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. Changes in the semantics of a connector can be accomplished by adding new building blocks to the connector, or by removing or replacing some of its existing building blocks. 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 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 then 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, 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 and re-verifying the overall system before a satisfactory design is achieved.

One major obstacle to the realization of this vision of design and design-time 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. Experimenting with alternative choices of interaction semantics tends to be difficult and inefficient when changes made in the interactions require nontrivial 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 the semantics of interactions intertwined with the semantics of computations, 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 defines a small set of standard interfaces between components and connectors that allows the interaction semantics represented by a connector to be changed with no, or minimal, change to the components. Connectors with specific interaction semantics can be specified by selecting and composing a subset of pre-defined building blocks from a library. The semantics of a connector can be modified by adding new building blocks, or removing or replacing some of its existing building blocks. This approach allows system designers to experiment more easily with alternative design choices in a plug-and-play manner. It also facilitates design-time verification by saving on the cost of re-constructing models for both connectors and components.

Section 2 gives an overview of this plug-and-play design approach. In particular, we illustrate how it is realized for message passing, one of the most commonly used interaction mechanisms. In Section 3, we discuss how this plug-and-play design approach facilitates design-time verification. Section 4 describes related work, and Section 5 discusses the status of our work and some future directions.

2. THE PLUG-AND-PLAY APPROACH

2.1 Overview

In the previous section, we have noted the difficulties with choosing appropriate semantics for the connectors in a system design. Our approach tries to address these difficulties by providing system designers with an efficient way to experiment with alternative design choices for connectors and to use finite-state verification to evaluate the correctness of their designs.

To achieve this, we first introduce a small set of standard interfaces that components can use to communicate with each other through different connectors. This set of standard interfaces allows connectors to be modified or replaced without causing significant changes in the components. To support the standard interfaces, we decompose connectors into *ports* and *channels* that capture different aspects of the interaction semantics represented by connectors. Ports work directly with the components' standard interfaces and are responsible for hiding the semantic differences between connectors from the components. Ports capture such aspects of interaction semantics as under what condition a component should be blocked or whether a component should wait for an acknowledgement after a communication with other components. While such semantics can be easily embedded in components' computation, with our approach, they can be captured in the ports, as part of the connectors. This way, changes in the interaction semantics can be made completely in the connectors and independently of the components' computation. Channels are used to represent the other aspects of interaction semantics represented by a connector. For example, a channel may represent a message buffer for message passing communication or an event service used in publish/subscribe systems.

The decomposition of connectors into ports and channels not only makes it possible to support the standard component interfaces, but also makes it easier to provide a library of reusable building blocks from which a wide variety of connectors can be constructed. With our approach, designers can easily experiment with alternative design choices of interaction semantics in a plug-and-play manner. Constructing a connector with specific semantics is a matter of combining a subset of the building blocks from the library. Changes can be made to a connector by selecting a new subset of building blocks for the connector. With the standard component interfaces, such changes in the connectors often require no or few changes in the components.

Our approach uses finite-state verification to provide designers with feedback about the correctness of the overall system design while they experiment with alternative design choices. The plug-and-play style of design facilitates verification in a number of ways. First, since changes in the connectors usually do not require changes in the components, component models often do not have to be reconstructed when verification needs to be re-applied because of the changes. In addition, pre-defined formal models can be constructed for the library of building blocks. These models can be reused in the model of any system that uses these building blocks. Therefore, our approach can create significant savings in model-construction time during design-time verification.

The next section illustrates how the approach described above can be used to support the plug-and-play design of a family of message passing semantics. More details of this approach can be found in [21]. Section 3 discusses how design-time verification is supported, and how the plug-and-play approach facilitates verification.

2.2 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 [11], Occam [5], and Linda [4] incorporate message passing facilities. There are also message passing libraries such as MPI [17] and PVM [8]. Although the fundamental message passing semantics come from two basic operations, send and receive, 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 aspects of message passing semantics such as how messages are stored in a buffer, how they are delivered, and what kinds of information regarding the status of message delivery are 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.

As we can see from the description of the building blocks in the figure above, 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. It 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

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

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. As shown 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, allowing the component to continue its execution. For synchronous message passing, however, the connector returns the *SendStatus* message after the sender's message has been delivered, thereby blocking the component until a message is received. 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 achieved simply by substituting one kind of send port for the other kind.

