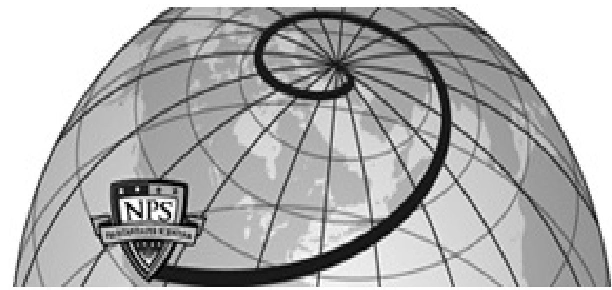
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

2015

Composition of Behavior Models for Systems Architecture

Whitcomb, Clifford A.; Auguston, Mikhail; Giammarco, Kristin

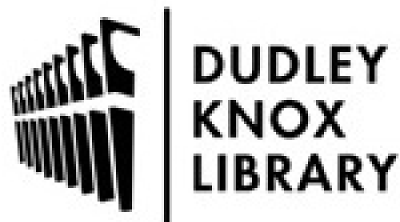
John Wiley & Sons

Whitcomb, Clifford A., Mikhail Auguston, and Kristin Giammarco. "Composition of Behavior Models for Systems Architecture." Modeling and Simulation Support for System of Systems Engineering Applications (2015): 361-391.

http://hdl.handle.net/10945/59427

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



DUDLEY KNOX LIBRARY

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

Chapter 14

Composition of Behavior Models for Systems Architecture

Clifford A. Whitcomb, Mikhail Auguston, and Kristin Giammarco
Naval Postgraduate School, Monterey, CA, USA

14.1 INTRODUCTION

The specification of a system's architecture has emerged in the last two decades as one of the fundamental concepts in systems and software engineering. ISO (2011) defines architecture as the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." The current interest in understanding architecture and applying the methods across new disciplines as a basis for systems design and evaluation can be tied to recent systems failures and the fact that many can be traced to problems in their early stage definition (Maier and Rechtin, 2000). Architecture development methods have been used for many complex situations, even when the designers were not aware that this was the case. "Architectural methods, similar to those formulated centuries before in civil works, were being used, albeit unknowingly, to create and build complex aerospace, electronic, software, command, control, and manufacturing systems" (Maier and Rechtin, 2000). Indeed, the concept of developing architecture is very old, predating engineering, and continues to this day. "Architecting, the planning and building of structures, is as old as human societies—and as modern as the exploration of the solar system" (Rechtin, 1991). Architecture provides structure for stakeholders of a system of interest to express their

Modeling and Simulation Support for System of Systems Engineering Applications, First Edition.
Edited by Larry B. Rainey and Andreas Tolk.
© 2015 John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons, Inc.

respective needs and wants and plays a role as the bridge between those needs, requirements, and implementation of a system. Decisions made at the architecture level during the earliest conceptual stages propagate through to detail design and then beyond into the implementation and operation. Errors exposed during conceptual architecture development and design can be corrected less expensively using models than those discovered during later life cycle phases of testing and implementation. Earlier discovery of design problems, especially those related to stakeholder needs and engineering feasibility, is the motivation behind a new approach to formal systems and software architecture specification presented in this chapter.

Consider this definition of a system of systems (SoS): "a set or arrangement of systems that results when **independent** and task-oriented systems are **integrated into a larger systems construct**, that delivers unique capabilities and functions in support of missions that cannot be achieved by individual systems alone" (Vaneman and Jaskot, 2013, boldface added by author). For our SoS modeling approaches to support **independent** systems behavior models and their subsequent **integration**, they must address each system and the systems interactions as separate concerns. Separation of concerns is a design principle adopted by the software engineering and computer science communities to write highly cohesive software modules such that each module is associated with exactly one main function and to reduce unnecessary coupling among modules within a software program, such that a given module needs to access only a minimum of other modules to perform its functions. Just as programmers use this principle to keep their code organized and maintainable, systems and SoS engineers may use this concept in structuring their systems behavior models.

The modeling community uses many terms to describe the physical manifestation of natural and technological objects. Among these terms are system, SoS, object, component, performer, actor, asset, participant, and so on. These natural language terms may take on different meanings in different communities who adapt a term for their use. For simplicity, and to amplify the hierarchical nature of physical entities, the term **component** is primarily used throughout the remainder of this chapter, with a few exceptions where it is necessary to make a particular point about a component being a part of a larger system. When the term **component** is used, this is a reference to any type of physical entity at any level of abstraction, from an SoS to a configuration item, from the universe to a subatomic particle, from a human person to human-created technology, from a concept to a creation, and from hardware to software to organization. In other words, a **component** can be anything that exhibits behavior. If another term that fits this description is preferred, the reader is encouraged to make the substitution, since the hierarchical nature of these behavior-exhibiting building blocks is more important than the label given to the building blocks.

The new behavior-modeling approach, called Monterey Phoenix (MP), is predicated on the following foundational premises:

- A component's **behavior** is central to stakeholder satisfaction. Behavior is the way in which a component acts on its own and responds to stimuli. For human-designed components, predicting functional and dysfunctional behavior during the design stage reduces the risk of stakeholder dissatisfaction and lack of engineering feasibility during the component's operation. Modeling the behavior of components of complex natural and technological systems increases human understanding of overall systems behavior.

- Modeling a component in the context of its environment is necessary, but not sufficient, for predicting the full range of component behaviors before physical implementation. Modeling the behavior of **each** interacting component rather than just one component's interactions with external components has the potential to expose many more design flaws and tacit assumptions pertaining to the component operation in a larger construct.
- Describing component interactions at a high level of abstraction, orthogonal to descriptions of component behavior, enables automatic solving for distinct instances of behaviors (scenarios, use cases) from an exhaustive superset of possible behaviors and early testing of systems behavior against stakeholder expectations/requirements using scenario inspection and assertion checking. The assumption related to an "exhaustive superset" is supported up to the scope limit. This concept is addressed in more detail later in the chapter.

14.2 COMMON CHARACTERISTICS FOR ARCHITECTURE DESCRIPTIONS

Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to satisfy the requirements and serve as a basis for the design (Perry and Wolf, 1992). An architecture description has converged on the concept of architectural elements, such as component, connector, and relationships among them. "When designers discuss or present a software architecture for a specific system, they typically treat the system as a collection of interacting components. Components define the primary computations of the application. The interactions or connections between components define the ways in which the components communicate or otherwise interact with each other" (Abowd et al., 1995). A conclusion in Rozanski and Woods (2012) states: "Every system has an architecture, whether or not it is documented and understood."

The following aspects have emerged as characteristic for architecture descriptions (Perry and Wolf, 1992; Bass et al., 2003):

- An architecture description belongs to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.
- An architecture specification should be supportive for the refinement process and needs to be checked carefully at each refinement step (preferably with tools).
- There should be flexible and expressive composition operations for the refinement process.
- The architecture specification should support the reuse of well-known architectural styles and patterns. Practice has provided several well-established, reusable architectural solutions.
- An architecture of a system should be considered in the context of the environment in which it operates, as suggested in the international standard ISO/IEC IEEE 42010 "Systems and Software Engineering Architecture Description" (ISO, 2011).

- The software architect needs a number of different views of the software architecture for the various uses and users (Kruchten, 1995) (including visual representations, like diagrams).

MP utilizes these characteristics and complements existing languages and notations by extending them to include an abstract interaction specification capability.

14.3 RELATED WORK

The following ideas of behavior modeling and formalization have provided inspiration and insights for the MP approach.

Literate programming introduced in Knuth (1984) set the directions for hierarchical refinement of structure mapped into behavior, with the concept of pseudocode and tools to support the refinement process.

Campbell and Habermann (1974) and Bruegge and Hibbard (1983) have demonstrated the application of path expressions for program monitoring and debugging. Path expressions in Perry and Wolf (1992) have been used (semiformally) as a part of software architecture description.

Hoare's Communicating Sequential Processes (CSP) (Hoare, 1985; Roscoe, 1997) is a framework for process modeling and formal reasoning about those models. This behavior-modeling approach has been applied to software architecture descriptions to specify a connector's protocol (Allen, 1997; Allen and Garlan, 1997; Pelliccione et al., 2009).

Rapide (Luckham and Vera, 1995; Luckham et al., 1995) uses events and partially ordered sets of events (posets) to characterize component interaction.

Statecharts (Harel, 1987) became one of the most common behavior-modeling frameworks, integrated in broader modeling and specification systems (Unified Modeling Language (UML) (Booch et al., 2000) and AADL (Feiler et al., 2009)). UML has four behavior diagrams: activity, sequence, state machine, and use case.

Wang and Parnas (1994) have proposed to use trace assertions to formally specify the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior. The approach is based on algebraic specifications and term rewriting.

The Alloy modeling framework (Jackson, 2007) has strongly influenced this work through ideas of integration of sets and first-order predicate logic within the relational logic framework; inheritance structure; emphasis on lightweight formal methods as opposed to the full-scale theorem proving, with the fundamental concept of small scope hypothesis; and the principles of immediate feedback and visualization during model design.

The concept of software behavior models based on events and event traces was introduced in Auguston (1991, 1995) and Auguston et al. (2002, 2006) as an approach to software debugging and testing automation. The early draft of MP has appeared in Auguston (2009a, b).

