Architectural Building Blocks for Plug-and-Play System Design

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. One of the distinguishing features of distributed systems is the importance of the interaction mechanisms that are used to define how the sequential components interact with each other. Given the complexity of the behavior that is being described and the large design space of various alternatives, choosing appropriate interaction mechanisms is difficult. In this paper, we propose a component-based specification approach that allows designers to experiment with alternative interaction semantics. Our approach is also integrated with design-time verification to provide feedback about the correctness of the overall system design. In this approach, connectors representing specific interaction semantics are composed from reusable building blocks. Standard communication interfaces for components are defined to reduce the impact of changing interactions on components' computations. The increased reusability of both components and connectors also allows savings at model-construction time for finite-state verification.

1 Introduction

One of the distinguishing features of distributed systems is the importance of the interaction mechanisms that are used to define how the sequential components interact with each other. Consequently, software architecture description languages typically separate the computational *components* of the system from the *connectors*, which describe the interactions among those components (e.g., [1, 2, 3, 4]). Interaction mechanisms represent some of the most complex aspects of a system. It is the interaction mechanisms that primarily capture the non-determinism, interleavings, synchronization, and interprocess communication among components. These are all issues that can be particularly difficult to fully comprehend in terms of their impact on the overall system behavior.

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

Because of this complexity, design-time verification of distributed systems is particularly important. One would like to be able to propose a design, use verification to determine which important behavioral properties are not satisfied, and then modify and reevaluate the system design repeatedly until a satisfactory design is found. With component-based design, existing components are often used and glued together with connectors. In this mode of design, one would expect that the interaction mechanisms represented by the connectors would need to be reconsidered and fine-tuned several times during this design and design-time verification process, whereas the high-level design of the components would remain more stable. If using a finite-state verifier, such as SPIN [5], SMV [6], LTSA [7], or FLAVERS [8], a model of each component and each connector could be created separately and then the composite system model could be formed and used as the basis for finite-state verification.

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

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

- Defining a small set of *standard interfaces* by which components can communicate with each other through different connectors: These standard interfaces allow designers to change the semantics of interactions without having to make significant changes to the components.
- Separating connectors into *ports and channels* to represent different aspects of the semantics of connectors: This decomposition of connectors allows us to support a library of parameterizable and reusable *building blocks* that can be used to describe a variety of interaction mechanisms.
- Combining the use of standard component interfaces with reusable building blocks for connectors: This separation allows designers to explore the design space and experiment with alternative interaction semantics more easily.
- Facilitating design-time verification: With the increased reusability of components and connectors, one can expect savings in model-construction time during finite-state verification.

This paper presents the basic concepts and some preliminary results from an evaluation of our approach. Section 2 illustrates the problem we are trying to address through an example. Section 3 shows how the general approach can be applied to the message passing mechanism. In section 4, we demonstrate through examples how designers may experiment with alternative interaction semantics using our approach. Section 5 describes the related work, followed by conclusions and discussions of future work in Section 6.

2   An Illustrative Example

As an example, consider a bridge that is only wide enough to let through a single lane of traffic at a time [7]. For this example, we assume that traffic control is provided by two controllers, one at each end of the bridge. Communication between controllers as well as between cars and controllers may be necessary to allow appropriate traffic control. To make the discussion easier to follow, we refer to cars entering the bridge from one end as the blue cars and 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. We start with a simple "exactly-N-cars-per-turn" version of this example, where the controllers take turns allowing some fixed number of cars from their side to enter the bridge. Note that since each controller counts the fixed number of cars entering and exiting the bridge, no communication is needed between the two controllers.

For an architectural design of this simple version of the system, one needs to identify the components and the appropriate interactions among the components. It is natural to propose a system composed of a *BlueController* component, a *RedController* component, and one or more *BlueCar* components and *RedCar* components. In such a distributed system, message passing seems to be a natural choice for the component interactions. Four connectors then need to be included to handle message passing among the components as indicated in Figure 1: a *BlueEnter* connector between the *BlueCar* components and the *BlueController* component, a *BlueExit* connector between the *BlueCar* components and the *RedController* component, and similarly a *RedEnter* connector and a *RedExit* connector.

