

Chapter 2

Behavior Modeling for Interaction

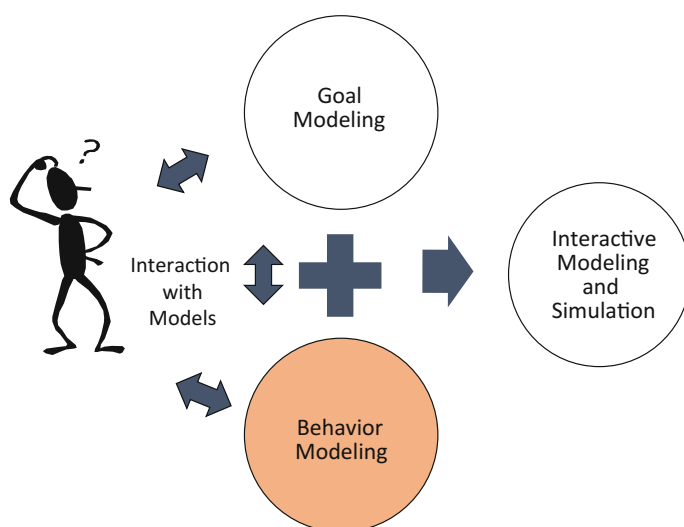


Fig. 2.1 Focus on behavior modeling

In the introduction, we named two foundations of Interactive Modeling and Simulation, namely, two comparable semantics for behavior modeling and goal modeling.

This chapter presents a system's behavior modeling semantics suitable for Interactive Modeling and Simulation (Fig. 2.1). In order to explain our choice, we begin with an overview of behavior modeling semantics, and then we describe the unique semantic elements of Protocol Modeling making it one of the foundations of Interactive Modeling and Simulation.

This chapter accumulates all the theory needed to understand details of the next chapters. The readers will get better understanding if they first read this chapter and return to it while reading and doing the exercises of other chapters. Reading this chapter will prevent many “why” questions.

2.1 Semantic Predecessors

The semantic elements of behavior modeling form the cornerstones of computer science. The names of these semantic elements are widely known, but the influence of the semantic variations on the expressivity of behavior modeling is rarely discussed.

In this chapter, we demonstrate the variations of behavior modeling semantics. Our choice of semantics for interactive modeling is easier to understand knowing the predecessor semantics that lay out the grounds and the common concepts.

2.1.1 Finite State Machine

A finite state machine (FSM) [13] is universally seen as a predecessor of all behavior modeling semantics. A finite state machine is a tuple of three sets

$$FSM = (S, E, T), \text{ where}$$

- S is a finite set of states. The initial state $s_0 \in S$ and the set of final states $F \in S$ are optionally specified in the set of states. A machine is situated in one state $s \in S$ at a time.
- E is a finite set of recognized events. Instances of events $e \in E$ are often called actions.
- T is a finite set of transitions. Each transition is a pair of states labeled with an event. Any state can be an input state or an output state of a transition or both. A set of transitions is a subset of the Cartesian Product of three sets: states, events, and states

$$T \subseteq S \times E \times S; t \in T; t = (s_i, e, s_j), s_i, s_j \in S.$$

A Finite State Machine of the Google Screens

An example of a Finite State Machine is shown in Fig. 2.2. The model represents the familiar Google Screens (simplified) that we all use every day for the Internet search.

The closed state of our browser is the initial state. The initial state is depicted as a dark oval. The other states depicted as ovals are: *Google Screen*, *Found Links*, *Lucky Links*, *Choose an Account*, and *Closed*.