14.4 THE MP APPROACH TO BEHAVIOR MODELING

The behavior of the system is usually the main concern for the developer, and the presence of unintended behaviors manifests errors in the design and ultimately the implementation and operation of the system. Many detectable errors made early in the systems design go

undetected until later in the development life cycle, when they are more expensive to fix. For example, systems architects may be interested in detecting errors in a system's interaction with the operational environment, for example, by querying a systems model to find scenarios that contain potential hazard states. SoS architects are concerned with detecting emergent behaviors resulting from the interactions of subsystems, some of which may lead to undesirable behavior.

The considerations in Section 14.2 suggest the importance of architecture models and the practical need to test and verify the systems architecture early in the design phase. Behavior modeling is at the core of the MP systems and software architecture modeling framework, which has the following main principles:

- A view of the architecture as a high-level description of possible systems behaviors, emphasizing the behavior of subsystems and interactions between subsystems.
- The concurrency of actions is a default, unless ordering is imposed (thus representing a design decision introducing a dependency between activities).
- Specifying the interaction between the system and its environment is important. A model of the system and its environment behaviors and interactions can be a contribution to the system's requirements specification.
- The event grammar provides a view of the behavior as a set of actions (event trace) with two basic relations, where the PRECEDES relation captures the dependency abstraction, and the IN relation represents the hierarchical relationship. Since the event trace is a set of events, additional constraints can be specified using set-theoretical operations and predicate logic.
- The behavior composition operations support architecture reuse and refinement toward design and implementation models.
- The MP architecture description is amenable to deriving different views, including a structural view (traditional architecture box-and-arrow diagrams) or those desired by the Department of Defense Architecture Framework (DoDAF) (DoD, 2009).
- The executable architecture models provide the possibility to automatically generate examples of behaviors (use cases) for early systems architecture testing and verification with tools.

The main objective of the MP approach is to provide a formal framework for specifying behaviors of the system, its parts and environment, and the interaction between them. From a Systems Engineering point of view, the following two main principles of MP are the key for complex system and SoS behavioral analysis:

- In addition to modeling the behavior of the system along with its interfaces to external systems, also model the behavior of the environment in which the system operates.
- Model component interactions abstractly and separately, rather than instantiated in specific use cases.

The MP approach provides extensions to current modeling notations to significantly expand the coverage of the design space explored. MP does this by applying a *separation of concerns* that has previously not been done in behavior modeling. Specifically, the MP approach leverages the power of abstraction to model internal and external *interactions* among components as a separate concern from the *behavior* of each component to extract the overall possible behaviors of all components acting together. Separation of concerns

is a design principle adopted by the software engineering and computer science communities that aids in the writing of highly cohesive software modules such that each module is associated with exactly one main function and to reduce unnecessary coupling among modules within a software program such that a given module needs to access only a minimum of other modules to perform its functions. This same concept is accessible to systems architects through MP to structure and organize systems behavior models, just as programmers have used it to keep their code organized and maintainable. The partitioning of component behavior models and the component interaction specification into separate concerns enables the component behaviors and interactions to be woven together during model execution, automatically generating use cases from the separate behavior and interaction specifications.

MP models may be used for early design testing and verification, for early performance and safety assessment estimates, and for generating examples of scenarios (use cases), which in turn can be used to support test case construction and monitor for systems implementation testing. MP architecture models can be integrated into standard frameworks, like UML, Systems Modeling Language (SysML), and DoDAF, providing the level of abstraction convenient for architecture models with the emphasis on behavior and interaction aspects (see Example 14.7 in Section 14.5 for more details).

14.5 MODELING COMPONENT BEHAVIOR

In a certain sense, an executable architecture model is a compact description for a set of required behaviors. The architecture model—a finite object by itself—may specify a potentially infinite number of execution paths. Computers are used to solve problems usually by finding an algorithm that describes these possible execution paths and mapping it on the appropriate computational platform, that is, by applying a step-by-step procedure to design a behavior to solve the problem at hand. A component operates in a certain environment, which has its own behavior that interacts with the system and causes systems responses. In the MP approach, behavior of the environment in which a component operates is described in addition to the system itself to increase the likelihood of predicting these responses.

The behavior of a system of components is defined in MP as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of an event trace is specified using event grammars and other constraints organized into schemas. Behaviors for both system and its environment are specified within the same framework. Suggested composition operations on schemas are based on event pattern matching and provide for behavior merging and abstract interface specification. The schema framework is amenable to stepwise refinement, reuse, visualization of multiple architecture views, and application of automated tools for consistency checks and systems behavior verification early in the design process.

14.5.1 Event Concept

The MP behavior model is based on the concept of an event as an abstraction of activity. The event has a beginning and an end and may have duration (a time interval during which the action is accomplished). The behavior of a system is modeled as a set of events with two binary relations defined for them: precedence (PRECEDES) and inclusion

(IN)—the event trace. One action is required to precede another if there is a dependency between them, for example, the send event should precede the receive event. Events may be nested, when a complex activity contains a set of other activities. Imposing one of these basic relations on a pair of activities represents an important design decision. Usually, systems behavior does not require a total ordering of events. Both PRECEDES and IN are partial ordering relations. If two events are not ordered, they may occur concurrently. Appendix 1 in Auguston and Whitcomb (2012) provides more details specifying the properties of the basic relations.

14.5.2 Event Grammar

MP uses an event grammar that allows for the compact specification of behavior for each component. Events are abstractions of activities that may be experienced from the perspective of system or its environment. Data inputs and outputs are not modeled in a separate class as in an Enhanced Functional Flow Block Diagram (EFFBD) and other data flow-oriented notations, but are represented by actions (events) that may be performed on that data, following the concept of abstract data types (ADT) introduced in Liskov and Zilles (1974). Behavior is modeled in MP as an algorithm for each component, describing the step-by-step procedure by which it achieves a well-defined goal.

Events have two main binary relations used to construct *event traces*, or particular instances of behavior. Sequencing of events is denoted using the PRECEDES relation, and decomposition of events is denoted using the IN relation. An event grammar rule specifies structure for a particular event type in terms of these two relations and has the form

```
A: right-hand-part;
```

where **A** is an event type name. Event types that do not appear in the left-hand part of rules are considered atomic and may be refined later by adding corresponding rules. More details about event grammar notation can be found in Auguston (2009a). For brevity, this chapter only describes the composition operations that appear in the example models.

Events are composed to describe possible event traces using composition operations in the right-hand part of the event grammar rule. The composition operations comprise an algorithm for each root event. Behavior is described using composition operations such as ordered sequence of events **A B C**; alternative **(A | B | C)**; ordered iteration **(* A B C *)** (A B C repeated zero or more times); **(+A B C +)** (one or more times); optional event **[A]**; **{A, B, C}**, set of unordered (potentially concurrent) events; **{* A *}**, set of zero or more of unordered events A; and **{+A +}**, set of one or more of unordered events. An event grammar, as in Example 14.1, is essentially a graph grammar, which specifies directed acyclic graphs of events with the arcs representing relations IN and PRECEDES.

Similar to context-free grammars, event grammars can be used as production grammars to generate instances of event traces, as in Example 1.

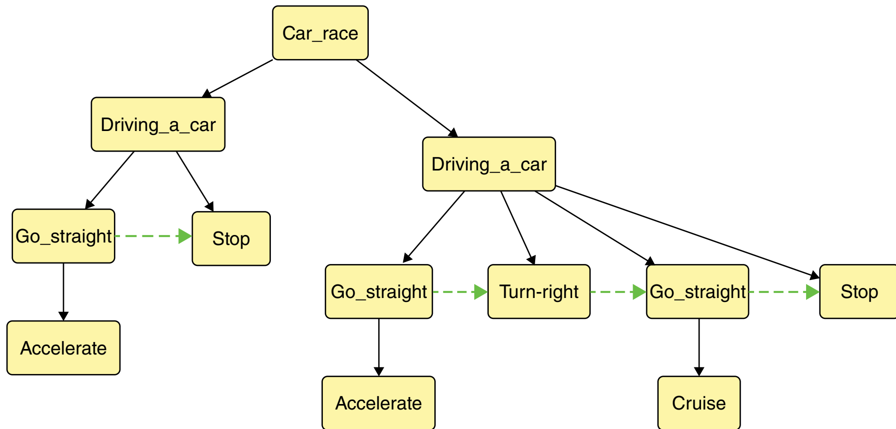


Figure 14.1 An event trace derived from the event grammar in Example 14.1.

Example 14.1 An event grammar for car race scenarios.

```
car_race:       {+ driving_a_car +};
driving_a_car: go_straight (* ( go_straight |
turn_left | turn_right ) *) stop;
go_straight:    ( accelerate | decelerate | cruise );
```

An instance of an event trace satisfying the grammar can be visualized as a directed graph with two types of edges (one for each of the basic relations) (Figure 14.1).

14.6 MODELING COMPONENT INTERACTION AND ARCHITECTURE VIEWS

The behavior of a particular system is specified as a set of all possible event traces using a *schema*. The concept of the MP schema has been inspired by the Z schema (Spivey, 1989). The purpose is to define the structure of all possible event traces (in terms of IN and PRECEDES relations) using event grammar rules and other constraints. A schema usually contains a collection of events called *roots* representing the behaviors of parts of the system (e.g., components and connectors in common architecture descriptions), *composition operations* specifying interactions between these behaviors, and additional constraints on root behaviors.