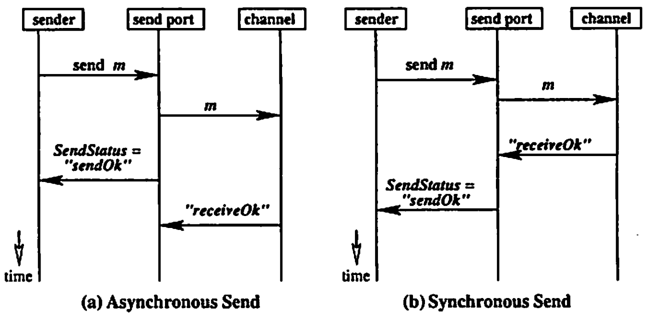

Figure 3: Example scenarios of message passing interactions (using send ports)

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 a null 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 the status back 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.

This section has shown how our approach supports plug-and-play design for a family of message passing semantics. 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. Our approach, however, is not restricted to message passing. In [20], we describe how to extend this approach to support other kinds of interaction mechanisms, such as the publish/subscribe and RPC (remote procedure call).

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

Before showing how our plug-and-play approach facilitates design-time verification, we first briefly introduce the modeling language and model checker we have chosen 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.

3.1 Creating and Composing Building Block Models

For an initial evaluation of our approach, we have chosen the SPIN [12] 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 our library using 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 when the channel is not full; otherwise, the sending process is blocked. 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 receiving process does not block as long as there is at least one matching message in the channel.

It is important to notice the difference between the Promela channels and the channels used as building blocks for connectors in our approach. Promela channels are used for sending and receiving messages between Promela processes. Promela channels can support only a limited number of simple message buffers, such as FIFO queues. On the other hand, our channels are architecture-level building blocks for connectors that can capture essentially 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 the native channels in Promela as *Promela channels* to distinguish them from the architecture-level channels in our approach.

All the ports, channels, and components in a design are modeled as communicating processes in Promela. We use Promela channels to model the internal communications between components and ports and between ports and channels.

```
/* internal communication signals */
mtype = {SEND_SUCC, SEND_FAIL, IN_OK, IN_FAIL, OUT_OK,
        OUT_FAIL, RECV_OK, RECV_SUCC, RECV_FAIL};

typedef InternalMsg{
  mtype signal;
  byte port_pid;
}

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,eval(_pid);
               break;
            :: channelChan.signal?IN_FAIL,eval(_pid);
            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

Figure 4 shows the Promela model for a synchronous blocking send port. We first define a set of signals that are used to represent the status of sending and receiving a message. These signals are defined as the enumerated type mtype in Promela. The type SynChan defines two Promela channels that are used for communications between components and ports, and between ports and channels. The Promela channel signal is used to communicate message delivery status signals, and the Promela channel data is used to communicate application-specific data messages. The port is modeled as a Promela process (proctype) that takes two parameters of type SynChan, one of which represents the set of Promela channels for the communication with the component (component), and the other set of Promela channels for the communication with the channel (channelChan).

The main part of the Promela code for the port is a loop in which the port accepts a message from the component and forwards it to the channel, and then, when the message has been accepted by the channel, forwards the appropriate status message back to the compoent. As we can see from the model in Figure 4, in Promela, a block of repeating statements is enclosed in a pair of do and od keywords. A number of statement blocks can be selectively executed when the loop is entered. The symbol `::` is used to identify the beginning of a selective block. A block is executable when the first statement in the block is enabled. When more than one block is executable, one of them is selected arbitrarily. In our model for the send port, we only need one selective block, since the send port only has one thing to do, that is, to wait for a message m to be sent from the component and then deliver it to the channel. When the component is ready to send a message, the statement `componentChan.data?m` is enabled and therefore the rest of the statements can be executed.

As we can see from the model, after receiving a message from the component, the send port attaches its own process ID `_pid` to the message. Since one channel may be connected to multiple send ports, this `_pid` will be sent along with the data message to the channel so that the channel can use it to notify the appropriate port of the delivery status of the message. Any status signals that are addressed to this port will be tagged with its process ID number.

The send port then tries to forward the message m to the channel (`channelChan.data!m`). After that, it 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`. To model this nondeterministic choice, we use the selective statement `if...fi` in Promela that allows a selective execution of one of its blocks. The semantics of how blocks are selected are the same as as for the `do...od` statement, as we have described above. The send port makes sure that the signals from the channel are indeed addressed to itself by matching its own process ID with the tag attached to the signal that is sent back. This is done by specifying its process ID as a constant matching criteria in a receive statement. For example, the receive statement `channelChan.signal?IN_OK,eval(_pid)` will only be executed when both constants `IN_OK` (an enumerated type in Promela) and `eval(_pid)` (`eval(_pid)` gives the constant value of `_pid`) match the values in a message that can be retrieved from the channel.

Since this is a synchronous blocking send, if the channel sends back an `IN_FAIL` signal, the port has to send the message to the channel again and keep trying until an `IN_OK` signal is received indicating that the message has been successfully stored in the channel. It then can break out of the loop and wait for a `RECV_OK` signal from the channel which indicates that a receiver has successful received the message. 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 since the component process does not care about the ID of the port, we simply send an invalid process ID number -1 along with the `SEND_SUCC` signal.

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 5 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 any value).

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

Figure 6: A sender component

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

Figure 7: A receiver component

Figure 6 shows the component interface for sending messages through a send port. 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 (`SEND_FAIL`) or success (`SEND_SUCC`) . But no matter what kind of send ports the component is communicating with, the same interface can be used. As noted previously, this often allows the model of the port to be 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 then 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