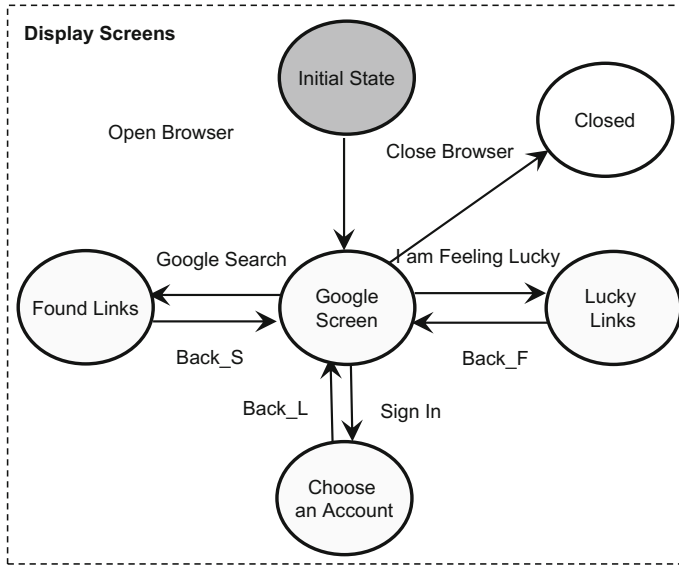


Fig. 2.2 Google Screens presented as a finite state machine

The transitions are presented with arcs from one state to another. The arcs are labeled with events. The set of events includes $\{Open\ Browser, Close\ Browser, Google\ Search, I\ am\ Feeling\ Lucky, Sign\ In, Back_S, Back_F, Back_L\}$.

The FSM *Google Screens* shows that, from the *Initial State*, the machine may accept an instance of event *Open Browser* and transit to the state *Google Screen*. The possible transitions from the state *Google Screen* are

- (*Google Screen*, *Google Search*, *Found Links*)
- (*Google Screen*, *I am Feeling Lucky*, *Lucky Links*)
- (*Google Screen*, *Sign In*, *Choose an Account*)
- (*Google Screen*, *Close Browser*, *Closed*)

There are also transitions directing back from the states *Found Links*, *Lucky Links*, and *Choose an Account* to the *Google Screen*.

- (*Found Links*, *Back_S*, *Google Screen*)
- (*Lucky Links*, *Back_F*, *Google Screen*)
- (*Choose an Account*, *Back_L*, *Google Screen*).

Basic Rules for Composition of Transitions

Here, we refer to the composition of transitions into processes (behaviors) and, further, to the composition of processes (behaviors) into other processes (behaviors).

The basic composition rules are studied by process algebras. They formulate the rules of composition that need to be applied in order to compose transitions into processes. The transitions can follow each other, alternate, or be concurrent.

In order to build processes and manipulate processes in an algebraic way, process algebras use formulas. Each formula may be visualized as a finite state machine.

- The basic element of formulas in process algebra is an event (an action).
- A state is a beginning or an ending of an event.
- Any event is identified by a small letter: a , b , or by its name, for example, “*Open Browser*”.
- It is assumed that any event can execute itself and terminate.
- A sequence of events in a process is expressed using the multiplication $*$ symbol, for example, $a * b$.
- An alternative choice between two actions is expressed with the plus symbol: $a + b$.
- A repetition of a process is shown with the star symbol a^* .
- Round brackets $()$ are used to combine event expressions.
- Symbol \surd expresses a process termination, $\surd \notin \{a, b, c, \dots\}$.
- Symbol ε represents an empty process, $\varepsilon \notin \{a, b, c, \dots\}$.
- Symbol δ is reserved for a deadlock where all processes are waiting for each other, $\delta \notin \{a, b, c, \dots\}$.

The following axioms of the basic process algebra are used for proofs of process properties.

- A1 : $a + b = b + a$;
- A2 : $(a + b) + c = a + (b + c)$;
- A3 : $a + a = a$;
- A4 : $(a + b) * c = a * c + b * c$;
- A5 : $(a * b) * c = a * (b * c)$;
- A6 : $x + \delta = x$;
- A7 : $x * \delta = \delta$;
- A8 : $x * \varepsilon = x$;
- A9 : $\varepsilon * x = x$.

Further elaborating in process algebra, formal axioms and rules is not the aim of this work. The reader interested in this subject may consult [1, 9]. In this book, we use the process algebra axioms to explain the results of composition of behaviors.

For example, the process corresponding Fig. 2.2 can be represented with the following formula:

$$\begin{aligned} & (Open\ Browser * \\ & ((Google\ Search * Back_S) + (I\ am\ feeling\ Lucky * Back_F) + \\ & (Sign\ In * Back_L) + \varepsilon) * \\ & Close\ Browser). \end{aligned}$$

By looking at Fig. 2.2, we may see that the process algebraic events represent transitions.

The sequences, alternatives, and repetition of events are the results of the process algebraic behavior composition. The basic composition rules of the process algebra tell us that

- In a sequence of events, only the first event can execute itself and enable another event or terminate.
For example, in the sequence $(Google\ Search * Back_S)$, only event *Google Search* can execute itself and then enable event *Back_S*.
- An alternative does not influence the execution and termination of the events of the other branch.
For example, event *Google Search* does not influence event *Sign In*.