There is precisely one instance of each root event in any trace. The schema also may contain auxiliary grammar rules defining composite event types used in other rules. Roots in turn may be defined as schemas, thus providing for architecture reuse and composition. A schema may define both finite and infinite traces, but most analysis tools for reasoning about a system's behavior assume that a trace is finite.

The schema represents instances of behavior (event traces), in the same sense as Java source code represents instances of program execution. Just as a particular program

execution path can be extracted from a Java program's source code by running it on a Java virtual machine (JVM), a particular event trace specified by an MP schema can be generated from the event grammar rules by applying behavior composition operations and constraints.

In addition to describing specific systems behavior, MP can also be used to describe behavior patterns. In order to establish coordination between sending and receiving messages, we use the behavior composition operation **COORDINATE**. In Example 14.2, the composition operation takes two traces and defines a modified event trace (merges behaviors of Task_A and Task_B) by adding the **PRECEDES** relation between the selected **send** and **receive**.

Example 14.2 A simple pipe/filter architecture pattern.

```
SCHEMA simple_message_flow
ROOT Task_A:  (* send *);
ROOT Task_B:  (* receive *);
COORDINATE    (* $x: send *)  FROM Task_A,
    (* $y: receive *)  FROM Task_B  ADD $x PRECEDES $y;
```

The first part of composition operation (the source) uses event patterns to specify segments of root traces that should be selected. The **(* $x: send *)** pattern identifies the sequence of totally ordered **send** events (with respect to the transitive closure of **PRECEDES** relation—**PRECEDES***). Use of the **(* P *)** pattern for selection means that all events P in the source root should be ordered, both iterations should have the same number of selected elements (**send** events from the first trace and **receive** events from the second), and the pair selection follows this ordering (**synchronous coordination**). Labels **$x** and **$y** provide access to the events selected within each iteration. The **ADD** composition completes the behavior adjustment, specifying that an ordering relation will be imposed on each pair of selected events. Behavior specified by this schema is a set of matching event traces for Task_A and Task_B with the modifications imposed by the composition.

The composition operation may be considered as an abstract interaction description for root behaviors. In the case when **asynchronous coordination** is needed, an iterative set pattern can be used. For example,

```
COORDINATE  {* $x: E1 *} FROM A, {* $y: E2 *} FROM B ADD
$x PRECEDES $y;
```

In this case, matching root traces for A and B still should contain an equal number of selected events of types E1 and E2, correspondingly. But now the resulting merged traces will include all permutations of events E2 from B matching events E1 from A, with the **PRECEDES** relation imposed on each selected pair. This assumes that other constraints, like the partial ordering axioms from Appendix 1 in Auguston and Whitcomb (2012), are satisfied. Each permutation yields one potential instance of a resulting trace for the schema deploying this composition. In order to reduce the exponential explosion, optimizations similar to symmetry reduction in model checking tools should be considered. Changing **(* … *)** for **{* … *}** in Example 14.2 may increase the number of composed traces in the schema.

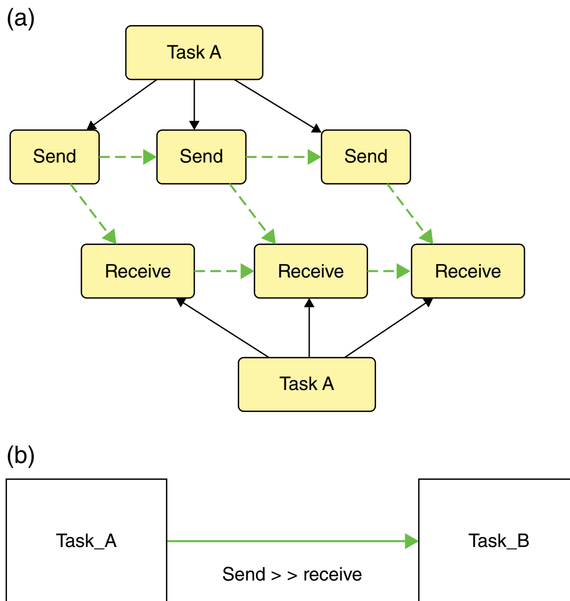


Figure 14.2 An example of a composed event trace and corresponding architecture view for the simple_message_flow schema. (a) The composed event trace for the simple_message_flow schema is labeled. (b) The architecture view for the simple_message_flow schema is labeled.

Different views for different stakeholders can be extracted from MP schemas. For example, each root may be visualized as a box (Figure 14.2), and if there is a composition operation specifying an interaction between root behaviors, the boxes are connected by an arrow marked by the interaction type. The root behavior may be visualized with UML activity diagrams (Booch et al., 2000) (see Figure 14.6). The MP developer's environment may have a library of predefined views providing different visualizations for schemas.

Data items in MP are represented by actions (events) that may be performed on that data. This principle follows the ADT concept introduced in Liskov and Zilles (1974), as in Example 14.3.

Example 14.3 Data flow.

```
SCHEMA Data_flow
ROOT Process_1:   (* work write *);
ROOT Process_2:   (* ( read | work ) *);
ROOT File:        (* write *) (* read *);
Process_1, File SHARE ALL write;
Process_2, File SHARE ALL read;
```

Behavior of the file requires that all **write** operations should be completed before any **read** operations. The view of this schema in Figure 14.3b renders root interaction with a line where the shared event name is attached as a label.

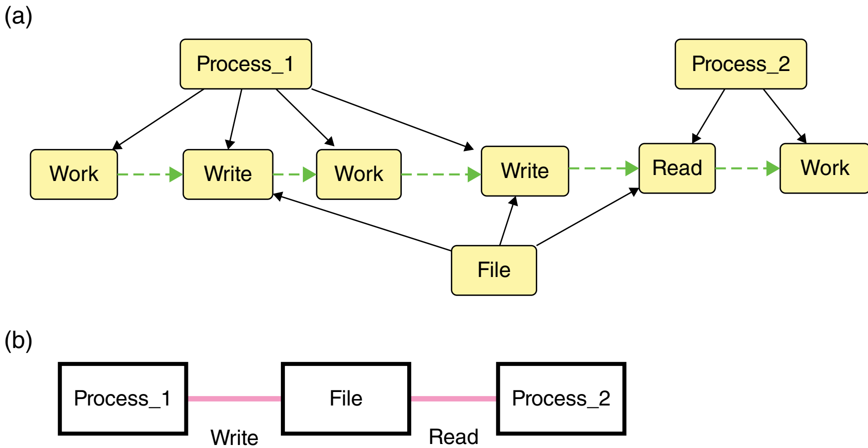


Figure 14.3 An example of a composed event trace and corresponding architecture view for the Data_flow schema. (a) The composed event trace for the Data_flow schema is labeled. (b) The architecture view for the Data_flow schema is labeled.

The schema in Example 14.4 specifies the behavior of a stack in terms of stack primitive operations.

Example 14.4 Stack behavior.

```
SCHEMA Stack
ROOT Stack_operation:  (* ( push | pop ) *);
SATISFIES FOREACH $x:  pop FROM Stack_operation
   (Number_of (pop) before ($x) < Number_of (push)
before ($x));
```

Let **IN*** denote the transitive closure of the **IN** relation (similarly as **PRECEDES*** is a transitive closure for **PRECEDES**). The domain of the universal quantifier is the set of all **pop** events **e** such that **(e IN* Stack_operation)**. The function **Number_of (pop) before ($x)** yields the number of **pop** events **e** such that **(e PRECEDES* $x)**. The set of event traces specified by this schema contains only traces that satisfy the constraint. This example presents a filtering operation as yet another kind of behavior composition.

The reuse of a schema is demonstrated through Example 14.5.

Example 14.5 Reuse of a schema.

```
SCHEMA Two_stacks_in_use
INCLUDE Stack;
ROOT Main:  {* (do_something | use_S1 | use_S2) *};
   use_S1:  (push | pop);
   use_S2:  (push | pop);
ROOT S1:    Stack;
ROOT S2:    Stack;
```

```
S1, Main SHARE ALL $x:  (pop | push) SUCH THAT
Has_enclosing (use_S1)($x) WITHIN Main;
S2, Main SHARE ALL $x:  (pop | push) SUCH THAT
Has_enclosing (use_S2)($x) WITHIN Main;
```

The **INCLUDE** statement brings the schema Stack into the scope. This means that all constraints specified in the Stack also will be included. The rule for Main is intentionally left lax without imposing any specific ordering on embedded activities. Roots **S1** and **S2** represent the presence of two independent stacks as data items. The ordering of **pop** and **push** events inside **use_S1** and **use_S2** in each stack behavior is ensured and will be brought into the resulting trace by the included Stack behaviors as a result of sharing these events with the Stack behavior. The **SHARE ALL** composition operation uses event patterns and context conditions to accomplish the necessary event trace construction. The predicate **Has_enclosing(T)(e1)** is true iff there exists an event e2 of the type T in the trace specified by the **WITHIN** clause such that **e1 IN* e2**.

Predicates and functions like **Has_enclosing(T)(e)** and **Number_of (T) before (e)** are used for convenient navigation in the event graphs.

Connectors and components, which are core elements in an architecture description, can be uniformly modeled in MP as behaviors. The idea that connectors should be elevated to the first-class-citizen status on a par with components is often discussed in the literature, for example, in Taylor et al. (2010), as in Example 14.6.