```
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

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 accommodate the standard interface of the receiver component.

Note that other variations of receive ports can be defined similarly. 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*). A receive port may also support *selective receive* where a tag is used as the matching criteria to retrieve messages 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 channel 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.

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.

```
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

Figure 9 only gives an example of a fixed-sized message buffer. It is possible to create a model for a channel that has 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. For example, 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 for 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.

As we have described above, ports and channels are modeled as communicating Promela processes and they can be connected through specific Promela channels that handle the communications between them. To construct the model for a connector, we can simply compose the pre-defined Promela processes for its building blocks by matching the specific Promela channels associated with them. Component models and connector models can be composed in a similar way. 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 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 substitute a different send port for the

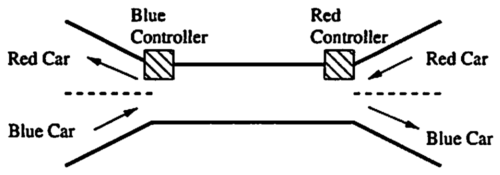

Figure 10: A single-lane bridge with two controllers

existing one, and pass in the same Promela channels that allow the new send port process to communicate properly with the Promela process for the component. In Section 3.2, we give an example illustrating how system models can be constructed from the building block models and how they can be re-constructed when changes are made in the design of connectors.

Note that the Promela models we have created for the message passing building blocks are not necessarily the most efficient ones and there may be a number of different ways to model them in Promela. Instead of aiming for elegance or efficiency, our models are coded to clearly reflect the protocols that are used by the building blocks. These models can often be simplified and optimized for verification in a number of ways. We briefly discuss some of the possible optimizations in Section 5.

Also note that our approach is not tied to any particular model checker or modeling language. By using Promela and Spin, we are only showing one possible way of modeling our building blocks and applying design-time verification. In fact, we have defined the same set of building blocks in the process algebra FSP and used LTSA (the Labeled Transition System Analyzer) [13] to verify the system designs. Somewhat different strategies may be appropriate when modeling the building blocks in a different modeling language.

3.2 The single-lane bridge example

In this section, we use an example to illustrate how designers may use the building blocks and the techniques we have described above in the design and verification of a small message passing system. In particular, we show how design-time verification may benefit from our plug-and-play approach by saving on model construction time when repeated changes are made to the connectors in a software architecture.

As an 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 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. Blue cars send enter requests to the blue controller when they try to enter the bridge and notify the red controller when they exit the bridge. A similar situation applies to red cars.

There are a number of possible ways to control the traffic on the bridge. For a simple version of the bridge example, which we refer to as "exactly-N-cars-per-turn", controllers may take turns to allow some fixed number (N) of cars from

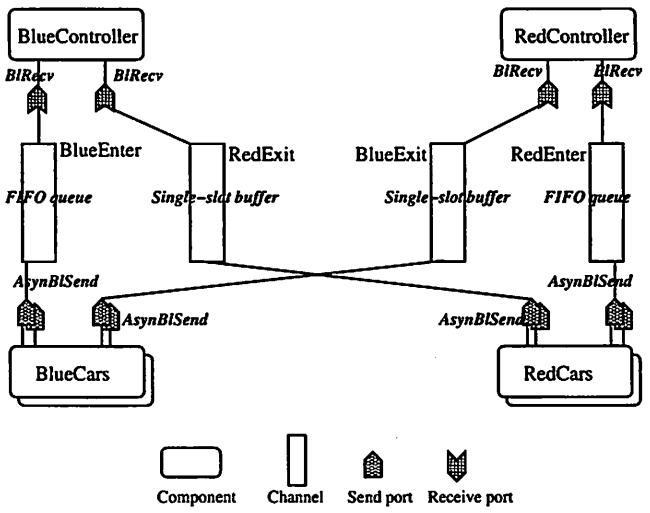

Figure 11: An initial design of the "exactly-N-cars-per-turn" single-lane bridge example

their side to enter the bridge. A more efficient single-lane bridge system, which we refer to as "at-most-N-cars-per-turn", may allow turns to be yielded immediately to the other controller by the controller that is currently sending cars to cross the bridge, when there are no cars waiting to cross the bridge from its side. 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 this safety property requires a careful design of not only the components (cars and controllers) in the system, but also the specific semantics of the connectors used for the interactions between the components.

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 needs 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.