The execution of the process corresponding Fig. 2.2 can result in different traces (sequential processes)

$$(Open\ Browser * \varepsilon * Close\ Browser) = (Open\ Browser * Close\ Browser);$$

$$(Open\ Browser * (Google\ Search * Back_S) * Close\ Browser);$$

$$(Open\ Browser * (Google\ Search * Back_S) * (I\ am\ feeling\ Lucky * Back_F) * Close\ Browser).$$

Finite State Machines Can Be Deterministic and Nondeterministic

Each state of a deterministic FSM has exactly one transition labeled with a possible event. The FSM in Fig. 2.2 is deterministic.

From a state of a nondeterministic FSM can be more than one transition labeled with the same event.

The concept of a finite state machine has a wide area of applications. The notions of states, events, and transitions appear in any behavior model. However, it is difficult to see states and events of finite state machines as elements of interaction. They are just labels. Especially for modeling of interaction, the FSM model has some expressivity limitations.

Expressivity Limitations of FSM:

1. *The notions of events and transitions do not capture the information transfer (transfer of data values) during interactions;*
2. *An FSM is able to remember only its state. An FSM cannot express the details of states important for interpretation of the states by humans;*
3. *The size of an FSM is restricted by the cognitive abilities of humans. They are usually able to understand only a part of a system: an object or a particular aspect of system behavior;*
4. *The composition of FMSs (of different objects and aspects) into the system model does not belong to the semantics of FSM.*

Finite state machines can be seen as a common terminological ground for many other behavior modeling approaches extending their semantics. The state machines with modified and extended semantics are often called Labeled Transition Systems (LTS) [9].

2.1.2 Holistic Approaches

One of the semantic extensions is the distribution of the state of the model in space. If the state of a system is distributed in space, one state can be represented with a tuple of substates often called places. Such a state semantics is the basis of holistic approaches to behavior modeling. This family of holistic approaches includes Petri Nets [25], the UML Activity diagrams [24], Colored Petri Nets [12], and Business Process Model and Notation (BPMN) [23].

As a result of the distributed state, a holistic model is capable of capturing the behavior of the whole system; it shows a map of the system. This is definitely an added value of this semantic extension.

A Petri Net by Example

Figure 2.3 shows an example of a Petri Net of the Google Screens.

- All the places, presenting the states, are shown by ellipses.
- The tokens are situated in the places *Initial State* and *One User*. (We constrain the model to represent the behavior of one user.) The tokens are shown by number “1” near each of these places.¹

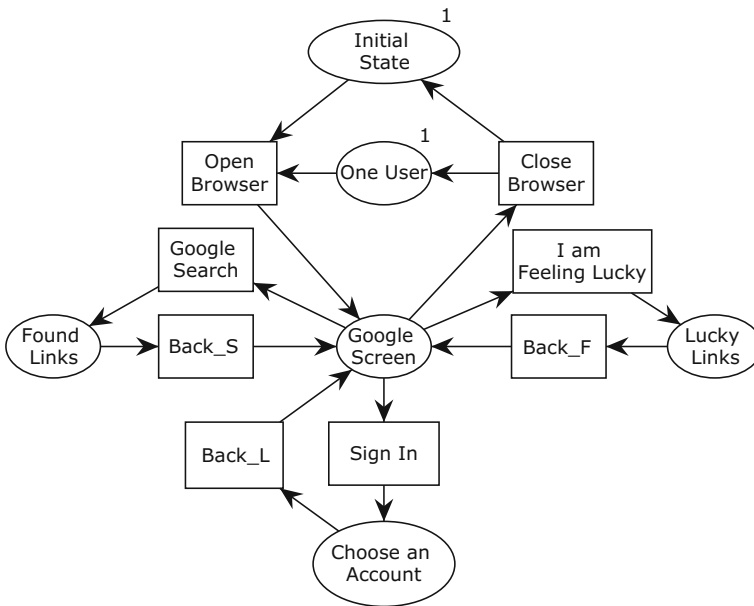


Fig. 2.3 A Petri Net of the Google Screens

¹We use the CPN tools [7] to produce the figures of Petri Nets.

- The transitions are depicted as boxes. In this model, transitions are labeled with events. The set of transitions are listed below

{Open Browser, Close Browser, Google Search,

I am Feeling Lucky, Sign In, Back_L, Back_S, Back_F}.

Let us show, how the distributed state is used to capture different aspects or concerns of the system in one model. To illustrate this, let us extend the Petri Net of Google Screens with a security aspect and an aspect of email access.

A Petri Net of the Google Screens with a Security Aspect

Figure 2.4 adds the following security aspect to the previous model in Fig. 2.3.