As described in Figure 1(a), a car sends an *enter_request* message to the controller at the end of the bridge it wants to enter and then proceeds onto the bridge. When it exits the bridge, it notifies the controller at the exit end by sending an *exit_request* message. Controllers receive *enter_request* and *exit_request* messages, update their counters, and decide when to switch turns. Since there may be multiple cars that communicate with each controller, messages are buffered in the connectors between car components and controller components.

Astute readers will notice that according to the description in Figure 1(a), cars from different directions can be on the bridge at the same time, which could cause a crash. This is due to an erroneous design in the component interactions. With this design, a car sends an *enter_request* message and immediately goes onto the bridge without confirming that its request has been accepted by the controller. This controller, however, may still be waiting for exit requests from cars from the other direction, and the enter request message from this car may still be in the buffer, waiting to be retrieved and handled. Therefore, a car may enter the bridge while there are still cars traveling in the opposite direction. Obviously, what is needed here is synchronous communication between a car and its controller rather than asynchronous communication.

One way to fix this problem is to have the controller send a *go_ahead* message after receiving each enter request to authorize that car to enter the bridge. After
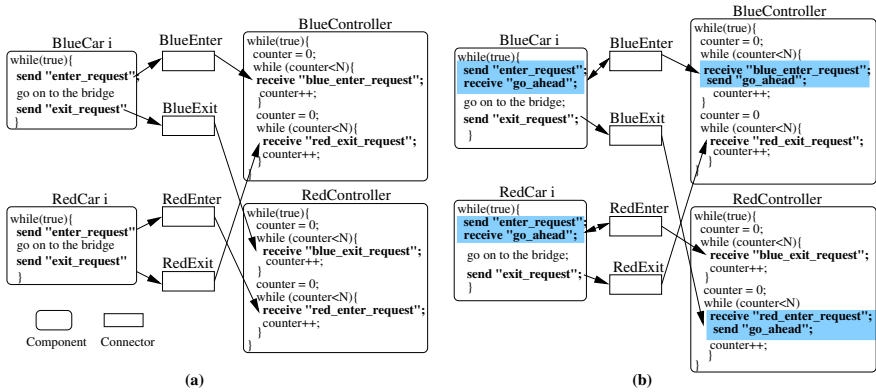
Fig. 1. Architecture design and illustration of component interactions for the single-lane bridge example

sending the enter request, the car would wait for this acknowledgement before entering the bridge, as shown in Figure 1(b) (the highlighted areas indicate the changes). These changes, involving both the car components and the controller components, effectively make the communication between them synchronous and solve the problem caused by the asynchronous communication.

This example shows the typical design practice in which the semantics of the interactions are not specified independently, but instead are spread among the connectors and the components. This is a trivial example, but it is easy to envision how the intertwined semantics of the connectors and components increases the challenge of discovering and correcting errors in the design of more complex systems. Therefore, we prefer an approach that allows us to modify connectors and components more independently of each other.

3   Plug and Play with Message Passing

As illustrated in the example above, changing from asynchronous message passing to synchronous message passing requires changes in the components, not just in the connectors. In practice, designers must consider a wide range of alternative semantics when selecting the appropriate interaction mechanism for a connector. If it is subsequently discovered, perhaps through verification, that the selected interaction mechanism is wrong, then it is likely that, not only the connector, but the associated components will need to be modified and then reevaluated. Therefore, the impact of changes in connectors on components will not only make it more challenging for designers to find a suitable design, but will also affect the maintainability and reusability of the system components. Our approach tries to address these problems by decomposing connectors into ports and channels, by representing the semantic variations for both ports and channels as building blocks that can be assembled to provide the desired interaction mechanism, and by designing these building blocks so that components

can communicate through standard interfaces that are designed to work with
any kinds of connectors.

In this section, we show how our plug-and-play approach can be realized for
message passing, one of the most commonly used interaction mechanisms for dis-
tributed systems. 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 com-
municate with each other through a set of protocols. We also discuss how finite-
state verification can be employed to facilitate the plug-and-play style of design.
Finally, we mention that this approach is not restricted to message passing, but
can be applied to many of the most common interaction mechanisms. In partic-
ular, we discuss briefly how this can be accomplished for the publish/subscribe
interaction mechanism.

