.selective
              && buffer.selectiveData
                 == recvRequest.selectiveData) ->
            receiverChan.signal!OUT_OK,-1;
            receiverChan.data!buffer;
            senderChan.signal!RECV_OK,buffer.sender_id;
            if
            :: recvRequest.remove ->
                 buffer_empty = 1
            :: else
            fi
       :: else ->
            receiverChan.signal!OUT_FAIL,-1
       fi
    :: senderChan.data?m;
       if
       :: buffer_empty ->
           senderChan.signal!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 ->
           senderChan.signal!IN_FAIL,-1
       fi
    od
}
```

Figure 9: Promela model for a single-slot buffer channel

eled as communicating Promela processes. They can be easily composed to form a connector model and then composed with the component models by matching the specific Promela channels associated with them. When design decisions about the semantics of a connector are changed and the system design needs to be re-verified, formal models of the system can be modified by simply replacing the Promela processes of the existing building blocks of the connector with those of the new ones. For example, when different semantics for sending messages are needed for a component, we can use the model for a different send port in place of the existing one, and pass in the same Promela channels that allow the new send port process to communicate properly with the component process. With component models implementing the standard interfaces, such a change will not require any changes in the models of the components.

Through an example, Section 3.2 shows how the way we have defined the standard component interfaces, and the way we have modeled our building blocks helps create savings in model-construction time, when verification is applied to designs that are subject to frequent changes in their interaction semantics.

3.2 The single-lane bridge example

Consider a bridge that is only wide enough to let through a single lane of traffic at a time. An appropriate traffic control mechanism is necessary to prevent crashes on the bridge. For this example, we assume traffic control is managed by two controllers, one at each end of the bridge. Communication is allowed between two controllers as well as between cars and

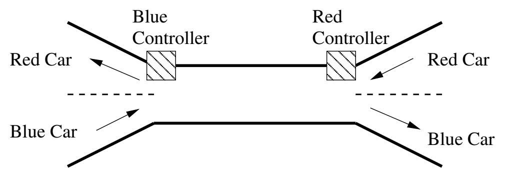

Figure 10: A single-lane bridge with two controllers

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 10.

There are a number of possible ways to control the traffic on the bridge. We first consider a very simple traffic control mechanism called "exactly-N-cars-per-turn". With this version of the bridge system, controllers take turns to let cars enter the bridge, and at each turn, exactly N cars are allowed to enter the bridge. 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 turns, 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. Notice that with this version of the bridge example, no communication is required between the two controllers.

A more efficient traffic control mechanism, which we refer to as "at-most-N-cars-per-turn", may allow turns to be switched immediately when no more cars are waiting to enter the bridge from the end that is currently in control, before the controller's count reaches N. In this case, communications are necessary between two controllers to notify the yielding of turns. No matter what traffic control mechanism is used, we want to make sure the bridge is safe, that is, no cars traveling in the opposite directions can be allowed on the bridge at the same time. Designing a bridge system that ensures the safety property requires careful design of the specific interaction semantics for the connectors between the two controllers, and between the cars and the controllers.

In particular, a designer may have to decide whether it is more appropriate to use message passing or event-based notification for the communication between components; whether the communication between cars and controllers need to be synchronous or can be asynchronous; if message passing is chosen, what types of buffers should be used to store messages; what happens if a message gets dropped by a buffer, and so on. It is very easy to make mistakes on such matters when designing appropriate interaction semantics. Design-time verification can be very useful in evaluating the appropriateness of these design decisions. In particular, the designer can use verification to check the safety property of the bridge against an initial design of the system. When a violatin is reported, the designer may have to experiment with other alternative choices of interaction semantics and re-verify the system design to make sure the safety property still holds.

For this example, we choose message passing to handle the communications between components. Specifically, a

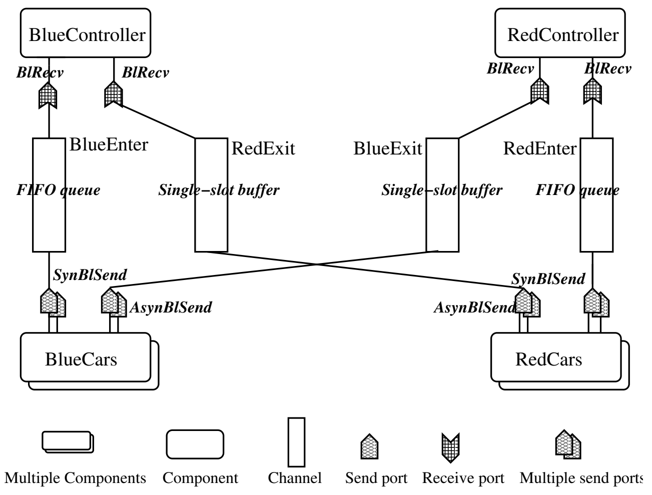

Figure 11: The architecture design of the "exactly-N-cars-per-turn" single-lane bridge problem

blue car component sends an `enter_request` message to the blue controller when it tries the enter the bridge and sends an `exit_request` message to the red controller when it exits the bridge. Similarly, a red car component sends an `enter_request` message to the red controller when it tries to enter the bridge and sends an `exit_request` message to the blue controller when it exits the bridge. For the "exactly-N-cars-per-turn" version of the bridge example, no message passing is needed between two controllers. 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, `enter_request` and `exit_request` messages need to be buffered in the connectors between car components and controller components.

