# Plug-and-Play Architectural Design and Design-time Verification

Shangzhu Wang, George S. Avrunin, and Lori A. Clarke

Department of Computer Science
University of Massachusetts Amherst, MA 01003, USA
{shangzhu,avrunin,clarke}@cs.umass.edu

**Abstract.** In software architecture, components are intended to represent the computational units of a system and connectors are intended to represent the interactions among those units. Making decisions about the semantics of these interactions is a key part of the design process. It is often very difficult, however, to choose the appropriate interaction semantics due to the wide range of alternatives to choose from and the complexity of the system behavior affected by these choices. Techniques such as finite-state verification can be used to evaluate the impact of these design choices on the overall system behavior.

This paper proposes an approach that allows designers to experiment with alternative design choices of component interactions in a plug-and-play manner. In this approach, connectors representing specific interaction semantics are composed from a library of predefined, reusable building blocks. In addition, standard interfaces for components are defined to reduce the impact of interaction changes on the components' computations. This approach facilitates design-time verification by improving the reusability of component models and by providing reusable formal models for the connector building blocks, thereby reducing model-construction time for finite-state verification.

## 1 Introduction

One of the distinguishing features of concurrent and distributed systems is the importance of defining how sequential components interact with each other. Consequently, software architecture description languages typically separate *components* that represent computations from *connectors* that represent interactions among these components [2,21,25,27]. Connectors are considered first-class design entities since they often capture some of the most important yet subtle aspects of a system, such as non-determinism, interleavings of computations, synchronization, and so on. These are concerns that can be particularly difficult to fully comprehend in terms of their impact on the overall system behavior.

Adding to this difficulty is the wide variety of alternative choices for the interaction semantics. Choosing the appropriate interaction semantics for a connector often involves not only a choice from commonly used interaction paradigms, 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. As a result, it is often necessary to make frequent changes to the design of connectors in the course of experimenting with alternative interaction semantics. Design-time verification can be useful in helping designers evaluate their design choices. Typically, design-time verification uses finite-state verification techniques (e.g., SPIN [17], SMV [19], LTSA [22], FLAVERS [10]) to check whether certain properties of a system are satisfied. With design-time verification, designers can make sure that desirable properties of a system still hold when a connector or a component is modified. Usually several iterations involving proposing a design and then verifying that design are needed.

Although it is often necessary to make frequent changes to the connector semantics while designing a system, in practice it is often difficult and costly to make these changes. Changing the specific semantics of a connector often requires nontrivial changes to the components as well. For example, a change from an asynchronous communication to a synchronous one may require making changes to the components so that a callback can be placed to explicitly notify the sender of the receipt of messages. This intertwined semantics of components and connectors also complicates design-time verification. When using finite-state verification techniques, for instance, it is necessary to build a formal 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 the process of changing and re-verifying a design needs to be repeated frequently, the lack of reusability of the component and connector models could significantly increase the cost of design-time verification.

In this paper, we describe an approach that allows designers to experiment with alternative design choices of interaction semantics in a plug-and-play manner. This approach provides a library of pre-defined, reusable building blocks from which designers can construct connectors with a wide range of interaction semantics. A connector can be specified by composing a selected subset of the building blocks. Modifying the specification of a connector can be easily achieved by adding, removing or replacing one or more of its building blocks. To minimize the impact on components of changes to connectors, this approach also proposes a set of standard interfaces that allow components to communicate with each other through different connectors. This plug-and-play approach not only improves the reusability of the designs of components and connectors, it also provides savings in model construction time during design-time verification. Specifically, pre-defined models are 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 often do not require changes in the components, the component models can often be reused, reducing the modeling cost when verification needs to be re-applied.

Section 2 describes the plug-and-play design approach. Section 3 shows how verification can be supported. Section 4 illustrates the design and verification

of a small system using this approach. Section 5 describes related work, and Section 6 discusses the current status and future directions of our work.