3.1   Message Passing Variations and Building Blocks

Many languages such as CSP [9] and Linda [10] incorporate message passing
facilities. There are also message passing libraries such as MPI [11]. Although the
fundamentals of message passing interactions are sending and receiving messages,
there are a surprising number of semantic variations for these two operations, as
well as variations in the communication media used to store and deliver messages.

For example, a synchronous send operation will block the sender until the
message is delivered to the recipient, while other variations would allow the
sender to continue execution immediately or as soon as the message is stored
in the buffer. Similarly, a receiver component may be blocked or may return
immediately when a desired message cannot be retrieved from the buffer at the
moment. A receive may also allow messages to be selectively retrieved from
the buffer based on a matching criteria. Other variations of message passing
semantics involve the message buffers, such as the size of the buffer and the
ordering of messages been stored and delivered.

With such variations, determining a particular kind of message passing inter-
action for a system essentially means selecting a combination of these semantics.
As we have demonstrated in the previous sections, this large design space may
make it difficult for designers to choose the correct and desirable semantics. Our
approach helps designers with such choices by creating building blocks that cap-
ture the different combinations of the variations for each aspect of the message
passing semantics, and therefore allowing designers to experiment with the vari-
ations by plugging and playing with these building blocks. Our building blocks
include different kinds of send ports, receive ports, and channels that together
cover a number of variations for the most commonly used message passing se-
mantics. A small sample of the message passing building blocks, selected to
include those used in our examples, is given in Figure 2.

Figure 3(a) shows an example of how one may specify an asynchronous mes-
sage passing communication between a pair of sender and receiver components.
The connector is composed of an asynchronous blocking send port, a blocking
receive port, and a channel that buffers one message. Through this connector,

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 "asynchronos 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**	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	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. 2. A set of message passing building blocks

the sender component sends a message without waiting for an acknowledgement from the receiver but blocks until the message is stored in the channel. The receiver component blocks until a message can be received. By replacing the asynchronous send port with a synchronous one from the library, the new connector in Figure 3(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 5 messages, when messages need to be buffered (as shown in Figure 3(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. We have also found that our approach helps reduce the effort needed for repeated model construction when designers use design-time finite-state verification to check their design choices.

3.2   Component Interfaces and Protocols Among Building Blocks

In this section , we describe the standard component interfaces for sending and receiving messages and the protocols used between these interfaces and different kinds of connectors. The component interfaces are used as follows: A sender component first issues a send command and then waits to receive a *SendStatus* message from the connector; similarly, a receiver component first sends a receive request to the port, waits for a *RecvStatus* message, followed by another message from the connector that may contain the requested data. These interfaces are designed to work with connectors having different send and receive semantics. For example, in the case of asynchronous message passing, the connector returns the *SendStatus* message to the sending component immediately, while for synchronous message passing, the connector returns the *SendStatus* until after the sender's message has been delivered. The *RecvStatus* message indicates whether the requested message has been successfully retrieved, that is, whether the subsequent message contains the real data. Different connectors may send these
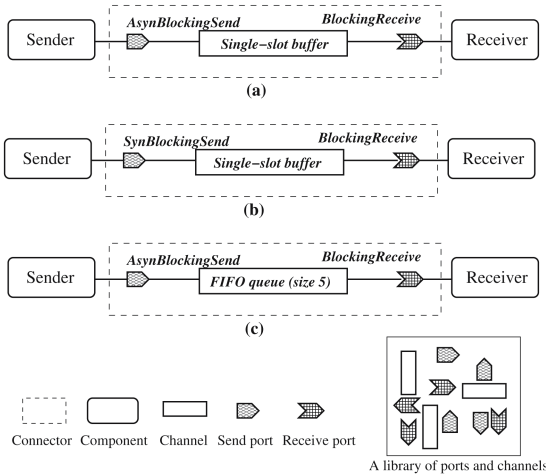
Fig. 3. Constructing message passing connectors

messages at different stages of retrieving a message. Moreover, always sending a message after the *RecvStatus* allows this interface to work with nonblocking receives that allow failure of retrieving messages.