Example 14.6 Connectors and components.
Suppose that the communication between the components is implemented via a buffer of size **max_buffer_size** and not necessarily all sent messages are consumed, that is, some of them could stay in the buffer indefinitely. Each message may be consumed no more than once, and the ordering of receiving does not necessarily correspond to the ordering of sending. The root **Buffered_channel** simulates the behavior of a connector between **Task_A** and **Task_B**. This behavior model does not provide details about what happens after a buffer overflow event:

```
CHEMA Buffered_transaction
ROOT Task_A:: (* Send *);
ROOT Task_B:: (* Receive *);
ROOT Buffered_channel: {* (Send [Receive]) *}
(Overflow | Normal);

Task_A, Buffered_channel SHARE ALL Send;
Task_B, Buffered_channel SHARE ALL Receive;
SATISFIES FOREACH $x: Receive FROM Buffered_channel
         ( Number_of (Send) before ($x) - Number_of
(Receive) before ($x) ) <= max_buffer_size;
SATISFIES FOREACH $x:  Overflow FROM Buffered_channel
         ( Number_of (Send) before ($x) - Number_of
(Receive) before ($x) ) > max_buffer_size;
SATISFIES FOREACH $x:  Normal FROM Buffered_channel
         ( Number_of (Send) before ($x) - Number_of
(Receive) before ($x) ) <= max_buffer_size;
```

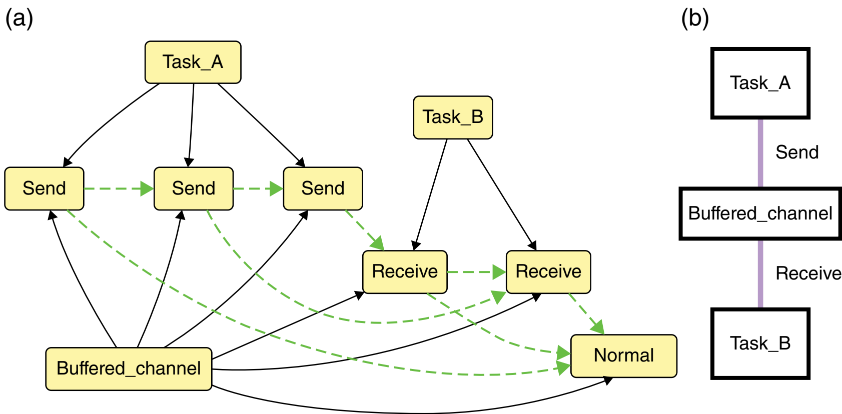


Figure 14.4 An example of an event trace and corresponding architecture view for the Buffered_transaction schema. (a) The event trace (without overflow) for the Buffered_transaction schema with max_buffer_size = 3 is labeled. (b) The architecture view for the Buffered_transaction schema is labeled.

If the schema should satisfy only behaviors without buffer overflow, the three **SATISFIES** conditions above can be replaced by the following constraint (and the **Overflow** event can be removed from the schema):

```
SATISFIES FOREACH $x:  Send FROM Buffered_channel
   Number_of ($y: Send) before ($x) SUCH THAT ( ¬
Has_next(Receive)($y)) < max_buffer_size;
```

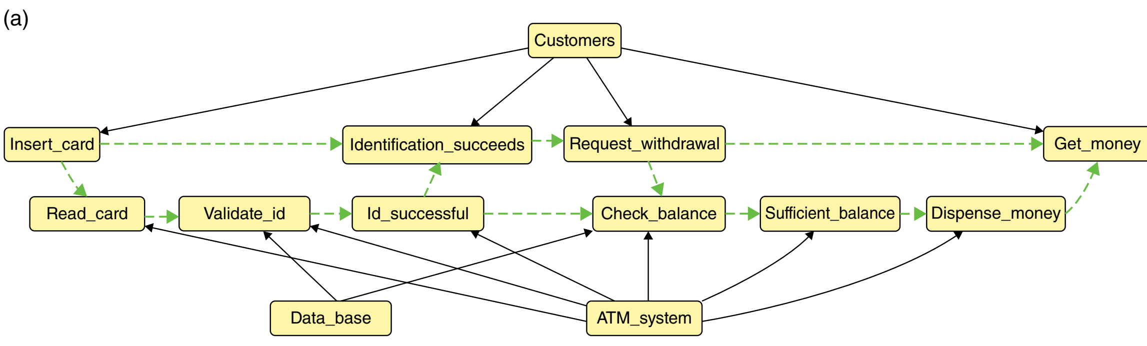
Note that **PRECEDES** relation is defined explicitly either in the grammar rule or by **ADD** composition operation and is a proper subset of its transitive closure **PRECEDES***. The predicate **Has_next(T)(e1)** is true iff there exists an event **e2** of the type **T** in the trace such that **e1 PRECEDES e2** (Figure 14.4).

Example 14.7 demonstrates how to integrate the behavior of an environment with the behavior of a system (Figures 14.5 and 14.6). The ATM_withdrawal schema specifies a set of possible interactions between the Customer, ATM_system, and Data_Base. An event trace generated from this schema can be considered as a use case example.

Example 14.7 Withdraw money from ATM.

```
SCHEMA ATM_withdrawal
ROOT Customer:    (* insert_card
    ( (  identification_succeeds
         request_withdrawal
        ( get_money | not_sufficient_funds )  )   |
       identification_fails  )    *);
ROOT ATM_system:  (* read_card   validate_id
    ( id_successful check_balance
      (   (sufficient_balance dispense_money) |
          unsufficient_balance )   |
     id_failed  )   *);
ROOT Data_Base:  (* (validate_id | check_balance) *);
```

(a)



(b)

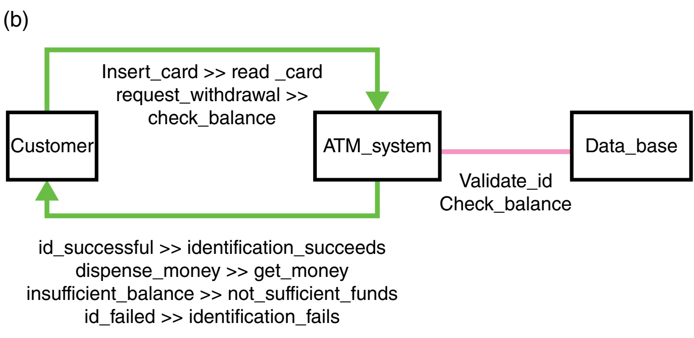


Figure 14.5 An example of an event trace and corresponding architecture view for the ATM_withdrawl schema. (a) The event trace for the ATM_withdrawal schema is labeled. (b) The architecture view for the ATM_withdrawal schema is labeled.

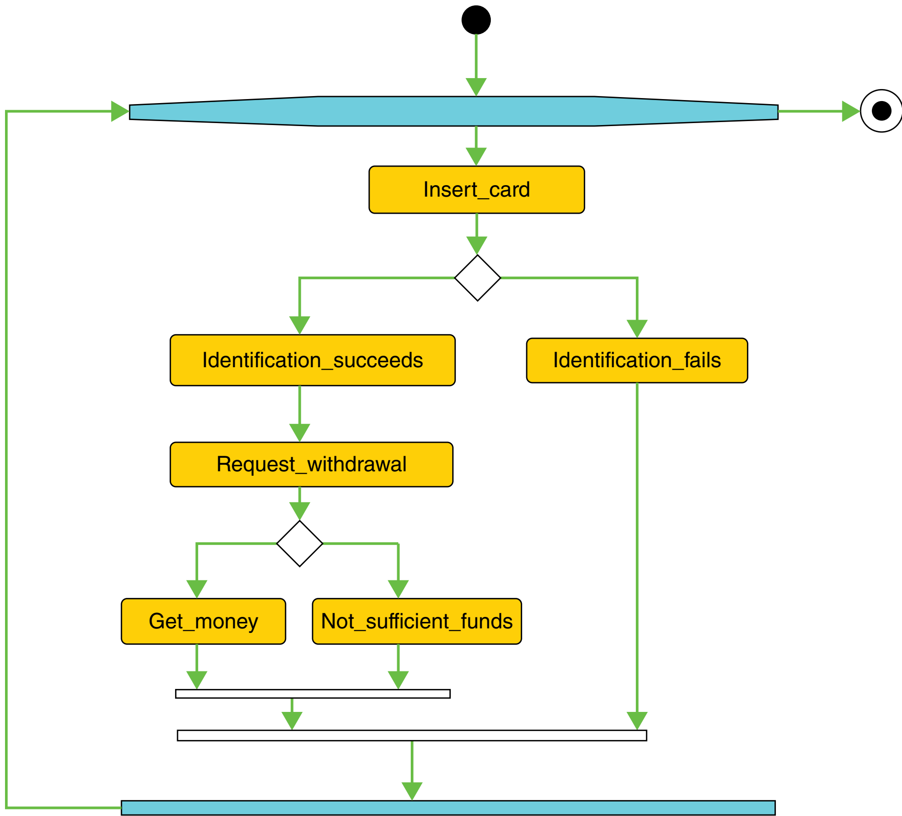


Figure 14.6 A view on the Customer root event behavior as a UML activity diagram.

```
Data_Base, ATM_system SHARE ALL validate_id, check_balance;

COORDINATE (* $x: insert_card *)  FROM Customer,
    (* $y: read_card *) FROM ATM_system ADD $x PRECEDES $y;
COORDINATE (* $x: request_withdrawal *) FROM Customer,
            (* $y: check_balance *)       FROM ATM_system
ADD $x PRECEDES $y;
COORDINATE (* $x: identification_succeeds *) FROM Customer,
    (* $y: id_successful *) FROM ATM_system ADD $y
PRECEDES $x;
COORDINATE (* $x: get_money *) FROM Customer,
    (* $y: dispense_money *) FROM ATM_system ADD $y
PRECEDES $x;
COORDINATE (* $x: not_sufficient_funds *) FROM Customer,
    (* $y: unsufficient_balance *) FROM ATM_system ADD $y
PRECEDES $x;
COORDINATE (* $x: identification_fails *) FROM Customer,
    (* $y: id_failed *) FROM ATM_system ADD $y PRECEDES $x;
```