## 2 The Plug-and-Play Design Approach

To support plug-and-play design, our approach currently provides two kinds of pre-defined building blocks: *ports* and *channels*. A connector is usually composed from one or more of the ports and one or more of the channels. 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 interactions as under what conditions a component should be blocked or whether a component should wait for an acknowledgement after sending a message to another component. While such semantics can be easily embedded in components' computation, with our approach those aspects of interaction semantics are captured in the ports, as part of the connectors. Consequently, changes in the interaction semantics can often be made completely in the connectors and, thus, 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. More detailed discussion about the plug-and-play design approach can be found in [35].

The rest of this section describes how this plug-and-play design approach is realized for message passing, one of the most commonly used interaction mechanisms. We first present examples of building blocks that are derived from a variety of commonly used message passing semantics. We then define standard component interfaces and show how connectors and components communicate with each other through a set of protocols.

### 2.1 Message Passing Building Blocks

Many languages, such as CSP [16], Occam [9], and Linda [7] incorporate message passing facilities. There are also message passing libraries such as MPI [28] and PVM [14]. 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 and a component that receives messages may block or continue when a requested message is not available. Other aspects of message passing semantics also vary, 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 or receiver components.

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

| | | |
|---|---|---|
| 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. |

**Fig. 1.** A set of message passing building blocks

of message passing semantics. Figure 1 gives a few examples of these message passing building blocks.

Figure 2(a) shows an example of how one may specify an asynchronous message passing communication between a pair of sender and receiver components using these building blocks. The connector is composed of an asynchronous blocking send port, a blocking receive port, and a channel that buffers one message. Through this connector, the sender component sends a message without waiting for an acknowledgement from the receiver but is blocked until the message is stored in the channel. The receiver component blocks until a message can be received. By replacing the asynchronous send port with a synchronous one from the library, the new connector in Figure 2(b) allows the sender to block not only until the message is stored in the channel but also until it has been delivered to the receiver. Similarly, channels can also be easily replaced. For example, the single-slot buffer can be replaced by a FIFO queue channel that holds up to five messages, when at most five messages need to be buffered (as shown in Figure 2(c)). Moreover, the replacement of channels can be done independently of the replacement of ports. This kind of "plug-and-play" development facilitates experimentation with alternative interaction semantics.

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 storage and delivery of messages. A send port is a mediator between a sender component and a channel. 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

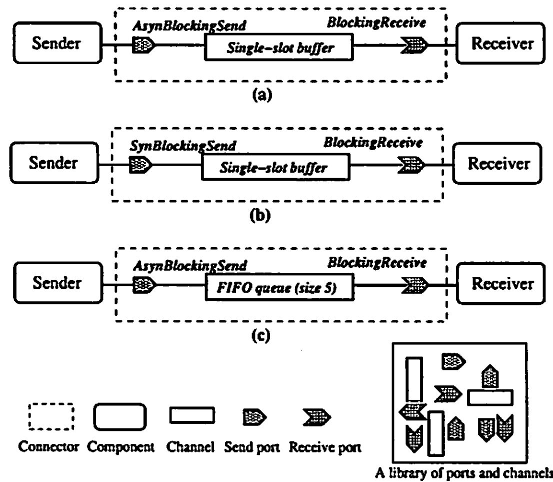


**Fig. 2.** Constructing message passing connectors

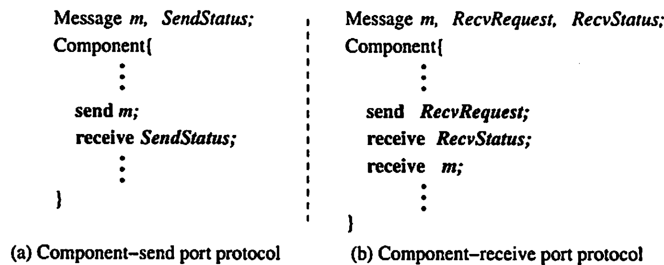


**Fig. 3.** Standard component interfaces

a message passing connector with specific semantics, one simply selects the appropriate channel to store and deliver messages, and then selects the appropriate ports from which components may send and receive messages.