   To see how different connectors may interact with these interfaces, one has to first understand the important role of ports in supporting the kind of plug-and-play design we propose. In our approach, connectors are decomposed into channels that represent the communication media (in this case the message buffers), and ports that capture the synchronization semantics of the communication. This separation frees components from being tied to any specific synchronization semantics and therefore allows easy manipulation of all aspects of interaction semantics. It is the ports that handle the interleavings of communications between components and channels and deciding when a specific status or data message should be forwarded, hiding all the details from both components and channels.

   Using a notation similar to Message Sequence Charts, Figure 4 and 5 show the typical protocols used between components, ports and channels for sending and receiving messages. In Figure 4, we see that for both asynchronous send and synchronous send, the same set of protocols are used between the sender component and the send port, and between the send port and the channel. It is the send port that controls the relaying and interleaving of the internal events, and thus whether the message passing is synchronous or asynchronous. In Figure 4(a), the asynchronous send port returns the *sendOk* message to the sending component without waiting for the channel to deliver the message and simply discards the *receiveOk* message from the channel when it arrives. The synchronous send port in Figure 4(b) waits to receive the *receiveOk* message from the channel before sending *sendOk* to the sending component, which is therefore blocked until after the message m is received. Neither the sending component nor the channel needs to know whether the connector is implementing synchronous or
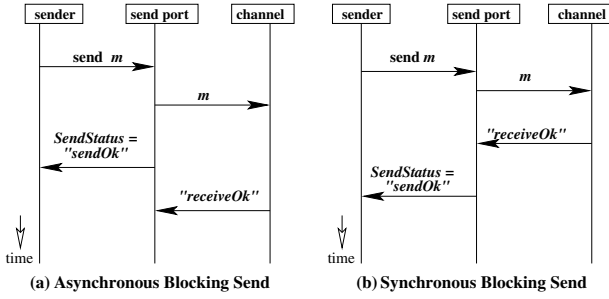
Fig. 4. Example scenarios of message passing interactions (using send ports)

asynchronous message passing; the designer can swap one send port for another to switch the semantics of the connector.

Similarly, Figure 5 shows that same protocols can be used for both blocking receive and nonblocking receive. In Figure 5(a), after forwarding the *ReceiveRequest* from the receiver to the channel, the port blocks until an *outOk* message is received from the channel indicating that the desired message is available. A *recvSucc* confirmation is then sent to the receiver following the retrieved message. To implement the semantics of nonblocking receive (Figure 5(b)), a receive port may immediately return when the desired message is not available (*outFail*) by sending a *recvFail* message followed by an empty message to the receiving component. In a fashion similar to that illustrated above, we are able to support the plug-and-play of a number of different send and receive ports as well as channels defined in Figure 2.

3.3   Design-Time Verification

In addition to providing a convenient and efficient way of specifying and experimenting with various interaction semantics, we also support 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. For the purpose of our approach, predefined and reusable formal models can be created for each building block in our 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 properties specified. Note that the designer is 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.
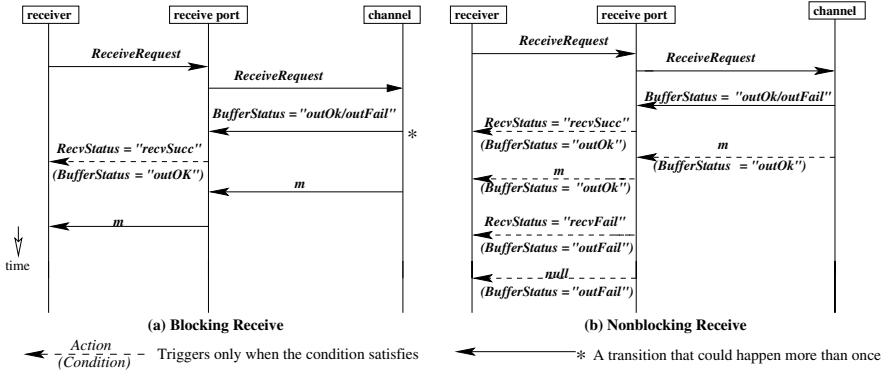
Fig. 5. Example scenarios of message passing interactions (using receive ports)

To evaluate our approach, we have used SPIN [5] to verify a series of designs using our building blocks. In our evaluation, the formal models of components and building blocks are described in Promela, the input language of SPIN. We use the default message passing operations ("?" and "!") in Promela to implement the communications among components, ports and channels. Each port is a Promela *proctype* that takes two Promela native channels as parameters for communications with the component and the channel that are connected to this port. For the purpose of the evaluation, we have coded models in a way that reflects our goal of reusable and parameterizable building blocks. For a particular choice of interaction mechanisms, it might well be possible to implement connectors more directly using features of the Promela language. The full description of the Promela models for the building blocks is given in [12].