If the view of the whole system's behavior emphasizing the interaction between the parts (components) can be visualized as in Figure 14.5b, the view of root's stand-alone behavior can be visualized as a UML activity diagram (Figure 14.6 provides an example for the Customer root behavior). Since event aggregates (iterations, alternatives, sets) in MP are well structured, it is possible to use Nassi–Shneiderman diagrams (Nassi and Shneiderman, 1973) as yet another kind of view. The event trace on Figure 14.5a can be viewed as an analog of UML sequence diagram's "swim lanes" for the Customer and ATM_system interaction. This example demonstrates that MP models can be integrated into standard frameworks, like UML, SysML, and DoDAF, providing the level of abstraction convenient for architecture models, where, in particular, MP focuses on the interaction aspects.

14.7 MERGING SCHEMAS

So far, we have seen examples of assembling schemas using previously defined schemas (Example 14.5). Each schema in the assembly holds its own roots and composition operations (**SATISFIES** filter and interaction constraints, like **COORDINATE** and **SHARE ALL**) within its scope.

The join operation for schemas looks like:

```
SCHEMA A EXTENDS B
Roots for A
Constraints and composition operations involving roots
from both A and B
```

The resulting schema A joins roots defined in A and roots defined in B, merges within its scope all constraints and composition operations defined in B, and may have additional constraints and composition operations involving all roots. A typical use of such schema composition may be for assembling the architecture of an SoS from the architectures of its constituent systems.

14.8 COMPARISON OF MP WITH COMMON SYSTEMS ENGINEERING NOTATIONS

MP complements and extends Systems Engineering behavior-modeling notations. The Functional Flow Block Diagram (FFBD) notation was developed in the 1950s to show systems functions and their chronological order of execution (NASA, 2007). The EFFBD was developed in the 1990s to show information flow on the diagrams as inputs/triggers and outputs (Long, 2000). The notion of an xFFBD has been proposed to extend EFFBD with additional formalisms to make it more expressive (Aizier et al., 2012). The SysML (OMG, 2012) was developed to extend UML for application on the systems scale and directly reuses all behavior diagrams except the activity diagram, which has been modified from UML for consistency with the EFFBD and to support a continuous flow of

matter or energy (Friedenthal et al., 2006). Although these notations have been successfully used in modeling slices of systems behavior and interaction, none are presently used to model the behavior of each component and the interaction of each component with other components in its environment, as separate concerns, nor do existing frameworks such as the DoDAF (DoD, 2009) address this separation of concerns when describing event-based interactions.

Many component models describe only a subset of possible behaviors with assumptions about possible component interactions in specific scenarios or use cases, since behavior of external components may be outside the scope of the component under design. This practice prevents the opportunity to observe behaviors that result from combinations of interactions that fall outside the scope of the assumptions made about external component behavior.

Example 14.8 considers a simple user authentication scenario done internal to a system.

Example 14.8 User authentication.
User provides a general identification.

1. System requests unique identification.
2. User provides a unique identification.
3. If the credentials are valid, the System authorizes the User to access the services; otherwise, the System notifies the User that credentials are invalid and the user may reattempt access up to two more times.
4. The User or the System ends the session.

This narrative gives rise to at least two possible use cases: user authentication succeeds and user authentication fails. The EFFBD activity model in Figure 14.7 is a first attempt at graphically depicting behavior for the scenario shown earlier, minus the access reattempts. The EFFBD uses functional **activities** transforming inputs into outputs, **exit conditions** documenting possible outcomes of an activity, and **inputs/triggers and outputs** consisting of matter or energy consumed or produced by an activity. The approach taken in the diagram illustrates some generally accepted conventions when modeling with EFFBDs, such as allocating the activities of each main component taking part in the thread (in this case the User (an "external" component) and the System (component under design)) onto its own primary branch, similar to the use of swim lanes on a UML or SysML activity diagram (Long, 2000; Long and Scott, 2011; Armstrong, 2013).

In this example, conditions leading to different possible behaviors based on the outcome of the credential verification are specified on the System branch. The User functions in this example, however, do not exhibit any structured logic for User behavior. All User functions instead simply serve as source or sink for information interactions with the System. The main limitation of this approach is that only a limited set of use cases can be generated from it since the User behavior is "hard coded" to respond the same way each time the model is executed.

A revision to this model takes a slightly different approach, compressing the User functions onto one main branch and placing C.3 Access Services and C.4 Process Access Failure as alternative exit conditions, since only one of these functions would be selected depending on whether the supplied credentials are valid or invalid (Figure 14.8).

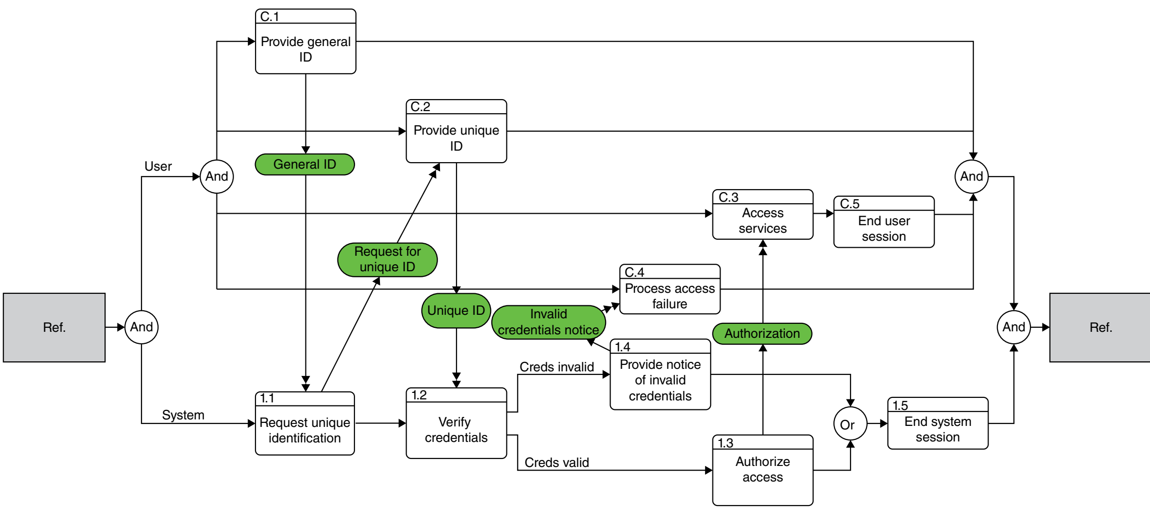


Figure 14.7 An example of EFFBD depicting an authentication behavior model that excludes dependencies on the behavior of an external system (the User).

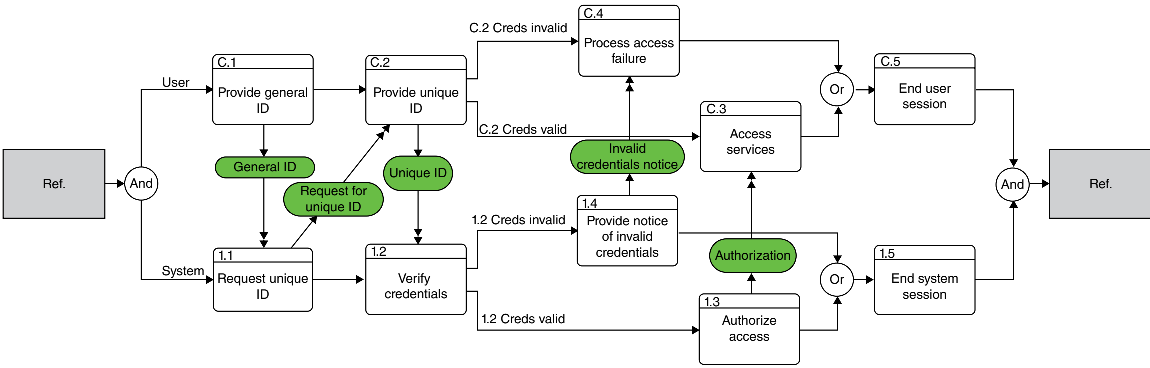


Figure 14.8 An example of EFFBD that includes a description of behavior for both systems in the authentication scenario.

In the revised model, two of the User activities are related to exit condition selections, consistent with corresponding activities on the System branch. If, after providing a unique ID, the System determines the credentials to be valid, the User receives authorization to Access Services. If, on the other hand, the System determines the credentials to be invalid, the User receives notice of this to Process Access Failure. To implement this approach correctly in simulation, a specification must be added to coordinate the branch selections, such that when exit condition "1.2 creds valid" is selected, "C.2 creds valid" should always occur and likewise for the case of invalid credentials. Without such a specification, a simulator has no way to know that this is a requirement and selects exit conditions on different branches at random.