## 2.2 Component Interfaces and Protocols among Building Blocks

Figure 3 shows the standard interfaces for components to send and receive messages. As shown in Figure 3(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. This difference is supported via the appropriate choice for the send port between the component and the channel.

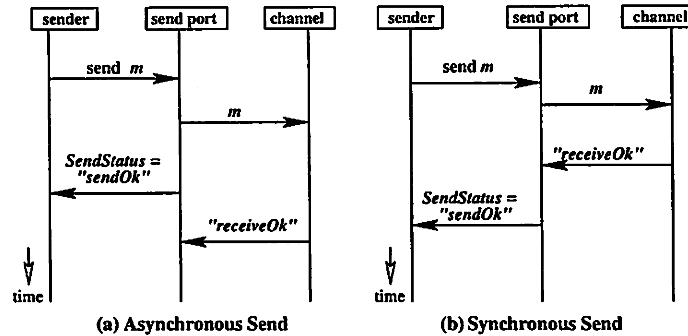


**Fig. 4.** Example scenarios of message passing interactions: asynchronous blocking send versus synchronous blocking send

Using a notation similar to Message Sequence Charts, Figure 4 illustrates how a send port controls the interleaving of the messages between the component and the channel to give different interaction semantics. Figure 4(a) shows part of a scenario for an asynchronous blocking send. In this scenario, the *SendStatus* message "sendOk" is delivered to the component immediately after the message "m" is stored in the channel but before a "receiveOk" message is received from the channel. In contrast, Figure 4(b) shows part of a scenario for a synchronous blocking send. In this scenario, the *SendStatus* message "sendOk" is not delivered to the component until after the message "m" has been stored in the channel and has been received by the receiver (indicated by a "receiveOk" message from the channel). 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.

Similarly, in Figure 3(b), a component that wishes to receive a message first sends a receive request to the port and waits for feedback (the *RecvStatus* message) about 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.

With the standard interfaces described here, 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. So if verification reveals a problem in the behavior of the system due to inappropriate interactions between components, the designer can often modify the connectors without needing to change the components. Of course, taking advantage of certain kinds of changes in the connectors may require changes in the components. For example, a designer might replace an asynchronous blocking send port by an asynchronous checking send port that informs the sender when the channel is full. But this change will only lead to more efficient execution if the sending component is designed to take advantage of the communication status information sent by the port.

This approach is not restricted to message passing. Extensions to this approach to support other kinds of interaction mechanisms, such as the publish/subscribe and RPC (remote procedure call), are described in [34].

## 3 Verification Support for the Plug-and-Play Design Approach

In addition to providing a convenient and efficient way to specify and experiment with various interaction semantics, this approach also supports design-time verification for checking specification properties of the system. For finite-state verification techniques such as model checking, formal models of the system need to be constructed before verification can be applied. In this approach, predefined and reusable formal models can be created for each building block in the library. Formal models of the selected building blocks are then composed at verification time with formal models of the components to form a system model that is then checked against the specified properties. Note that the designer is still responsible for providing the models of the components and specifying the properties.

Through verification, designers may find unexpected behaviors or errors in their system design. If the problems are caused by the interaction mechanisms, changes can be made by simply adjusting the building blocks of the connectors, perhaps without having to modify the components. When this occurs, there is no need to recreate the component models. Moreover, predefined models for the building blocks can be used in most cases to represent the modified interaction mechanisms, also reducing the cost of model construction. In this section, we describe how reusable models for the message passing building blocks can be defined, how they can be composed to form different connectors, and how the connector models are composed with component models using standard component interfaces.