Notice that by using SPIN and Promela to support design-time verification, we are showing only one possible way to combine our design approach and verification. Our approach is not tied to particular formalisms or verification techniques. In fact, we have defined the same set of building blocks in the process algebra FSP and used LTSA [7] to verify the system designs. It is reasonable to expect, however, that when using different formalisms and verification techniques, specialized optimizations will need to be developed.

3.4   Other Interaction Mechanisms

Although here we have described this approach for message passing interactions, we believe that the overall approach can be applied to most commonly used interaction mechanisms. To validate this claim, we have also applied this approach to publish/subscribe interactions, another commonly used interaction mechanism. In publish/subscribe systems, the fundamental communications between components and connectors are the announcement of events by components, the delivery of events to components, and the subscription or unsubscription by which components indicate their interest in particular events. It is straightforward to map these communications to sending and receiving messages; therefore they
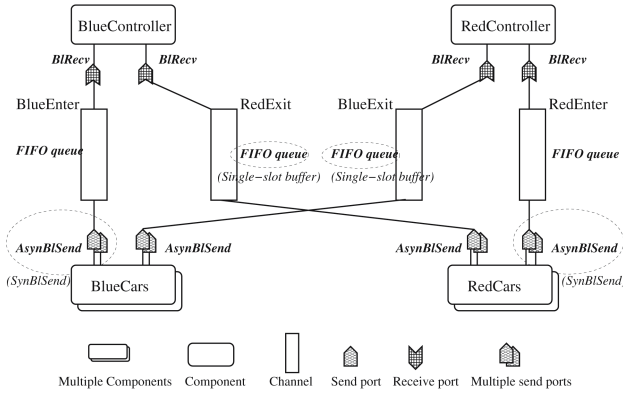
Fig. 6. An initial design of the "exactly-*N*-cars-per-turn" single-lane bridge

can be described using available message passing building blocks. In message passing, it is almost always the case that the sender initiates the communication by pushing messages to the connector and the receiver pulls messages from the connector. Unlike message passing, however, most publish/subscribe systems support one or more combinations of *push/pull* on both the publisher side and the subscriber side. To describe these semantics, new kinds of send and receive ports that capture such *push/pull* semantics are defined. A more detailed discussion about the building blocks for publish/subscribe can be found in in [12].

4   The Single-Lane Bridge Example Revisited

We now return to the single-lane bridge example introduced in Section 2 to illustrate how the techniques described above facilitates iterative exploration and verification of designs. Figure 6 shows an architecture design of the exactly-*N*-cars-per-turn version of the system. All the cars from the same direction (indicated as having the same color) communicate with the controller at each side through a single connector. For the initial design, asynchronous message passing is chosen for both the communication between a car and the controller on its entering side and the communication between the car and the controller on the other side. FIFO queues are selected for buffering messages.

One important property of the system that we want to check is that cars traveling in opposite directions can never be on the bridge at the same time. By composing the Promela models of the components provided by the designer and the prebuilt models of the building blocks from the library, we can use SPIN to determine whether the system satisfies the property. In this case, of course, SPIN produces a counterexample in which a blue car sends an *enter_request* message and enters the bridge, followed by a red car sending an *enter_request* message and entering the bridge. As noted above, the problem is obviously the result of the careless design of the asynchronous communication between cars and the controller handling enter requests, which allows cars to enter the bridge before their

enter requests have even been received by the controller. 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. To confirm that the system now satisfies the property, the verification can be repeated with the formal models of the asynchronous ports replaced by those of the synchronous ones.

In fact, astute readers may notice that the FIFO queues used for buffering *exit_request* messages are not necessary since the exact ordering in which the *exit_request* messages are received does not matter. Therefore, the FIFO queue channels used in *BlueExit* and *RedExit* connectors can be safely replaced with single-slot buffers. This modification again requires no further changes in other parts of the architecture. Similarly, the modified design can be re-verified as before to make sure the system still satisfies the property.

