Concurrent Object-Oriented Development with Behavioral Design Patterns

Benjamin Morandi[1], Scott West[1], Sebastian Nanz[1], and Hassan Gomaa[2]

[1] ETH Zurich, Switzerland [2] George Mason University, USA
firstname.lastname@inf.ethz.ch hgomaa@gmu.edu

Abstract. Architectural modeling using the Unified Modeling Language (UML) can support the development of concurrent applications, but the problem of mapping the model to a concurrent implementation remains. This paper defines a scheme to map concurrent UML designs to a concurrent object-oriented program. Using the COMET method for the architectural design of concurrent object-oriented systems, each component and connector is annotated with a stereotype indicating its behavioral design pattern. For each of these patterns, a reference implementation is provided using SCOOP, a concurrent object-oriented programming model. Given the strong execution guarantees of the SCOOP model, which is free of data races by construction, this development method eliminates a source of intricate concurrent programming errors.

1 Introduction

Writing concurrent applications is challenging because of the complexity of concurrent software architectures and the hazards associated with concurrent programming, such as data races. For object-oriented applications, support for the architectural design of concurrent software is fortunately available. Standard notations, such as the Unified Modeling Language (UML), can provide such support when used with a method for developing concurrent applications, such as COMET [1]. The remaining difficulty is the mapping of the concurrent object-oriented model to an implementation that avoids common concurrency pitfalls.

This paper describes a development method that starts with a concurrent UML design, annotated with behavioral stereotypes, and maps the design systematically to an implementation that is guaranteed to be data-race free. Each component and connector in the UML model is given a behavioral role, based on COMET. For each of COMET's component and connector types, this paper defines a mapping to an implementation in SCOOP (Simple Concurrent Object-Oriented Programming) [2,3], a concurrent object-oriented programming model. Choosing this model over others simplifies concurrent reasoning [4] and offers strong execution guarantees: by construction, the model is free of data races [2]; also, a mechanism for deadlock avoidance is available [5]. To evaluate the approach, the development process is applied to a case study of an ATM system that covers all important connector and component patterns.

A companion technical report [6] contains additional material. The remainder of the paper is structured as follows. Section 2 describes behavioral design

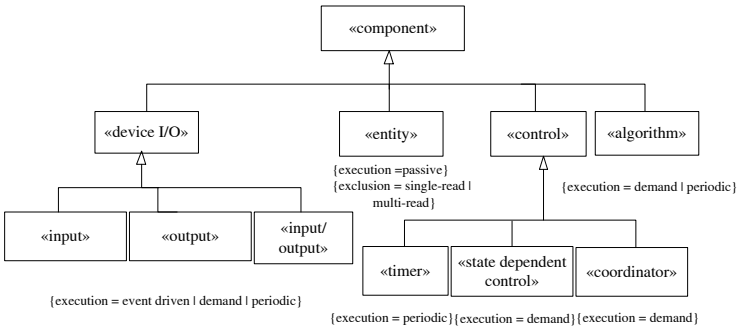


Fig. 1: Classification of components using stereotypes

patterns of the COMET method in UML. The implementation of the design patterns is described in Section 3. Section 4 presents the case study. Section 5 presents a survey of related work, and Section 6 draws conclusions.

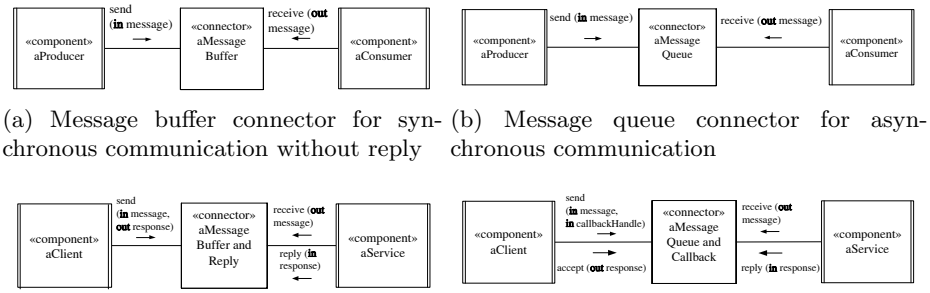
2 Behavioral design patterns

Behavioral design patterns used by the COMET method [1, 7] address design issues in concurrent and distributed applications. There are two main categories: component patterns and connector patterns.