- When the user is in the *Initial State* of your browser, he is also in the place *My Account Logged Out*. Both places contain tokens (shown by number “1” near each of the places). So, there are three places that contain tokens.
- In order to sign into a user’s Google account, a user chooses event *SignIn*. Transition *SignIn* is enabled, if the user is logged out (place *My Account Logged Out* contains a token and place *Google Screen* contains a token).
- After firing of the transition *SignIn*, place *unsigned: Password Request* gets a token. Transition *Login* becomes enabled. The user needs to type a user name and the corresponding password.²
 - If the combination of the user name and the password is wrong, then the transition *Refuse* may take place. When transition *Refuse* fires, the token goes to the place *unsigned: Password Request*. Transitions *Login* and *Back_L* are enabled.
 - If the combination of the user name and the password is correct, then the place *My Account Logged On* gets a token and the user sees the place *Google Screen*. From this state one may use transition *toMyAccount*. If this transition fires, net comes to place *signed: Screen to Sign Out*, where one may *Sign Out*.

We invite our reader to replay this model and validate if it captures the behavior of the Internet browser using *www.google.com*.

A Petri Net of the Google Screens with a Security Aspect and an Email Access

Let us add another email access aspect to our Petri Net model. Figure 2.5 adds the possibility to read emails from Google Displays. In the distributed state when each

²Google keeps the user names and the passwords. Our model does not capture this information, but it is possible to capture it in Colored Petri Nets.