For this example, message passing seems to be a natural choice for the communications between components, but we still have to make sure the appropriate message passing semantics are chosen for each connector. With our approach, this can be achieved by selecting and composing a subset of the message passing building blocks from the library for each connector, and using design-time verification to make sure that our decisions do not yield a system that violates the safety property of the bridge.

Figure 11 shows an initial design of the "exactly-N-cars-per-turn" single-lane bridge example. In this design, asynchronous message passing is chosen for both the communication between the car and the controller on its entering side and the communication between the car and the controller on the other side. In this case, asynchronous blocking send ports are used for sending enter and exit request messages from the cars to the controllers. A FIFO queue channel is selected for buffering the enter request messages that are sent from different car components to the same controller, so that the requests are processed by the controller in a first-in-first-out order. A single-slot buffer channel may be used for exit request messages. Finally, blocking receive ports are used by each controller component to process enter and exit request messages. Notice that with this version of the bridge example, no communication is necessary between the two controllers.

To make sure that our bridge system does not cause cars traveling from opposite directions to crash, we can use verification to check our design. In this case, not surprisingly, verification reports a violation of the property. The cause of this violation is obviously that we have selected a wrong type of send port for sending enter request messages. Instead of using an asynchronous blocking send port, we should have used a synchronous blocking send port so that the car component waits for an acknowledgement from the controller before it tries to enter the bridge. With our approach, the erroneous design can be easily corrected by replacing the asynchronous blocking send ports for sending enter requests with synchronous ones, and no changes in the components are necessary. Verification needs to be applied again to confirm that the system now satisfies the property. With our approach, re-applying verification does not require the complete re-computation of the system model.

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 system design and the component models should implement the standard interfaces defined in our approach.

In principle, models of the components can be automatically extracted from their designs in some suitable language. For the purpose of this example, however, we constructed Promela models of the car and controller components manually. 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 limitations, the complete Promela model for this version of the bridge example is not given here, but it can be found in [20]. The safety property of the bridge example is described in LTL (Linear Temporal Logic), which can then be checked by SPIN against the Promela model of the system.

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, in order 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.

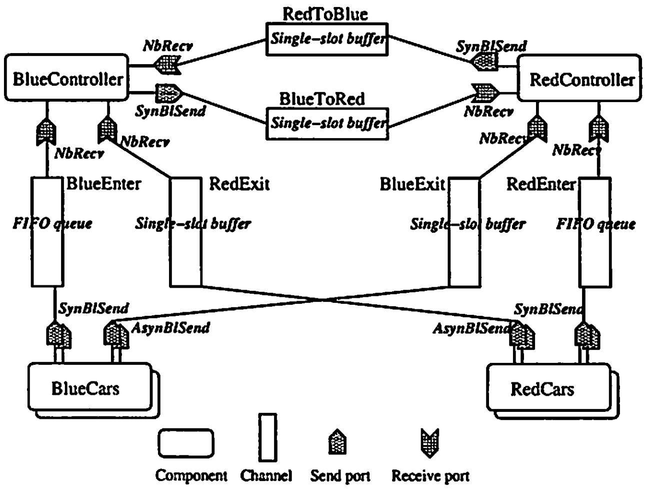

Figure 12: The architecture design of the "at-most-N-cars-if-waiting" single-lane bridge example

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.

From this single-lane bridge example illustrated above, we can see that our verification support works in the same plug-and-play manner as the design approach we have proposed. 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.

4. RELATED WORK

Our approach differs from previous work on architectural evolution (e.g., [14,19]) 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 [18], active interfaces [10], 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 number of approaches to specify complex connectors and model them for verification. The Wright architecture description language [1], for example, uses the CSP process algebra to describe arbitrary connectors, and the Architectural Interaction Diagrams (AIDs) of Ray and Cleaveland [16] 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 [2,15]. 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, 6, 9, 22]) 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