Component patterns. Component patterns address concurrent component design. Each component is depicted from two different perspectives, its role in the application and the behavioral nature of its concurrency. Models of the design use UML stereotypes to depict the decisions made by the designer. The stereotype depicts the component's role criterion, which describes the component's function in the application such as «I/O» or «control». A UML constraint is used to describe the type of concurrency of the component, which is based on how the component is activated. For example, a concurrent «I/O» component could be activated by an external event or a periodic event, whereas an «entity» component is passive and access to it is mutually exclusive or by means of multiple readers and writers. Components are categorized using the component stereotype classification hierarchy in Figure 1. Separate stereotypes can be used to depict the component role and the type of concurrency.

Connector patterns. Connector patterns describe the different types of message communication between the concurrent components. In both distributed and non-distributed applications, connector patterns include asynchronous communication and synchronous communication with or without reply.

A connector can be designed for each connector pattern to encapsulate the details of the communication mechanism. The *message buffer* and *message buffer*



(a) Message buffer connector for synchronous communication without reply

(b) Message queue connector for asynchronous communication

(c) Message buffer and reply connector for synchronous communication with reply

(d) Message queue and callback connector for asynchronous communication with callback

Fig. 2: Connectors for communication patterns

and reply connectors respectively implement the synchronous communication pattern without reply and with reply; the *message queue* and *message queue and callback* connectors implement the corresponding asynchronous communication patterns. These connectors can also be categorized using stereotypes.

Figure 2a depicts a synchronous communication without reply pattern, in which the concurrent producer component sends a message to a concurrent consumer component via a message buffer connector, and waits for the consumer to accept the message. Figure 2b depicts an asynchronous message communication pattern in which a producer communicates with a consumer through a message queue connector that encapsulates the details of the asynchronous communication by: (1) adding a message from the producer to a FIFO message queue and only suspending the producer if the queue is full (2) returning a message to a consumer or suspending the consumer if the queue is empty. Figure 2c depicts a synchronous communication with reply pattern in which the client component sends a message to a service component and waits for the reply via a message buffer and reply connector. Figure 2d depicts an asynchronous communication with reply pattern, in which the client sends a message to a service via a message queue and callback connector, continues executing and later receives the service response from the connector. In this pattern, the client needs to provide an id or callback handle to which the response is returned.

3 Implementation of design patterns

This section describes the SCOOP implementation of the behavioral design patterns with examples, and highlights the most relevant implementation properties. The full implementation is available online [8].

Implementing components. Components are implemented by providing a class hierarchy mirroring the component taxonomy in Figure 1. Specialized components in the end user application inherit from the appropriate abstract class.

To remove ambiguity, the term *component object* will be used to denote an instance of the *component class*, which is the implementation of the design pattern component.

We examine the implementation of the periodic task in detail. It is implemented as a pair of classes: one class represents the job to be done, the other represents a "pacemaker", which periodically calls an instance of the first class to perform its task. The instances of each class reside on two distinct processors.

The basic interface to `PERIODIC` defines:

- a single iteration (`step`),
- an indicator that the task is finished (`is_done`),
- integration with the pacemaker: `notify` executes a step then asks the pacemaker to schedule another call to `notify` (unless `is_done`).

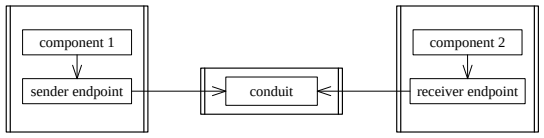
This design increases the availability of the `PERIODIC` object to other processors. If the waiting were to occur directly within the `PERIODIC` object, that object's processor would be unavailable for the duration of the waiting time; other objects would be unable to ask the periodic task simple queries such as `is_done`. This is why the pacemaker does the waiting and calls to the task after an appropriate delay. The interaction between the pacemaker and the periodic task allows the processor containing the periodic task to remain unoccupied between `step` executions.

Implementing connectors. Each of the connectors is implemented using three dependent pieces: the sender endpoint, the receiver endpoint, and the conduit(s). These are implemented as a cohesive unit to guarantee the communication takes place correctly. Conduits are data channels; they sit as a bridge between endpoints, with the endpoints responsible for using the conduit correctly (e.g., ensuring synchronous access). We use the term *connector objects* to denote the combination of *endpoint objects* and *conduit objects*, which are the realization of a particular connector.

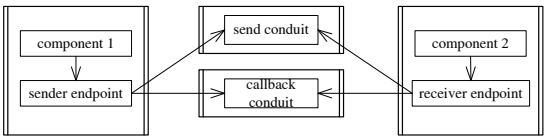
An example of a simple connector is the synchronous message buffer. It holds a single message, and the sender does not proceed until the receiver has received the message. The usage of this connector can be seen in the object diagram in Figure 3a, which is the SCOOP implementation of Figure 2a.