For an initial evaluation of our approach, we have chosen the widely used SPIN model checker as the verifier along with its input language Promela as the modeling language. 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`, 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 if the value of the first message in the channel does not match the constant specified. There are a number of variations for the send and receive operations supported by Promela. For example, with the receive operation "??", the first matching message in the channel will be retrieved, and thus 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 architectural designs. 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 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 always refer to the native channels in Promela as *Promela channels* to distinguish them from the architecture-level channels in our approach. Here we model all the ports, channels, and components in a design as communicating processes in Promela. We use Promela channels to model the internal communications between components and ports and between ports and channels.

Figure 5 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

```
/* 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;
}
```

**Fig. 5.** Promela model for a synchronous blocking send port

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 (`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 component. As we can see from the model in Figure 5, 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

```
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
}
```

**Fig. 6.** Promela model for an asynchronous nonblocking send port

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 for the `do...od` statement described above. The send port makes sure that the signals from the channel are indeed addressed to it 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 that must match 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. If for some reason the message cannot be successfully delivered to the receiver, the channel will issue a `RECV_FAIL` signal instead of a `RECV_OK` signal. In this case, the statement `channelChan.signal?RECV_OK,eval(_pid)` will not be able to execute and the port process is blocked. Since the port cannot send a `SEND_SUCC` signal to the

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

**Fig. 7.** A sender component

component, the component is also blocked. This is consistent with the semantics of synchronous message passing where the component is blocked until the message is successfully delivered to the receiver. 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 6 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 of 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).

Figure 7 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 8 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 9 shows the Promela model for a blocking receive port. The receive port starts by waiting for a `recvRequest` mes-

```
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;
      ...
}
```

**Fig. 8.** A receiver component

```
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;
}
```

**Fig. 9.** Promela model for a blocking receive port

sage 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 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

```
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
}
```

**Fig. 10.** Promela model for a single-slot buffer channel

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 10 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.

In addition to the single-slot buffer, we have defined the Promela models for a variety of other types of channels, including one that stores and delivers messages in a FIFO order and one that handles messages based on their priorities. It is also 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. The models for these 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 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 4, 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 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 possible optimizations in Section 6.

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) [22] to verify the system designs. Somewhat different strategies may be appropriate when modeling the building blocks in a different modeling language.

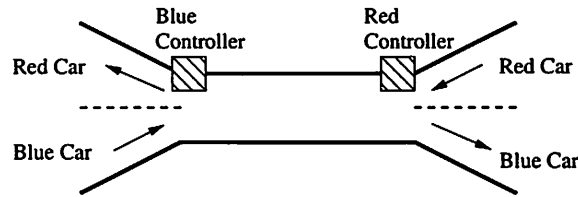


**Fig. 11.** A single-lane bridge with two controllers

## 4 The Single-lane Bridge Example

This section presents 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 this 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 11. 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 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 by one controller to the other if 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

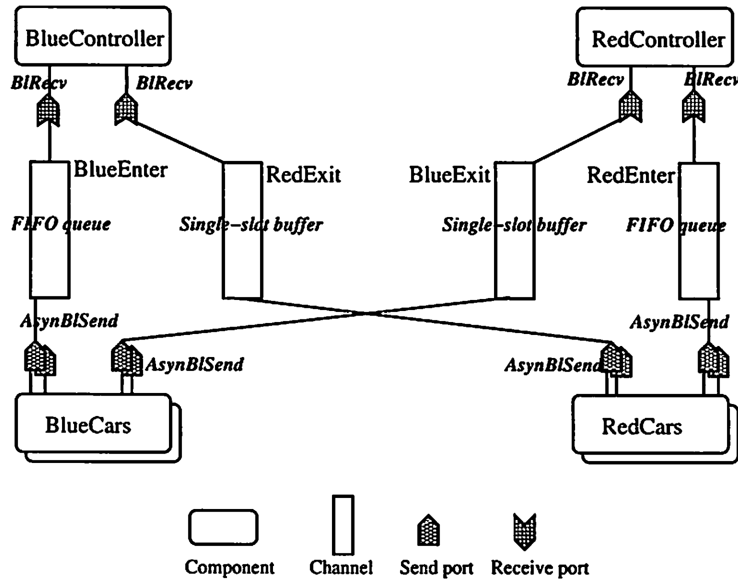


**Fig. 12.** An initial design of the "exactly-$N$-cars-per-turn" single-lane bridge example

dropped by a buffer, and so on. It is very easy to make mistakes on such matters when choosing the 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 to define each connector, incorporating the component designs provided by the designer, and then using design-time verification to make sure that the resulting system design does not violate the safety properties of the bridge.