Of course, not all modifications to a system require only simple changes in the interaction mechanisms. Suppose that, in order to improve traffic flow, the designer wishes to modify the bridge system so that when there are fewer than N cars crossing the bridge from one side, the turn can be yielded without waiting for N cars to cross, allowing cars from the other side to enter the bridge. To change the previous design of the single-lane bridge into this "at-most-N-cars-if-waiting" version, additional communication between the controllers needs to be added. Although this functional change of the system unavoidably requires changes in the controller components, we can see that with our approach, we can reduce the impact of these changes on both the design and the verification.

Figure 7 shows a possible architecture for the modified system, with two new connectors between the controllers to allow the communication of the current traffic status at each end. The interactions between two controllers are represented in a synchronous message passing connector composed of a synchronous blocking send port, a nonblocking receive port, and a reliable single-slot buffer. Since the controllers now have to actively poll enter request messages from cars to check if there is any car waiting to enter the bridge, we also need to change the blocking receive ports used by the controllers in the previous design into nonblocking ones. To verify that this new system still prevents crashes 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 models of the building blocks in the library.

A third and more realistic variation of the single-lane bridge example might involve traffic control of emergency vehicles. Although this again cannot avoid functional changes in the components, the necessary changes in the interaction mechanisms would not affect the components and can be made easily. For example, the FIFO queues used for buffering enter request messages may be replaced with priority queues to handle emergency requests. The new design can be verified again in the same manner as described above. The detailed design and formal models of the three versions of the example are described in [12].

Through this example, we illustrate how our plug-and-play approach, integrated with design-time verification, may assist the designer exploring a series
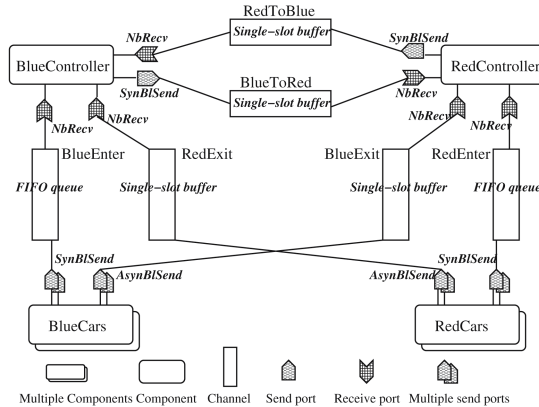
Fig. 7. The architecture design of the "at-most-N-cars-if-waiting" single-lane bridge

of system designs. With our approach, the impact of each change can be kept relatively local, in that components only need to be modified when they must handle new functionality. Changes in a connector can be made relatively easily by selecting alternative building blocks to define that connector.

5 Related Work

The limitations and frustrations of component-based development are well known (e.g., [13, 14]). Previous work, such as [15, 1, 2, 3, 4, 16], has proposed treating connectors as first-class entities in component-based development, although [16] in particular, has put the focus at a lower level of abstraction (programming level) than what we are interested in.

The idea of specifying complex connectors and modeling them for verification is, of course, not new. The Wright architecture description language [1], for example, uses the CSP process algebra to describe arbitrary connectors. The Architectural Interaction Diagrams (AIDs) of Ray and Cleaveland [17] 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 [18, 19]. In approaches like these, the burden is on the designer to construct connectors with the right semantics from powerful, but low-level, primitives. Our approach is aimed more at providing a library of building blocks from which connectors representing widely used interaction mechanisms can be easily constructed, offering "ready-to-use" pieces that hide from the user most of the details of how these pieces are actually constructed and modeled. The interaction mechanisms we describe are at a lower level of abstraction than the communication patterns described in [20]. Our approach defines finer-grained patterns that express specific semantics of interactions, and provide a mechanism that allows the designer to work with the detailed semantics.

Although a similar notion of ports has been proposed in architectural description languages such as ACME [21] and ArchJava [22], 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 mechanism we use to realize this is closely related to the connector wrappers of [23], although their emphasis is on adapting existing connectors whereas ours is on building up new connectors that can be easily exchanged for one another. The term *building blocks* has been often used in different contexts. For example, in [24], building blocks are referred to as parts of software used to build a system. The building blocks in our approach are design-level elements used to construct connectors representing interactions.