To incorporate additional possible use cases, the EFFBD model could continue to be expanded to encompass more behaviors for both the User and the System. For example, consider the possibility that the User does not respond to the request for unique ID, as tacitly assumed in Figure 14.8. How shall the System behave then? An important consideration is that an EFFBD showing *all* potential behaviors for *all* components interacting in a use case will likely become unwieldy and prone to human error, the larger it grows. This is one likely rationale for the scoping mechanism employed in Figure 14.7.

MP is an approach that resolves this conundrum by employing a divide-and-conquer strategy involving the creation of a *separate* behavior model for each component and specifying interactions between the components as a *separate* concern from that of the behavior of each component. This concept would be akin to creating a separate EFFBD or SysML activity model for each component, allowing elaboration on each component's behavior without concern about adding clutter to a diagram already busy with multiple components and their interactions. This approach allows an architect to focus on describing behavior for one component at a time and then separately specify the general rules (interaction patterns appearing in *all* use case instances) for interaction among components. These component interactions (e.g., the general ID trigger the request for unique ID) may be captured abstractly in a specification of general interaction rules that apply in many similar use cases. Such a specification is what is missing from contemporary Systems Engineering approaches, notations, and frameworks. By separating these concerns, the component behaviors and interactions can be woven together during model execution, automatically generating use cases from the separate behavior and interaction specifications, thereby achieving increased coverage of predictable component interaction.

In MP event grammar, the authentication scenario is described as follows. Each component's behavior is specified separately as a root event in the left-hand part. For example, root events (lines 01 and 08) specify the behaviors of the User and the System, correspondingly. The User's behavior is described in lines 02–07:

```
01  ROOT User:
02  (* request_access
03  (* creds_invalid request_access *)
04  (creds_valid (run_services | abandon_access_request)|
05  creds_invalid (attempt_exhausted |
    abandon_access_request))
06  end_User_session *);
07  request_access: provide_general_ID provide_unique_ID;
```

First, the user requests access (line 02). If the credentials are invalid, the user repeats the request for access (line 03). Line 04 specifies what the user does when credentials are valid: the user may run services having been granted authorization or may abandon the access request for some reason (e.g., experiences an interruption). Line 05 specifies more events that can occur when credentials are invalid: the number of allowable attempts may be exhausted (the number of access attempts is constrained in the systems model), or perhaps the user may abandon the access request. The User session ends (line 06) at the conclusion of event traces for both valid and invalid credentials. In line 07, request_access is decomposed into provide_general_ID followed by provide_unique_ID, to demonstrate the ability to create a hierarchy of events similar to a hierarchy of functions.

The System's behavior is specified in lines 09–17:

```
08  ROOT System:
09  (* request_unique_ID
10  [ creds_invalid request_unique_ID
11  [ creds_invalid request_unique_ID
12  [ creds_invalid attempt_exhausted
13  invalid_creds_notice cancel_access_request]]]
14  [(creds_valid ( authorize_access run_services |
15  long_wait_for_User cancel_access_request ) |
16  creds_invalid long_wait_for_User cancel_access_request)]
17  end_System_session *);
```

The first event in the System for this authentication scenario is request_unique_ID (line 09). If invalid credentials are supplied, the System requests the unique ID up to two more times (lines 10–11). If invalid credentials are supplied for a third time, the number of attempts is exhausted (line 12), and the System provides an invalid credentials notice and cancels the access request (line 13). If valid credentials are supplied, then the System may authorize access and run services (line 14) *or* cancel the access request after a long time elapses while the System is waiting for input (line 15). Yet another alternative is that if invalid credentials are supplied, then there is a long wait for User input; in that case also, the System will cancel the access request (line 16). Regardless of the presence or absence of valid or invalid credentials, the system will always end the session (line 17).

Note that each of these models describes events independent of interactions between the User and the System. The separation of concerns about component behavior and component interaction allows the development of detailed algorithms for every component in the environment and furthermore allows the clean specification of access attempt repetition.

The concept of **abstract** interaction specification is a crucial missing link in current notations and frameworks. As seen in Figures 14.7 and 14.8, systems interactions are often manually embedded in specific use cases or instances of behavior by hard-coding sequenced interactions through multiple components on the same diagram. Many use cases are slight variations of another (such as an authentication scenario resulting in success or failure), so changes to the decomposition or sequence of activities in one use case thread may trigger changes in all affected threads. For example, one may wish to specify that in **any** authorization scenario, the general ID from the User always precedes a request for a unique ID from the System. In MP, this is accomplished using the **COORDINATE** composition operation:

```
18  COORDINATE (* $x: provide_general_ID *) FROM User,
19  (* $y: request_unique_ID *) FROM System
20  ADD $x PRECEDES $y;
```

This composition operation adds the **PRECEDES** relation between selected **provide_general_ID** and **request_unique_ID** events. The first part of composition operation uses event patterns to specify segments of root traces that should be selected. The **(* $x: provide_general_ID *)** pattern in line 18 identifies the sequence of totally ordered **provide_general_ID** events (with respect to the transitive closure of the **PRECEDES** relation). Use of the **(* P *)** pattern for selection means that all events P should be ordered, both iterations should have the same number of selected elements (**provide_general_ID** events from the first trace and **request_unique_ID** events (line 19) from the second), and the pair selection follows this ordering (synchronous coordination). Labels **$x** and **$y** provide access to the events selected within each iteration. The **ADD** composition in line 20 completes the behavior adjustment, specifying that an ordering relation will be imposed on each pair of selected events.

Likewise, one can state that the request for a unique ID from the System always precedes the providing of the unique ID from the User:

```
21  COORDINATE (* $x: request_unique_ID *) FROM System,
22  (* $y: provide_unique_ID *) FROM User
23  ADD $x PRECEDES $y;
```

Note that both the User and System behavior algorithms have event names in common. A constraint must be written to explicitly state that the User and the System share all instances of those events when they occur. For example, there should be no event traces in which credentials are valid from the User perspective but not from the System perspective—such a trace would be invalid. The **SHARE ALL** composition ensures that the schema admits only event traces where corresponding event sharing is implemented:

```
24  User, System SHARE ALL creds_valid, creds_invalid,
25  attempt_exhausted, run_services;
```

Event sharing is in fact yet another way of behavior coordination. Shared events may appear in the root event at any level of nesting.

MP is an executable architecture modeling framework. Event traces (use cases or examples of behavior) can be generated by automated tools from the MP models. Events may be visualized as boxes, and dependencies between pairs of events as arrows marked by the relation type (Figures 14.9, 14.10, and 14.11). Each **PRECEDES** relation may correspond to a control flow or trigger commonly used in flow-oriented notations (e.g., Figure 14.8). Architecture views can also be extracted from MP schemas for different stakeholders to answer typical questions. The root behavior may be visualized with UML activity diagrams (see Figure 14.6). An MP developer's environment may have a library of predefined views providing different visualizations for schemas.

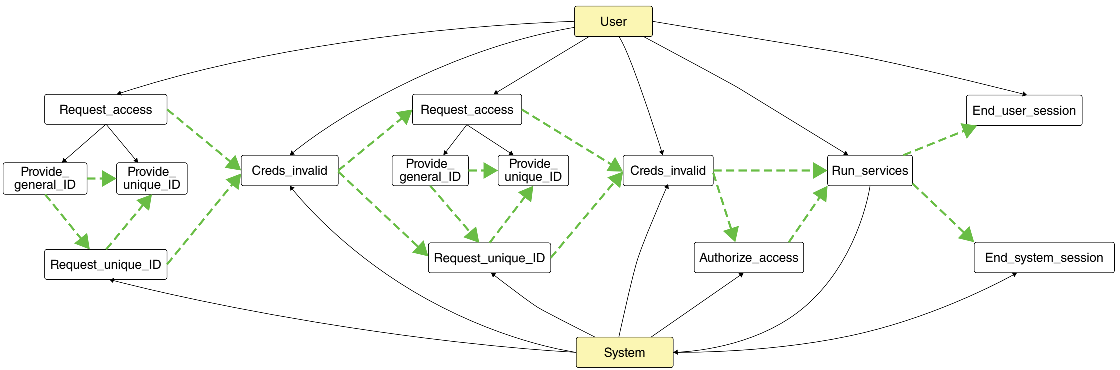


Figure 14.9 An example of event trace (use case) where the User gets access to the System after one unsuccessful attempt. Solid arrows denote IN relations, and dashed arrows depict PRECEDES relations.

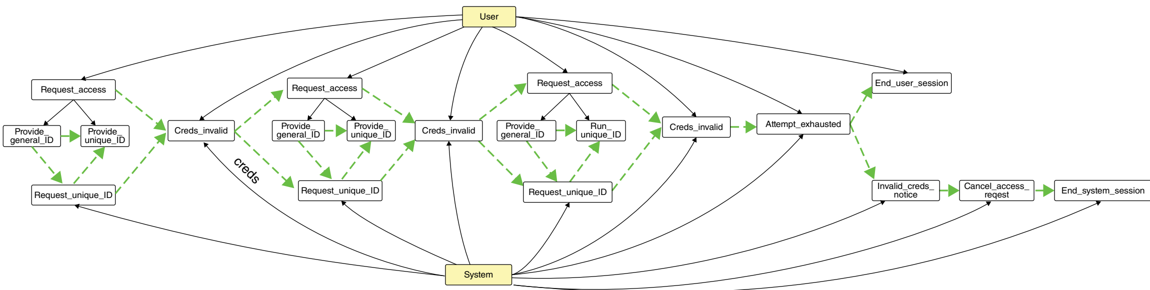


Figure 14.10 An example of event trace (use case) where the User is denied access after three unsuccessful attempts.