Figure 11 shows the design of the "exactly-N-cars-per-turn" single-lane bridge example. In this design, interactions between components are described using the message passing building blocks. Since we want to make sure that an enter request has been received and acknowledged by the controller before a car component can enter the bridge, a synchronous blocking send port is used for a car component to send the `enter_request` message. These messages are buffered in a FIFO queue channel so that the requests are processed by the controller in a first-in-first-out order. A controller component uses a blocking receive port to receive `enter_request` messages from different car components. For the exit request messages, we use asynchronous blocking send ports since in this case, no acknowledgement is needed. But we do need to make sure the exit notification message has been safely stored in the channel so that eventually it will be delivered to the controller. In this case, we can use a blocking asynchronous send port so that the car component is blocked when the message buffer in the channel is full. And a single-slot buffer channel defined in the previous section can be used to handle exit request messages. Finally, a blocking receive port is used by a controller component to handle exit request messages.

To make sure that our bridge system functions properly, that is, it does not run into a deadlock state or cause cars from opposite directions to crash, we can use finite-state verification to check the design. Any misuse of the message passing building blocks may cause the verification to report a violation of the properties. For example, if an asynchronous blocking send port is used for sending enter requests in place of a synchronous blocking send port, a car can head on to the bridge without being acknowledged by the controller, causing a possible crash on the bridge. If a nonblocking send port is used for sending exit requests and a channel that drops new messages when its buffer is full is used for storing those messages, the system may run into a deadlock state where a controller is blocked waiting for an exit request message which has been lost.

To apply design-time verification using Spin, the Promela model of the overall system design needs to be constructed. With our approach, the system design is composed of components and various message passing building blocks. Therefore, a system model is simply a composition of all the Promela models for the message passing building blocks and components in the system. Specifically, models of the selected message passing building blocks are pre-defined (as described in Section 3.1) and can be simply included in the system model at the verification time. In general, our approach expects designers to provide formal models for the components in a design that implement the standard interfaces. These component models can often be automatically generated from their design. For the purpose of this example, we simply use hand-constructed Promela models for the car components and controller components. To allow the component models to be composed properly with the building block models, appropriate Promela channels are used to set up the connections between component processes and building block processes at the start of the Promela system. Due to space limitation, the complete Promela model for this vesion of the bridge example is presented in [19]. The safety property of the bridge example is described in LTL (Linear Temporal Logic), which can then be checked by Spinagainst the Promela model of the system. Possible deadlock of the system can be automatically reported by Spin.

As we can see from this example, the pre-defined building block models can be easily composed with component models to create a system model. These pluggable models also make it easier to make changes in the model, especially when such changes only involve the semantics of the connectors. Suppose that we now wish to modify the previous version of the bridge example so that controllers may give higher priority to emergency vehicles when they are issuing acknowledgement for enter requests. To make the change, we simply replace the FIFO queue channel used to buffer enter request messages with a channel that implements a priority queue. With this new channel, enter requests from emergency vehicles will always be at the front of the queue, allowing the controllers to handle them before enter requests from non-emergency vehicles. To apply verification to make sure that this change does not violate the safety property of the bridge, the model for the new design can be created by simply replacing the model for the FIFO queue channel with the pre-defined model for a channel that implements a priority queue. With our approach, neither the component models nor any building block models need to be reconstructed; only the composition of these models need to be re-computed.

Of course, not all modifications to a system require only simple changes in the connectors. Suppose that, in order

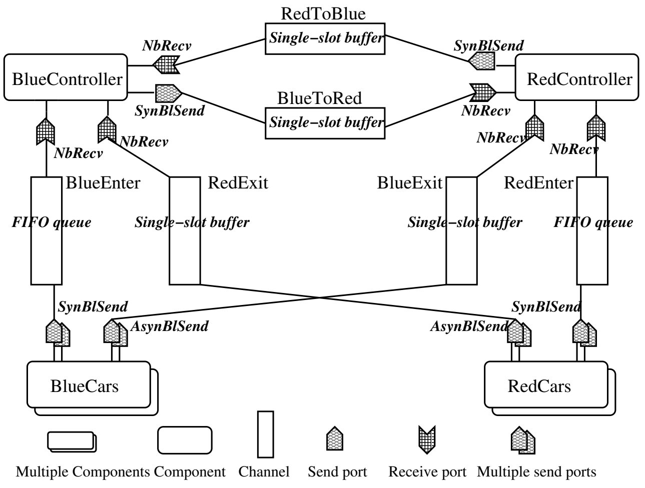

Figure 12: The architecture design of the "at-most-N-cars-if-waiting" single-lane bridge

to improve traffic flow, the designer wishes to change the "exactly-N-cars-per-turn" version of the bridge system into the "at-most-N-cars-per-turn" version. This requires the addition of new communication between the controllers and the modification of the controller components. 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 12 shows a possible design for the modified system, with two new connectors between the controllers, 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 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 connectors 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 the building blocks.