Our work on the semantics of interaction mechanisms is related to the work on categorizing connectors (e.g. [25,26]). In particular, our analysis of the variations of message passing semantics is similar in spirit to the analysis of publish/subscribe systems in [27]. There has been extensive work on applying verification to systems employing a single type of interaction mechanism (e.g. [28, 29, 30]). Our approach is intended to support many kinds of mechanisms, rather than being restricted to a single type.

A number of middleware frameworks support component-based development, although each typically allows a somewhat limited range of interaction mechanisms and provides no direct support for verification. Some work, such as the Cadena system [31], has been directed at providing verification support for systems built on standard middleware. There is also work on the verification of middleware-based software architecture [32]. A number of tools and approaches have also been proposed for assembling existing components into applications, including mediators [33], Piccola [34], and various techniques for wrapping components. Our interest here is more in the choice of interaction mechanisms between components and less on the adaptation of existing components to interact with each other. Our approach also differs from previous work on architectural evolution (e.g., [35, 36]) in our focus on supporting the exploration of different interaction mechanisms at the design stage and our emphasis on modeling and verification.

6   Conclusion and Future Work

In this paper, we propose a compositional specification approach that helps designers more easily experiment with different interaction mechanisms between components. By decomposing the connectors into ports and channels, and using ports as mediators between components and channels, we are able to keep the interface of the components simple and standardized so that changes to the interaction mechanisms can be made with little or no modification to the components. The decomposition also allows us to build a library of ports and channels as reusable building blocks to construct connectors with different semantics. Our

approach is also integrated with finite-state verification techniques, facilitating design-time verification and the early detection of design errors. Using our approach, designers may experiment with their choice of design for a variety of interaction semantics by simply plugging in, or replacing, building blocks and then using verification to check their design choices. Since this design process may be repeated to reflect system changes, our approach allows considerable reuse of the models of components and connectors. Consequently, we also save on model-construction time while doing the finite-state verification.

We are currently implementing our approach by developing plugins to the architecture design environment AcmeStudio[1] developed at CMU. Our prototype tool will allow designers to define and use building blocks to specify component interactions. It will also allow the specification of component models and the use of a model checker to verify the design. We are also carrying more case studies to demonstrate and further evaluate our approach.

We intend to explore other commonly used interaction mechanisms and, when necessary, to construct additional building blocks to express their semantics. There are a number of interesting issues related to design-time verification. For instance, optimizations could be developed to reduce the system models that are composed from the building blocks and models of the components; these depend, of course, on the particular modeling formalism and verification tools being applied. We need to explore these optimizations and learn when they can be profitably applied.

Acknowledgements

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. on Softw. Eng. and Methodol. (1997) 140–165
2. Shaw, M., Garlan, D.: Softw. Architecture:Perspectives on an Emerging Discipline. Prentice-Hall (1996)
3. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Proc. 5th European Softw. Eng. Conf., Sitges, Spain (1995) 137–153