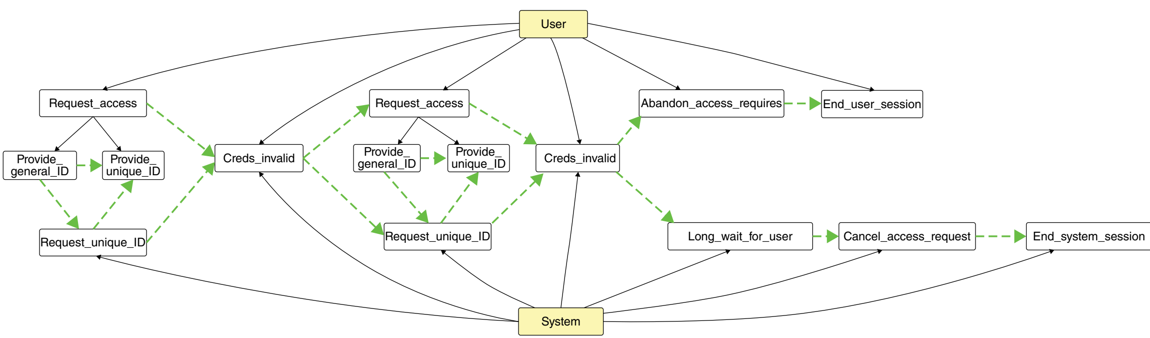


Figure 14.11 An example of event trace (use case) where the User abandons the access request after two unsuccessful attempts.

14.9 ASSERTIONS AND QUERIES

An event trace represents an example of particular execution of the system (or use case, especially if the behavior of the environment is included) that can evolve from the architecture specified by a schema. Event traces can be effectively generated from the event grammar rules and then adjusted and filtered according to the composition operations in the schema. This justifies the term **executable architecture model** for MP. It is possible to obtain all valid event traces within a certain limit. Usually, such a limit (*scope*) may be set by the maximum total number of events within the trace or by the upper limit on the number of iterations in grammar rules (recursion in the grammar rules can be limited in similar ways). For many purposes, a modest limit of a maximum three iterations will be sufficient. This process of generating and inspecting event traces for the schema is similar to the traditional software testing process.

In the case of MP models, it is possible to automatically generate all event traces within the given scope (exhaustive testing). Careful inspection of generated traces (scenarios/use cases) may help developers identify undesired behaviors. Usually, it is easier to evaluate an example of behavior (particular event trace) than the generic description of all behaviors (the schema). The small scope hypothesis (Jackson, 2006) states that most errors can be demonstrated on relatively small counterexamples.

Certain properties of behavior can be formalized as assertions about traces (similar to the **SATISFIES** constraint in Example 14.4 and Example 14.6) and verified exhaustively for all event traces within the scope, yielding the counterexamples when the assertion is violated. For example, hazard states can be specified as a result of certain interactions between the system and its environment, and then the traces within scope can be searched for a trace that matches the hazard scenario. An example of such assertion checking performed on an MP prototype is given in Auguston and Whitcomb (2010). Since assertion checking is performed on a complete event trace, it becomes possible to refer to events following a given event, for example, to specify fairness conditions. This brings the expressiveness of MP assertions closer to temporal logic (Pnueli, 1981).

In a similar fashion, queries can be performed on the traces, providing different kinds of statistics. For example, events may have **attributes**, such as estimated duration, and system's performance estimates can be obtained from collecting a representative amount of event traces and calculating durations for event sets of interest.

Another example of an event attribute may be the probability of an event in alternatives, like **(A [0.3] | B [0.7])** establishing that **A** happens with the probability of 0.3 and **B** with probability of 0.7. Now, it becomes possible to estimate probabilities of certain event traces, for example, probability for the system to get into a hazard state. This opens a whole direction for systems simulation and statistical experiments based on executable systems architecture models and their environment models.

Using MP to automatically generate use cases from component behavior models and abstract interaction specifications, a much larger set of systems behaviors can be predicted. Inspection can be used to expose design errors early in the life cycle by examining each generated use case for logic flaws or undesirable sequences of events. The maximum benefits are gained with assertion checking. The small scope hypothesis provides that the scope of use case generation may be limited by simulating only a specified number of loop iterations for every event trace. MP leverages the small scope hypothesis to provide a solution to expose far more design errors than do current approaches alone,

without requiring specialized skills. If an assertion results in a counterexample (an event trace that contradicts the assertion), it can be used to observe precisely why the assertion is false and, if needed, help the architect write a constraint to prevent the sequence of events that makes the assertion false. MP consequently provides a means for observing and correcting design errors in a modeled architecture, so that an architect can weed undesired behavior from the specification through the addition of abstract SoS interaction constraints.

14.10 IMPLEMENTATION PROTOTYPES

The first MP prototype (Auguston and Whitcomb, 2010) has been implemented as a compiler generating an Alloy model (Jackson, 2006) from the MP schema and then running the Alloy Analyzer to obtain event traces and to perform assertion checks. It has benefited from Alloy's relational logic formalism and visualization tools. Performance depends on the performance of SAT solver used by the Alloy Analyzer.

Direct trace generation from the event grammar can be accomplished quite efficiently, and the process of generating all traces for the given schema and within a given scope can be roughly described by the following procedure:

1. Generate all possible traces within the given scope for each root in the schema.
2. Select one trace from each root's collection. Apply all the schema's composition operations and filters. If the resulting composed trace is consistent with the schema's filters and composition operations, it is included into the schema trace collection. Otherwise, proceed with the next selection.

This process may lead to an exponential explosion, but it has potential for optimization by applying early pruning whenever possible. The main optimization ideas stem from the considerations that composition operations (**COORDINATE** and **SHARE ALL**) usually require an equal number of selected events in the matching traces. Root traces can be sorted according to the number of required events to avoid selection of inconsistent root traces in step 2. Careful rearrangement of composition operations and filters may also provide a significant speed up in the trace assembly.

Other examples using this technique and an online demo of MP automated tools are found in Rivera (2010) and Rivera Consulting Group (2013), respectively. A prototype trace generator has been built to convert MP schemas into a C++ code and then compile and run it. This architecture solution is similar to the one that has been implemented, for instance, in the SPIN/PROMELA model checker (using C as a target language) (Holzmann, 2004).

Several optimizations similar to the one mentioned earlier have been implemented. A sample run on an iMac with 2.8 GHz/4 GB yields the following performance for a schema example with approximately 60 lines of MP source text, 31 event types including 9 roots, 10 composite event types, 12 atomic event types, and 12 **SHARE ALL** compositions and for a maximum scope of 3 for iterations (actually it is an architecture model for the MP → C++ prototype itself):

- Total of 1,328 traces generated, with total of 79,836 events, average of 60.1175 events/trace, and max trace length of 69
- Initial search space (number of all root trace selections before filtering) of 35,100

- Selection ratio of 3.78348% and generation speed of 18021.8 events/s
- Elapsed time (including compilation of the generated C++ code) of 4.42997 s

14.11 CHAPTER SUMMARY

The MP executable architecture models provide a high level of abstraction for testing, verifying, and documenting systems architecture early in the conceptualization and design phases. The main advantages may be summarized as follows:

- The use of MP focuses the attention of developers early on the behavior of the system and provides tools to verify the assumptions.
- The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and application of automated tools for consistency checks.
- The executable architecture models integrated with the environment behavior models can be helpful for identifying emerging behaviors.
- The ability to generate use cases for requirements specification and for testing the system's implementation.
- The ability to create abstract views on the interfaces, composition, and coordination within the system.
- The ability to develop performance estimates based on statistics obtained from the event traces.
- The possibility to extract different architecture views, for example, based on stakeholder viewpoints, from the architecture model.

MP provides a uniform way to extract use cases from a single architecture model composed of component behavior algorithms and an abstract interaction specification—the latter being a capability that is absent from current Systems Engineering approaches. Use cases are based on generic descriptions of systems behavior, rather than on a limited number of use cases. This approach allows architects to expand their definitions of a "representative" set of use cases to increase the design space explored early in the life cycle and to correct undesired behaviors prior to the implementation. It also transfers the burden of maintaining consistency among similar use cases to automated tools.

Architecture modeling touches on the very fundamental issues in Systems Engineering and software design processes and has substantial consequences for the next phases in software systems design in particular. There are many threads of future research based on the ideas described earlier:

- Monitoring whether the behavior of an implemented system matches the MP architecture model (testing automation). If the source code of implementation can be marked up to indicate which segments of code start and end corresponding MP events, it becomes possible to log actual execution traces and to check them for consistency with expected behaviors.
- Developing methods and techniques for early performance, throughput, and latency estimates based on duration and frequency estimates for events within components and connectors.