Choosing appropriate interaction semantics for the connectors in a software architecture is often very difficult. In this paper, we present an approach that allows designers to easily experiment with alternative design choices of interaction semantics and to use design-time verification to evaluate their decisions based on the correctness of the overall system design. With our approach, components can interact with each other through different connectors using only a small set of standard interfaces. Because the interfaces usually do not need to change when changes are made to the connectors, the impact of such changes on the components is minimized. Our approach also provides a library of predefined building blocks to support the construction of a wide variety of different types of connectors. This plug-and-play approach provides significant savings in model construction time during design-time verification. With our approach, pre-defined models can be constructed for the library of building blocks, which can then be reused in the modeling of any system that uses these building blocks. In addition, since changes in the connectors do not often require changes in the components, the component models can often be reused, reducing the modeling cost when verification needs to be re-applied.

We are currently implementing a framework that supports this approach using the AcmeStudio architecture design environment[1], developed at CMU. Our tool is going to use the same notations for components, channels and ports as in Acme [7] but with extensions to their semantics. We also plan to integrate this architecture design environment with finite-state verification, using the SPIN model checker to provide support for applying verification directly on the designs created in AcmeStudio. In addition to the implementation, we are also working on extending the current approach to support other kinds of interaction mechanisms such as publish/subscribe and remote procedure call. We are also looking for larger, more compelling case studies to evaluate our approach.

One important direction for future work is to study what kind of techniques may be applied to simplify and optimize the models created using our plug-and-play approach so that finite-state verification can be applied efficiently. As we have mentioned previously, our current models for the library of building blocks are only intended for proof of concept and may not be the most efficient. These models often have unnecessary blocking statements or redundant data structures, which may unnecessarily increase the state space of the model. As an extreme example, consider our Promela model of a FIFO queue channel. Instead of implementing explicit data structures for buffering messages in FIFO order, we could simply use the native FIFO channel in Promela to handle the ordering of the messages.

We expect optimization to be extremely important since decomposing connectors into ports and channels that are modeled as separate processes introduces additional concurrency into the model, exacerbating the state explosion that limits finite-state verification. Without effective optimizations, our approach may be restricted to only small systems. Therefore, techniques that can reduce the size of the system model will be necessary to provide effective verification support. As an example of such a technique, commonly used connectors could be recognized and specially optimized models can be used for them instead of directly composing from the building block models. Note that the techniques that are used for optimization may largely depend on the specific modeling language and verification tool that are used.

Another concern with our approach is the ability to provide meaningful counterexample traces when the verification fails. In finite-state verification, when a property violation is found, a counterexample trace is often provided which gives an example trace through the model that leads to the violation of the property. With our approach, tracing an error may require delving into the details of the models of the building blocks, which requires a low-level understanding of their semantics. It would be helpful if our approach could provide a more meaningful representation of the cause of a property violation. For example, it would be useful to indicate that a deadlock in a system may be due to the use of a message buffer that drops new messages when it is full. In this way, designers can focus on the building blocks that appear to be problematic in the system and experiment with alternative choices using our plug-and-play approach.

6. ACKNOWLEDGEMENTS

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

This material is based upon work supported by the Na-

[1]`http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html`

tional 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, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proc. 10th Intl. SPIN Workshop on Model Checking of Softw.*, volume 2648, Portland, Oregon, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] S. Wang, G. S. Avrunin, and L. A. Clarke. Architectural building blocks for plug-and-play system design. In *Proc. 9th Intl. SIGSOFT Symp. on Component-Based Software Engineering*, Västerås, Sweden, June 2006. To appear.

[22] 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.