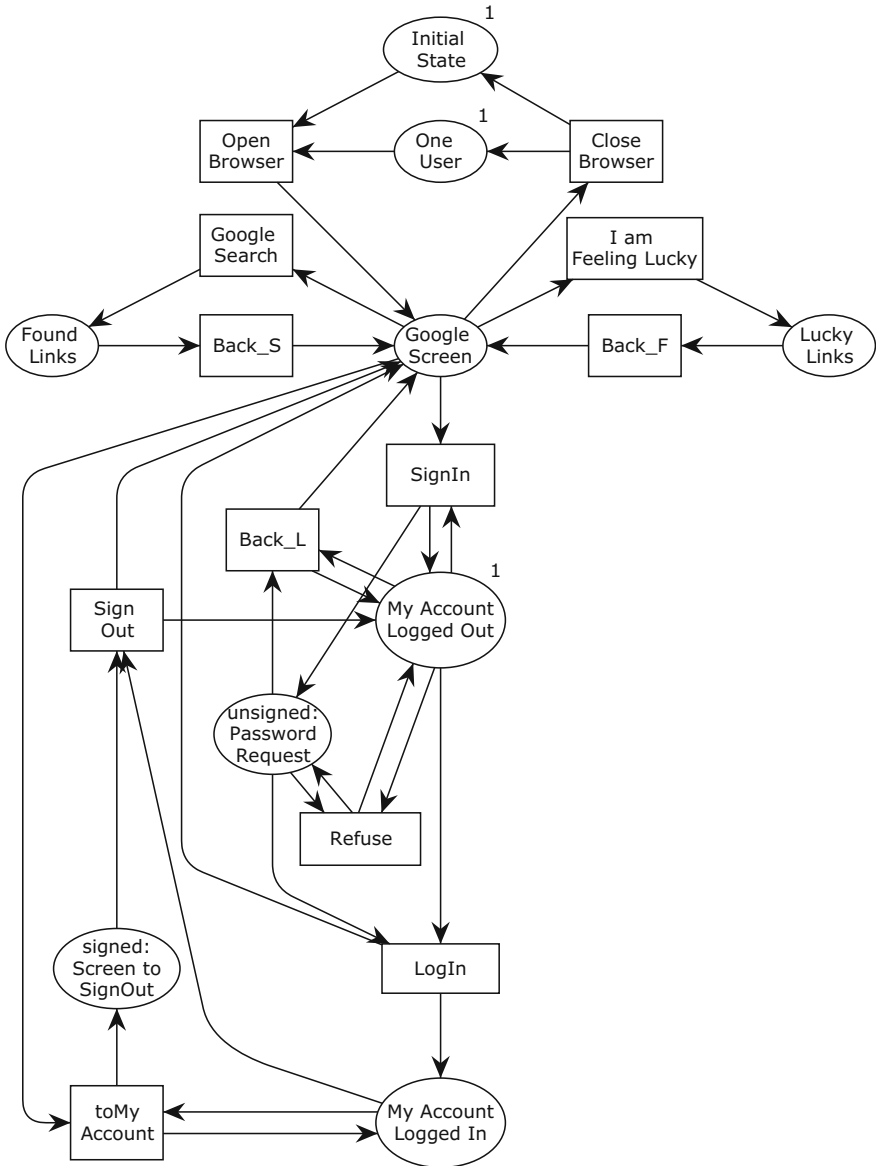


Fig. 2.4 A Petri Net of the Google Screens with a security aspect

of places *My Account Logged In* and *Google Screen* has a token, the transition *Gmail* is enabled. The firing of this transition produces a token to the state *Email Browser* and the user can *read a letter* and then *close* it in a cycle. The user can also return to the state *My Account Logged In* and *Google Screen* with transition *Back_G*.

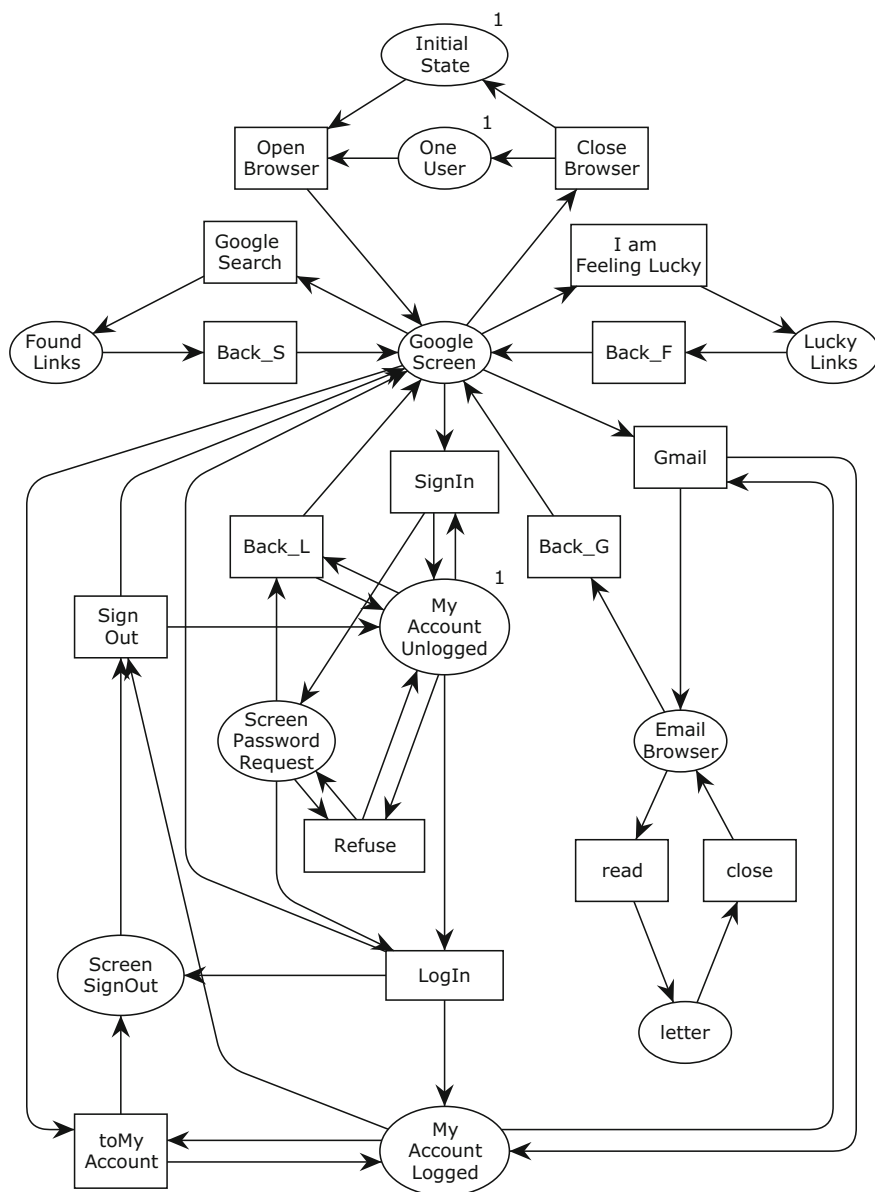


Fig. 2.5 A Petri Net of the Google Screens with a security aspect and a email access

Limitations of Holistic Approaches for Interactive Modeling

The family of holistic behavior approaches has many successful application domains. Colored Petri Nets [12] overcome two first semantic limitations mentioned for finite

state machines. Namely, they use data types to model events and extend the memory of the model with internal variables of different types. Colored Petri Nets also use hierarchical composition allowing to replace a transition with a new Colored Petri Net. Such a transition is similar to a hyperlink on a web page. However, holistic approaches do not meet some of the requirements of interactive modeling.