3.3 Discussion

From the examples illustrated in the previous section, we can see that our verification support works in the same plug-and-play manner as the design approach we have proposed. Models of components and connector building blocks are defined independently and can be plugged together in many different ways to represent different semantics. Model reconstruction is minimized when changes have to be made to the components or the connectors. Having reusable models for building blocks of connectors and the models of components stay relatively stable when only interactions are changed, we reduce the cost of repeated verification in the iterative design process, and therefore make it easier and more efficient to experiment with alternative choices of design of interaction mechanisms, and eventually help achieve a better system more efficiently.

Be aware that the way we have modeled the various message passing building blocks is not necessarily the most efficient. Special care needs to be taken to allow these models to work with the standard component interfaces and the different kinds of channels. In addition, for demonstration purposes, these models are defined to be intuitive and easy to understand in terms of the protocols used between building blocks. Such implementation may often have redundant data structures (e.g. channels, messages) or statements, which may result in inefficient code and the increase of the state space for the overall system model. Another problem related to this one is that by breaking connectors into ports and channels that are modeled as communicating processes, we introduce additional concurrency to the model, which may contribute to the state explosion problem with finite-state verification. Therefore, our approach may be restricted to only very small systems. Optimizations on the pre-defined building block models need to be applied right before the models are composed. Additional optimizations may be applied after the models have been composed. However, these optimizations can be applied in transparent to the users of the building block models. [**SW: still thinking about some concrete examples for such optimizations**]

Another concern of our approach is the difficulty with tracing counterexamples. In finite-state verification, a counterexample is often provided when a property violation is found. A counterexample gives an execution of the model that leads to the violation of the property. With our approach, tracing an error may require going into the models of the building blocks which requires a good understanding of the semantics. This is often not the case when interactions are specified by putting together a set of pre-defined building blocks. It would be helpful if our approach can indicate which of the building blocks of the connectors may be causing the property violation. For example, a deadlock in a system may be due to the use of a message buffer that drops messages inadvertently. This way, designers can directly investigate the building blocks that are used in the system and experiment with alternative choices using our plug-and-play approach until the problem is fixed.

Note 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 [12] 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.

4. RELATED WORK

[**SW: Related work still needs some work.**]

Our approach differs from previous work on architectural evolution (e.g., [13,18]) in our focus on supporting the exploration of different interaction semantics at the design stage and our emphasis on modeling and verification. A number of approaches have also been proposed for assembling existing components into applications, including mediators [17], active interfaces [9], and various techniques for wrapping components. Our interest here is more in the alternative design choices of interaction semantics of connectors and less on the adaptation of existing components to interact with each other.

There are a few existing work on specifying complex connectors and modeling them for verification. The Wright architecture description language [1], for example, used the CSP process algebra to describe arbitrary connectors, and the Architectural Interaction Diagrams (AIDs) of Ray and Cleaveland [15] 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 [2, 14]. 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 a variety of interaction semantics 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.

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. [3,8,20]) 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.

5. CONCLUSION AND FUTURE WORK

While designing a software architecture, choosing the appropriate interaction semantics for the connectors in a system tends to be very difficult given the large design space. In this paper, we describe an approach that allows easy experimentation with alternative design choices of interaction semantics. Our approach supports a library of ready-to-use building blocks for constructing connectors with a wide variety of interaction semantics. We also support a set of standard interfaces that components may use to communicate with different connectors and therefore minimize the changes to the components when connectors are changed. To support design-time verification, we create parameterizable and reusable formal models for the building blocks. System models are composed from component models and the pre-defined building block models. Using our approach, designers may experiment with their choice of design for various interaction semantics by plugging and playing with the building blocks in the connectors. And they can use design-time verification to evaluate their design decisions on interaction semantics. While this process may repeat, our approach allows considerable reuse of the models of components and connectors.

There are a few ongoing work with this approach. First, we have started the implementation of this approach in an architecture design environment called AcmeStudio `http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html`, developed at CMU. Our tool is going to use the same notation for components, channels and ports in Acme [6] but with an extension of their semantics. We will also integrate the architecture design environment with finite-state verification, in particular, the SPINmodel checker, to provide the capability of running verification directly on the designs created in Acme Studio. Another ongoing work is extending the current approach to support other kinds of interaction mechanisms such as publish/subscribe and remote procedure call.

One important future work related to verification is the development of optimizations to reduce the system models that are composed from 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. Some of the other future work may include the formalization of our definition for building blocks so that designers may be able to define their own building blocks. Finally, more extensive case studies need to be done to evaluate the effectiveness of our approach.

6. 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.

7. REFERENCES

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

[2] 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.

[3] 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.

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

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

[6] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, Nov. 1997.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.