Figure 12 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 use finite-state 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 the plug-and-play approach, the erroneous design can be easily corrected by replacing the asynchronous blocking send ports for sending enter requests with synchronous ones; no changes in the components are necessary. Verification needs to be applied again to confirm that the system now satisfies the property. With this 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 this 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 and can be simply included in the system model at verification time. In general, our approach expects designers to provide formal models for the components that employ the standard interfaces.

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 [34]. 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.

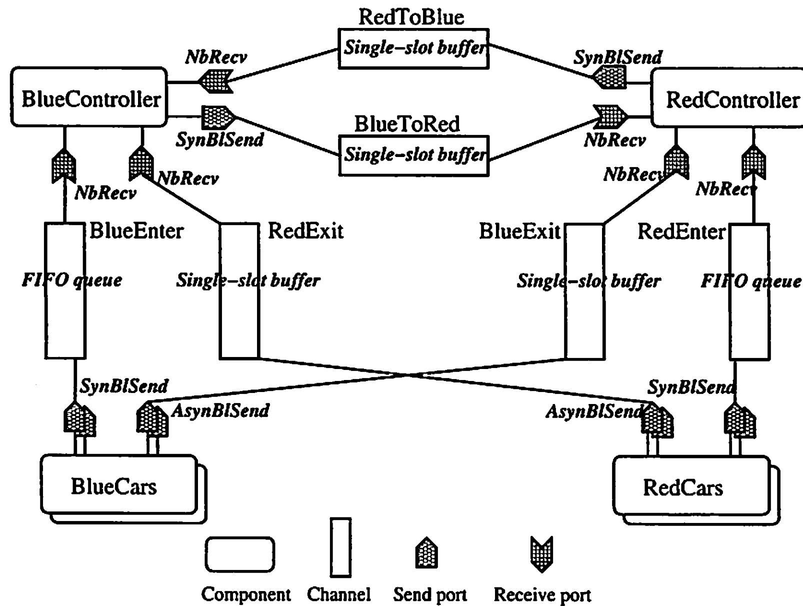


**Fig. 13.** The architecture design of the "at-most-$N$-cars-if-waiting" single-lane bridge example

Figure 13 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 poll for messages from cars and from the other controller, we must also change the connectors between the 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 the verification works in the same plug-and-play manner as the associated design approach. Having reusable models for building blocks of connectors and having the models of components stay relatively stable when only interactions are changed reduces the cost of repeated verification in an iterative design process. It therefore makes it easier and more efficient to experiment with alternative design choices for interaction semantics.

## 5 Related Work

The limitations and frustrations of component-based software development are well known (e.g., [11, 18]). Previous work, such as [2, 4, 15, 21, 25, 27], has proposed treating connectors as first-class entities in component-based development, although [15] in particular, has put the focus at a lower level of abstraction (programming level) than what we are interested in here.

There are a number of approaches that provide support for connectors and component composition. The Wright architecture description language [2], for example, uses the CSP process algebra to describe arbitrary connectors. The Architectural Interaction Diagrams (AIDs) approach [26] models connectors using process algebra. Constraint automata based approaches have also been proposed to specify and analyze the semantics of connectors composed from a set of primitive channels [3, 24]. 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 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.

Another approach to support component composition is the *mediator* approach [31, 32] which defines mediators as special components that are used to modularize how other components interact with each other in terms of their behavioral relationship. While providing a way of reasoning about the connections, the mediator approach does not support the compositional specification of connectors as our approach does.