---

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

4. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIG-SOFT Softw. Eng. Notes **17**(4) (1992) 40–52
5. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
6. K.L.McMillan: Symbolic Model Checking: An approach to the State Explosion Problem. Kluwer Academic (1993)
7. Magee, J., Kramer, J.: Concurrency State Models and Java Programs. John Wiley and Sons (1999)
8. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow analysis for verifying properties of concurrent software systems. ACM Trans. on Softw. Eng. and Methodol. **13**(4) (2004) 359–430
9. Hoare, C.A.R.: Communicating Sequential Processes. Englewood Cliffs, NJ:Prentice-Hall Intl. (1985)
10. Carriero, N., Gelernter, D.: Linda in context. Comm. ACM **32**(4) (1989) 444–58
11. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press (1996)
12. Wang, S., Avrunin, G.S., Clarke, L.A.: Architectural building blocks for plug-and-play system design. Technical Report UM-CS-2005-16, Dept. of Comp. Sci., Univ. of Massachusetts Amherst (2005)
13. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch, or, why it's hard to build systems out of existing parts. In: Proc. 17th Intl. Conf. on Softw. Eng., Seattle, Washington (1995) 179–185
14. Inverardi, P., Wolf, A.L.: Uncovering architectural mismatch in component behavior. Science of Computer Programming **33**(2) (1999) 101–131
15. Bálek, D., Plášil, F.: Software connectors and their role in component deployment. In: Proc. Third Intl. Working Conf. on New Developments in Distributed Applications and Interoperable Systems, Deventer, The Netherlands (2001) 69–84
16. Gensler, T., Lowe, W.: Correct composition of distributed systems. In: Tech. of Object-Oriented Languages and Systems. (1999)
17. Ray, A., Cleaveland, R.: Architectural interaction diagrams: AIDs for system modeling. In: Proc. 25th Intl. Conf. on Softw. Eng. (2003) 396–406
18. Arbab, F., Baier, C., Rutten, J.J.M.M., Sirjani, M.: Modeling component connectors in reo by constraint automata: (extended abstract). Electr. Notes Theor. Comput. Sci. **97** (2004) 25–46
19. Mehta, N.R., Medvidovic, N., Sirjani, M., Arbab, F.: Modeling behavior in compositions of software architectural primitives. In: 19th IEEE Intl. Conf. on Automated Softw. Eng. (2004) 371–374
20. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented software architecture: a system of patterns. John Wiley & Sons, Inc., New York, NY, USA (1996)
21. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press (2000) 47–68
22. Aldrich, J., Chambers, C., Notkin, D.: Archjava: Connecting software architecture to implementation. In: Proc. 26th Intl. Conf. on Softw. Eng., Orlando, FL, USA, ACM (2002)
23. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: Proc. 2003 Intl. Conf. on Softw. Eng., Portland, Oregon (2003)
24. van der Linden, F.J., Mller, J.K.: Creating architectures with building blocks. IEEE Softw. **12**(6) (1995) 51–60

25. Hirsch, D., Uchitel, S., Yankelevich, D.: Towards a periodic table of connectors. In: Proc. Third Intl. Conf. on Coordination Languages and Models, London, UK (1999) 418
26. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proc. 22nd Intl. Conf. on Softw. Eng., Limerick, Ireland (2000) 178–187
27. Garlan, D., Khersonsky, S., Kim, J.S.: Model checking publish-subscribe systems. In: Proc. 10th Intl. SPIN Workshop on Model Checking of Softw., Portland, Oregon (2003)
28. Bradbury, J.S., Dingel, J.: Evaluating and improving the automatic analysis of implicit invocation systems. In: Proc. 11th ACM Symp. on Found. of Softw. Eng., Finland (2003)
29. Zanolin, L., Ghezzi, C., Baresi, L.: An approach to model and validate publish/subscribe architectures. In: Proc. Specification and Verification of Component-Based Systems, Helsinki, Finland (2003) 35–41
30. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: Proc. 9th European Softw. Eng. Conf. / 11th ACM SIGSOFT Intl. Symp. on Found. of Softw. Eng., Helsinki, Finland (2003) 257–266
31. Childs, A., Greenwald, J., Ranganath, V.P., Deng, X., Dwyer, M.B., Hatcliff, J., Jung, G., Shanti, P., Singh, G.: Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems. In: Proc. of Fund. Approaches to Softw. Eng., 7th Intl. Conf. (2004) 160–164
32. Caporuscio, M., Inverardi, P., Pelliccione, P.: Compositional verification of middleware-based software architecture descriptions. In: Proc. 26th Intl. Conf. on Softw. Eng., Washington, DC, USA, IEEE Computer Society (2004) 221–230
33. Sullivan, K.J., Notkin, D.: Reconciling environment integration and software evolution. ACM Trans. Softw. Eng. Methodol. **1**(3) (1992) 229–268
34. Achermann, F., Lumpe, M., Schneider, J.G., Nierstrasz, O.: Piccola – a small composition language. In Bowman, H., Derrick, J., eds.: Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches. Cambridge University Press (2001) 403–426
35. Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A language and environment for architecture-based software development and evolution. In: Proc. 21st Intl. Conf. on Soft. Eng., Los Angeles (1999) 44–53
36. van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N.: Taming architectural evolution. In Inverardi, P., ed.: Proc. 8th European Softw. Eng. Conf./9th Symp. on the Found. of Softw. Eng., Vienna (2001) 1–10