Another example is an asynchronous message queue with callback, where the sender sends its message, continues on, and then waits for a reply. The connector is implemented using two independent conduits; one conduit is responsible for carrying outgoing messages and the other for replies (this pattern is common in connectors with a reply). This is seen in the implementation given in Figure 3b, which is the SCOOP implementation of Figure 2d. The sender uses the conduits in two basic ways:

- Sending a message, along with its identity. This allows the receiving end to send a message back to it.
- Receiving the callback from the other end. The sender's identity is used once again to select the correct message to receive.



(a) Message buffer connector



(b) Message queue and callback connector

Fig. 3: Object diagrams for conduit and endpoints

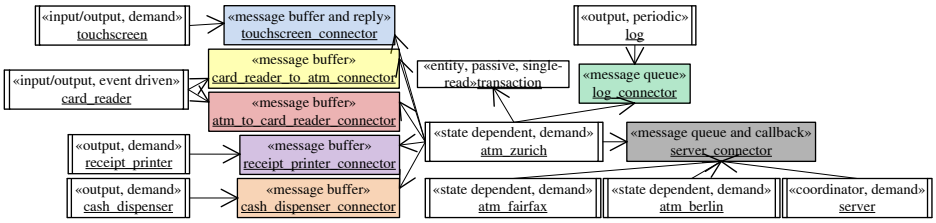
Since connector objects come in three parts: sender/receiver endpoints and the conduits, any component object that wants to use a connector must have access to the connector's endpoint functionality. This can either be done by creating an endpoint object, or inheriting from the appropriate endpoint class. Because the conduits are an implementation detail of the endpoints, component objects do not need a direct reference to the conduits.

4 Case study

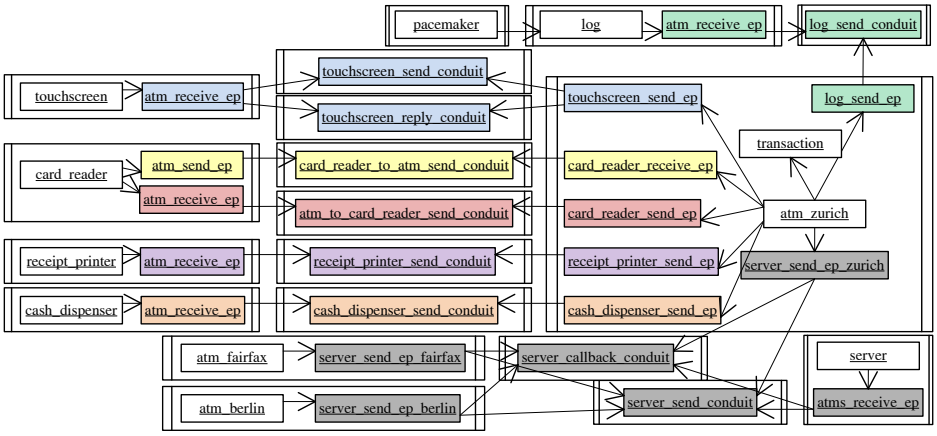
This section applies the suggested development method to an ATM system [1], shown in Figure 4a and available at [8].

Applying the design pattern implementations. Figure 4b shows the result of applying the design pattern implementations to Figure 4a. Active components become component objects handled by separate processors; passive components become component objects handled by one of the processors for an active component object. The class of a component object inherits from the framework class that corresponds to the component's stereotype. Connectors become conduit objects on separate processors and endpoint objects on existing processors.

Implementing interconnections. The root object sets up the objects representing control components, i.e., the server object and the ATM objects. It first creates the conduit objects that connect these control component objects. It then creates the control component objects using the conduit objects; the component objects then create local endpoint objects. After creation, the root object starts the new component objects. Each object representing a controlled component gets created by the controlling object. To do so, the control object first creates the conduit objects for the connectors along with local endpoint objects. It then creates the controlled object using the conduit objects.



(a) Design of the ATM system. Only one ATM with one customer is detailed; however, the server can be connected to multiple ATMs. To save space, the arrows omit the direction of the communication as done in Figure 2; instead, the names contain this information.