While our approach facilitates creating connectors from existing building blocks, the connector wrapper approach [29, 30] focuses on creating new connectors by incrementally transforming existing ones. The connector wrappers can be useful in reusing connector parts and supporting easy modification to connectors. Because of its restricted internal representation of operational semantics, however, this approach can only support connectors with limited kinds of interaction semantics. In contrast, our approach allows designers to explore a wide range of interaction semantics by providing a set of building blocks that can be expanded and that are not restricted to a specific formalism.

Although a similar notion of ports has been proposed in architectural description languages such as ACME [13] and ArchJava [1], in our approach, ports are used to explicitly capture some of the most important aspects of interaction semantics such as synchronization, and therefore are treated as parts of connectors. Our definition of ports makes it possible to support standard component interfaces that allow connectors to be modified or replaced with minimal impact on the components. The term *building blocks* has been used in many different contexts. For example, in [33], building blocks are referred to as parts of soft-

ware used to build a system. The building blocks in our approach are design-level elements used to construct connectors representing interactions.

There has been extensive work on applying verification to systems employing a specific type of component interactions. Our approach, however, is intended to provide a general framework that can support many kinds of mechanisms, rather than being restricted to a single type. Specific techniques have been used to model and reason about message passing systems. For example, in [5, 23], message passing systems are specified in sets of adapted message sequence charts (MSCs), and message channels are modeled as a finite-state automaton with inputs and outputs. A procedure is then taken to construct one single automaton that accepts all linearizations of the MSCs that meet the specification of the channels. This automaton is then checked by standard techniques for emptiness to decide whether the system satisfies the specification of the channel.

Work has been done on verifying implicit invocation or publish/subscribe systems (e.g., Garlan et al. [6, 12, 36]). In this work, the semantics of publish/subscribe systems are defined along several dimensions, which is very similar to what we have done for message passing. One of the limitations of the approach is its restriction to publish/subscribe systems: the user specifies choices on the dimensions, and a formal model, suitable for finite-state verification, is automatically constructed. In order to apply that approach to other interaction mechanisms, a suitable specification formalism and a model generation tool would have to be constructed. Our approach, on the other hand allows the verification of systems with different interaction mechanisms by supporting small, reusable formal models that can be used as building blocks to construct formal models representing different kinds of interactions. One advantage of their approach, however, is that both the formal models and the verifier may be optimized and refined specifically for publish/subscribe systems.

## 6 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 this 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 pre-defined building blocks to support the construction of a wide variety of different types of connectors. This plug-and-play approach provides savings in model construction time during design-time verification. With this 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 developing a prototype tool that supports plug-and-play design and design-time verification. This tool is implemented as an extension of the ArchStudio architecture design environment [8]. With ArchStudio, designers can model, visualize and analyze system architectures. Our extension provides additional functionalities that allow designers to select specific interaction paradigms for component interactions, to specify connectors from predefined building blocks, and to use finite-state verification to evaluate the design. In addition to the prototype tool, we are also working on extending the current approach to support other kinds of interaction mechanisms such as publish/subscribe and remote procedure call. For evaluation, we are undertaking a case study to evaluate how well this approach supports the design and verification of microkernel-based embedded systems that are based on the CAmkES component model [20]. Specifically, we plan to show how the set of building blocks can represent a rich set of interaction semantics and how this approach can be useful in practice to help produce high-quality designs and implementation.

We also plan to explore several important issues that are specific to the modeling and verification aspect of our approach. One of these is the techniques that can be applied to optimize and reduce the models created using our plug-and-play approach to allow finite-state verification to be applied more 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 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 could be made available 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 usually provided to give 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 the plug-and-play approach.

## 7 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, CCR-0205575, CCF-0541035, 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.