- Developing methods and techniques for an architecture model's static analysis, for example, by verifying MP models with a model checking tool (Zhang et al., 2012).
- Introducing architecture metrics for MP models for systems cost estimates.
- Developing a library of reusable architecture patterns.
- Developing a library of reusable architecture views.
- Developing a collection of reusable environment behavior models, including business process models in MP.
- Extending the MP approach to the meta-architecture level to support software product lines and domain-specific architectures by representing the variation points as macroconditions in schemas. The same mechanism may be used for architecture configuration management.

Because of its high abstraction level, application of the MP approach should not be considered limited to the improvement of human-designed software intensive and complex adaptive systems. Design flaws manifesting themselves at inopportune times in these classes of systems were merely the original motivation for developing this approach to behavior modeling. Future research may explore its application to the improvement of human understanding of emergent behavior in economic, biological, and ecological systems and to study the causality of events from patterns in cellular behavior to sustainable food and energy production.

Existing software engineering tools have codified the concepts described herein; the next step is to integrate them into notations and modeling environments used by systems engineers and other professionals concerned with complex technological and/or natural systems. The MP approach is a force multiplier for systems architects that is open for implementation in any academic, government, or commercial modeling tool or environment whose objective involves architecting complex systems.

This chapter described a novel approach for modeling and predicting systems behavior resulting from the interactions among subsystems and among the system and its environment. The approach emphasizes specification of component behavior and component interaction as separate concerns at the architectural level. MP provides a new capability for automatically verifying systems behaviors early in the life cycle, when design flaws are most easily and inexpensively corrected. MP extends existing frameworks and allows multiple visualizations for different stakeholders and has potential for application in multiple domains.

REFERENCES

Abowd, G., Allen, R., Garlan, D., 1995, Formalizing style to understand descriptions of software architecture, *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364.

Aizier, B., Lizy-Destrez, S., Seidner, C., Chapurlat, V., Prun, D., Wippler, J.l., 2012, xFFBD: Towards a formal yet functional modeling language for system designers, In *Proceedings of the 22nd INCOSE International Symposium*. Rome, Italy, July 9–12.

Allen, R., 1997, A formal approach to software architecture, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA. CMU Technical Report CMU-CS-97-144, May 1997.

Allen, R., Garlan, D., 1997, A formal basis for architectural connection, *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

Armstrong, J.R., 2013, Functional architecture's mental roadblocks and other things your mother didn't tell you, In *Proceedings of the 23rd Annual INCOSE International Symposium*, Philadelphia, PA, June 24–27.

Auguston, M., 1991, FORMAN—Program formal annotation language, In *Proceedings of 5th Israel Conference on Computer Systems and Software Engineering, Herclia*, IEEE Computer Society Press, Herclia, Israel, May 27–28, pp.149–154.

Auguston, M., 1995, Program behavior model based on event grammar and its application for debugging automation, In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.

Auguston, M., 2009a, Software architecture built from behavior models, *ACM SIGSOFT Software Engineering Notes*, 34:5.

Auguston, M., 2009b, Monterey phoenix, or how to make software architecture executable, *OOPSLA'09/ Onward Conference, OOPSLA Companion*, Orlando, FL, October 2009, pp. 1031–1038.

Auguston, M., Jeffery, C., Underwood, S., 2002, A framework for automatic debugging, In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, UK, IEEE Computer Society Press, September 23–27, pp. 217–222.

Auguston, M., Michael, B., Shing, M., 2006, Environment behavior models for automation of testing and assessment of system safety, *Information and Software Technology*, 48(10):971–980.

Auguston, M., Whitcomb, C., 2010, System architecture specification based on behavior models, In *Proceedings of the 15th ICCRTS Conference (International Command and Control Research and Technology Symposium*), Santa Monica, CA, June 22–24.

Auguston, M., Whitcomb, C., 2012, *Proceedings of the 24th ICSSEA Conference (International Conference on Software and Systems Engineering and their Applications*), Paris, France, October 23–25.

Bass, L., Clements, P., Kazman, R., 2003, *Software Architecture in Practice*, 2nd Edition, Boston, MA: Addison-Wesley.

Booch, G., Jacobson, I., Rumbaugh, J., 2000, OMG unified modeling language specification, http://www.omg.org/spec/UML/ (accessed October 15, 2014).

Bruegge, B., Hibbard, P., 1983, Generalized path expressions: A high-level debugging mechanism, *The Journal of Systems and Software*, 3:265–276.

Campbell, R.H., Habermann, A.N., 1974, The specification of process synchronization by path expressions, *Lecture Notes in Computer Science*, 16:89–102.

Department of Defense, 2009, *DoD Architecture Framework*, version 2.0, Washington, DC: ASD(NII)/ DoD CIO.

Feiler, P., Gluch, D., Hudak, J., 2009, The architecture analysis & design language (AADL): An introduction, Technical Note CMU/SEI-2006-TN-011, http://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html (Accessed June 2009).

Friedenthal, S., Moore, A., Steiner, R., 2006, OMG systems modeling language (OMG SysML™) tutorial, Presented at the 2006 *INCOSE (International Council on Systems Engineering) International Symposium*, Orlando, FL, July 11.

Harel, D., 1987, A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Hoare, C.A.R., 1985, *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice-Hall.

Holzmann, G., 2004, *The SPIN Model Checker*, Boston, MA: Addison-Wesley.

ISO, 2011, International Organization for Standardization. ISO Standard ISO/IEC 42010:2011, *Systems and Software Engineering—Recommended Practice for Architectural Description of Software-Intensive Systems*.

Jackson, D., 2006, *Software Abstractions: Logic, Language, and Analysis*, Cambridge, MA: The MIT Press.

Jackson, M., 2007, Consultancy and research in software development. Past research topics. http://mcs.open.ac.uk/mj665/topics.html (Accessed October 15, 2014).

Knuth, D., 1984, Literate programming, *The Computer Journal*, 27(2):97–111.

Kruchten, P., 1995, Architectural blueprints—The 4 + 1 view model of software architecture, *IEEE Software*, 12(6):42–45.

Liskov, B., Zilles, S., 1974, Programming with abstract data types, *ACM SIGPLAN Notices*, 9(4):50–59.

Long, D., Scott, Z., 2011, *A Primer for Model-Based Systems Engineering*, 2nd Edition, lulu.com.

Long, J.E., 2000, Relationships between common graphical representations used in system engineering, In *Proceedings of the SETE2000 Conference (Systems Engineering and Test and Evaluation)*, Brisbane, Queensland, November 15–17.

Luckham, D., Augustin, L., Kenney, J., Vera, J., Bryan, D., Mann, W., 1995, Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355.

Luckham, D., Vera, J., 1995, An event-based architecture definition language, *IEEE Transactions on Software Engineering*, 21(9):717–734.

Maier, M., Rechtin, E., 2000, *The Art of Systems Architecting*, Boca Raton, FL: CRC Press.

NASA, 2007, *Systems Engineering Handbook*, Washington, DC: NASA/SP-2007-6105 Rev1.

Nassi, I., Shneiderman, B., 1973, Flowchart techniques for structured programming, *ACM SIGPLAN Notices XII*, pp. 12–26.

Object Management Group, 2012, *Systems Modeling Language Specification*, version 1.3, http://www.omg.org/spec/SysML/ (Accessed July 22, 2014).

Pelliccione, P., Inverardi, P., Muccini, H., 2009, CHARMY: A framework for designing and verifying architectural specifications, *IEEE Transactions on Software Engineering*, 35(3):325–346.

Perry, D., Wolf, A., 1992, Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52.

Pnueli, A., 1981, A temporal logic of programs, *Theoretical Computer Science*, 13:45–60.

Rechtin, E., 1991, *Systems Architecting: Creating and Building Complex Systems*, Englewood Cliffs, NJ: Prentice Hall.

Rivera, J., 2010, Software system architecture modeling methodology for Naval Gun Weapon Systems, Doctoral Thesis, Naval Postgraduate School, Monterey, CA, December 2010.

Rivera Consulting Group, 2013, *Eagle 6 Modeling*, http://eagle6modeling.riverainc.com/ (Accessed July 22, 2014).

Roscoe, B., 1997, *The Theory and Practice of Concurrency*, London, UK: Prentice Hall International Series in Computer Science, pp. 580.

Rozanski, N., Woods, E., 2012, *Software Systems Architecture*, 2nd Edition, Upper Saddle River, NJ: Addison-Wesley.

Spivey, J.M., 1989, *The Z Notation: A Reference Manual*, 2nd Edition, Englewood Cliffs, NJ: Prentice Hall International Series in Computer Science, pp. 1992.

Taylor, R., Medvidovic, N., Dashofy, E., 2010, *Software Architecture, Foundations, Theory, and Practice*, Hoboken, NJ: John Wiley & Sons, Inc.

Vaneman, W., Jaskot, R., 2013, A criteria-based framework for establishing system of systems governance, In *Proceedings of the 7th Annual International IEEE Systems Conference*, Orlando, FL, April 15–18, pp. 491–496.

Wang, Y., Parnas, D., 1994, Simulating the behavior of software modules by trace rewriting, *IEEE Transactions on Software Engineering*, 20(10):750–759.

Zhang, J.Y., Liu, M., Auguston, J.S., Dong, J.S., 2012, Using monterey phoenix to formalize and verify system architectures, In *19th Asia-Pacific Software Engineering Conference APSEC 2012*, Hong Kong, December 4–7.

View publication stats