(b) SCOOP implementation of the ATM system. The boxes group objects handled by the same processor. Endpoint objects have the suffix `ep`. The names of the endpoint and conduit objects indicate the direction of the communication; for example, the atm_receive_ep object queries the touchscreen_send_conduit object to receive a message from the ATM. The colors link the connectors in Figure 4a to the resulting connector objects here.

Fig. 4: Design and implementation of the ATM system

Implementing interactions. The interactions between components can be implemented in the `start` features of the component objects. For instance, an ATM object executes a loop; each iteration begins with a message from the card reader object. Upon receiving this message, the ATM object proceeds according to the customer's choice. The server object executes a similar loop: it waits for messages from one of the ATM objects and acts accordingly.

Discussion. The case study was a manual effort; the development method has however potential for automation, using the following steps:

1. Generate one class for each component. The class inherits from the framework class corresponding to the component's stereotype. For each of the component's connectors, the class has a non-separate attribute for the connector's endpoint object; for each passive component, the class has a non-

separate attribute as well. The class has a creation procedure to initialize these attributes. For each connector, the creation procedure takes the connector's conduit objects as arguments and uses them to initialize the endpoint object. Finally, the creation procedure creates a non-separate component object for each passive component.
2. Generate one root class. The root object first creates the conduit objects for each connector. It then creates component objects on separate processors for each active component. It links the component objects according to the design by passing the conduit objects during construction. Lastly, the root object triggers the execution of all created component objects.
3. In each component class, add code for the component's interactions.

The first and second steps can be automated; the design contains the necessary information. However, it does not contain the data for the third step.

5 Related work

Software design patterns provide a tried and tested solution to a design problem in the form of a reusable template, which can be used in the design of new software applications. Software architectural patterns [9] address the high-level design of the software architecture [10, 11], usually in terms of components and connectors. These include widely used architectures [12] such as client/server and layered architectures. Design patterns [13] address smaller reusable designs than architectural patterns in terms of communicating objects and classes customized to solve a general design problem in a particular context. The patterns described in this paper are aimed at developing concurrent applications and are hence different from patterns for sequential applications.

Component technologies [11] have been developed for distributed applications. Examples of this technology include client-side Java Beans and server-side Enterprise Java Beans (EJB). Patterns for concurrent and networked objects are comprehensively described in [14]. However, these patterns are not used to systematically derive a concurrent program from a design, as it is the case in our approach. Pettit and Gomaa [15] represent UML models using colored Petri nets to conduct behavioral analyses (e.g., timing behavior). Our work focuses on obtaining an executable system with built-in behavioral guarantees.

6 Conclusion

With the increasing need of concurrency, offering adequate support to developers in designing and writing concurrent applications has become an important challenge. The approach taken in this paper is to base such support on widely used architectural modeling principles, namely UML with the COMET method, which should simplify adoption in industrial settings. We defined a mapping of COMET's behavioral design patterns into SCOOP programs and demonstrated

with a case study that using this approach entire concurrent UML designs can be systematically mapped to executable programs.

For future work, it would be interesting to integrate our method with other approaches based on UML and the COMET method, giving rise to a more comprehensive framework with additional analyses of concurrent designs. In the long term, we would also like to provide an automated method to translate UML concurrent software architecture designs to an implementation.

Acknowledgments This work was funded in part by the ERC under the EU's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). Hassan Gomaa thanks Bertrand Meyer for the opportunity to work, during his sabbatical, with the Chair of Software Engineering group at ETH.

References

1. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley (2000)
2. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall (1997)
3. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. PhD thesis, ETH Zurich (2007)
4. Nanz, S., Torshizi, F., Pedroni, M., Meyer, B.: Design of an empirical study for comparing the usability of concurrent programming languages. In: ESEM'11. (2011) 325–334
5. West, S., Nanz, S., Meyer, B.: A modular scheme for deadlock prevention in an object-oriented programming model. In: ICFEM'10, Springer (2010) 597–612
6. Morandi, B., West, S., Nanz, S., Gomaa, H.: Concurrent object-oriented development with behavioral design patterns. Technical report, `http://arxiv.org/abs/1212.5491` (2012)
7. Gomaa, H.: Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. Cambridge University Press (2011)
8. SCOOP implementations of design patterns. `https://github.com/scottgw/scoop_design_patterns` (2013)
9. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)
10. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall (1996)
11. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley (2009)
12. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. 2nd edn. Addison-Wesley (2003)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley (2000)
15. Pettit, IV, R.G., Gomaa, H.: Modeling behavioral design patterns of concurrent objects. In: ICSE'06, ACM (2006) 202–211