## References

1. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proc. 26th Intl. Conf. on Softw. Eng.*, pages 187–197, Orlando, FL, USA, May 2002. ACM.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Softw. Eng. and Methodol.*, pages 140–165, 1997.
3. F. Arbab, C. Baier, J. J. M. M. Rutten, and M. Sirjani. Modeling component connectors in reo by constraint automata: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 97:25–46, 2004.
4. D. Bálek and F. Plášil. Software connectors and their role in component deployment. In *Proc. Third Intl. Working Conf. on New Developments in Distributed Applications and Interoperable Systems*, pages 69–84, Deventer, The Netherlands, 2001.
5. B. Bollig and M. Leucker. Modeling, specifying, and verifying message passing systems. In *Proceedings of the Symposium on Temporal Representation and Reasoning (TIME'01)*, pages 240–248, 2001.
6. J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *Proc. 11th ACM Symp. on Found. of Softw. Eng.*, pages 78–87, Sept. 2003.
7. Carriero, N., and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–58, Apr 1989.
8. E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Meth.*, 14(2):199–245, 2005.
9. M. Day. Occam. *SIGPLAN Notices*, 18(4):69–79, Apr 1983.

10. M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. on Softw. Eng. and Methodol.*, 13(4):359–430, 2004.
11. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proc. 17th Intl. Conf. on Softw. Eng.*, pages 179–185, Seattle, Washington, Apr. 1995.
12. 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, pages 166–180, Portland, Oregon, 2003.
13. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
14. 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.
15. T. Gensler and W. Lowe. Correct composition of distributed systems. In *Tech. of Object-Oriented Languages and Systems*, pages 296–305, 1999.
16. Hoare and C.A.R. *Communicating Sequential Processes.* Englewood Cliffs, NJ:Prentice-Hall Intl., 1985.
17. G. J. Holzmann. *The* SPIN *Model Checker.* Addison-Wesley, Boston, 2004.
18. P. Inverardi and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.
19. K.L.McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem.* Kluwer Academic, 1993.
20. I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *The Journal of Systems and Software*, 2006.
21. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. 5th European Softw. Eng. Conf.*, pages 137–153, Sitges, Spain, Sept. 1995.
22. J. Magee and J. Kramer. *Concurrency State Models and Java Programs.* John Wiley and Sons, 1999.
23. B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *Proceedings of 27th Interational Colloquium on Automata, Languages andProgramming (ICALP'2000)*, pages 487–498, Springer Verlag, 2000.
24. 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.
25. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
26. A. Ray and R. Cleaveland. Architectural interaction diagrams: AIDs for system modeling. In *Proc. 25th Intl. Conf. on Softw. Eng.*, pages 396–406, 2003.
27. M. Shaw and D. Garlan. *Softw. Architecture:Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.
28. Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.
29. B. Spitznagel and D. Garlan. A compositional approach for construct connector. In *Proc. Working IEEE/IFIP Conf. on Soft. Architecture (WICSA'01)*, pages 148–157, Royal Netherlands Academy of Arts and Sciences Amsterdam, The Netherlands, Aug. 2001.

30. B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *Proc. 2003 Intl. Conf. on Softw. Eng.*, pages 374–384, Portland, Oregon, 2003.
31. I. K. Sullivan, K.J. and D. Notkin. Evaluating the mediator method: Prism as a case study. In *IEEE Transactions on Software Engineering*, volume 22, pages 563–579, Aug. 1996.
32. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, 1992.
33. F. J. van der Linden and J. K. Müller. Creating architectures with building blocks. *IEEE Softw.*, 12(6):51–60, 1995.
34. 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.
35. S. Wang, G. S. Avrunin, and L. A. Clarke. Architectural building blocks for plug-and-play system design. In *9th International Symposium on Compoennt-Based Software Engineering*, volume 4063, pages 98–113. Springer-Verlag Lecture Notes in Computer Science, 2006.
36. H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Implementation and verification of implicit-invocation systems using source transformation. In *Proceedings of the Fifth International Workshop on Source Code Analysis and Manipulation.* IEEE Computer Society, Sept. 2005.