No Separation of Objects and Concerns

Our examples in Figs. 2.4 and 2.5 show that holistic models capture many aspects of system behavior, but do not separate the behaviors of system objects and concerns. The holistic models use the basic composition rules and the hierarchy to compose processes. These composition techniques do not allow for separating requirements and for preservation of their separation in the model. The reused fragments should be copied and often corrected. Repeating and copying is always a source of model mistakes.³

Large Size

Figures 2.3, 2.4 and 2.5 illustrate the growth of a holistic model. For a real-size system, a holistic model is often too large to be easily observed and understood. The objects and concerns are woven one into another. So, the inability to separate objects and concerns is one of the semantic features that hinder interaction with the model. The difficulty is comparable with the difficulties of reading maps with unknown keys/legends.

Nondeterministic Semantics

Another limitation of holistic approaches is the nondeterministic semantics. In terms of interactions, the holistic approaches allow for collecting questions in buffers (as tokens in Petri Nets in places) and then providing the answers (consuming tokens) randomly, choosing the questions from buffers. Such a semantics cannot be the basis for interaction. In the interaction, if one asks a question, one expects an answer to his question, not to the question asked by someone else a week ago. Nondeterministic models are good for modeling such problems as the game of Chinese whispers (also known as Broken Telephone), but not for the goal-directed business interactions.

The nondeterministic semantics and the distributed states also make it difficult to interpret holistic models in terms of goals, which are usually formulated in terms of states of objects.

2.1.3 Compositional Approaches

Another family of approaches built on the semantic grounds of FSMs can be named compositional approaches. It includes the UML Behavior State Machines, the UML

³The limitations of holistic approaches for interactive modeling have been studied on several real cases [14, 26, 27].

Protocol State Machines [24], Abstract State Machines [5], Discrete Event System Specifications [22], Protocol Modeling [19], and many others. The compositional approaches use FSMs to present the behavior of classes of objects of the modeled system. In order to model the behavior of the whole system, the approaches have the notions of

- instantiation of objects from the LTSs of their classes and
- object composition (or weaving) rules defining how to combine the traces of object instances into the traces of the system model.

The composition rules are based on different semantics of events, states, and transitions. This difference in semantics of events, states, transitions, and object composition makes the modeling approaches suitable or not suitable for Interactive Modeling and Simulation.

Different Semantics of Events

Abstract Signals Versus Structured Messages

An event may be seen as a named signal coming from the environment. For example, *Open Google*, or *Sign In* are such named signals. In most cases, the named signals are not sufficient for interaction because any interaction usually involves data transferring.

An event needs some reserved space to carry data. There are approaches that define for an event a data structure [11]. An event instance in this case belongs to a certain event type with own data structure.

For example, an event type “*Sign In User XXX with Password YYYYYY*” may sign in an infinite number of users with different user names and passwords.

Inputs and Outputs Versus Inputs only

There are two different ways to model the origin of an event.

The models called *Transducers* (Fig. 2.6) recognize both input and output events. A transducer model consists of a *closed set of communicating subsystems* (Fig. 2.6).

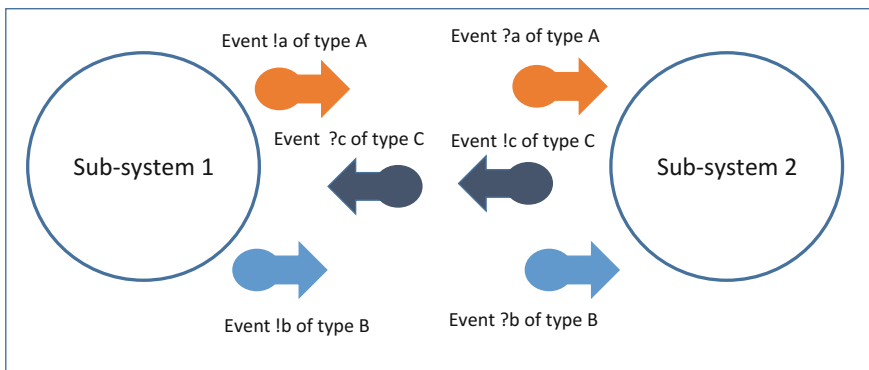
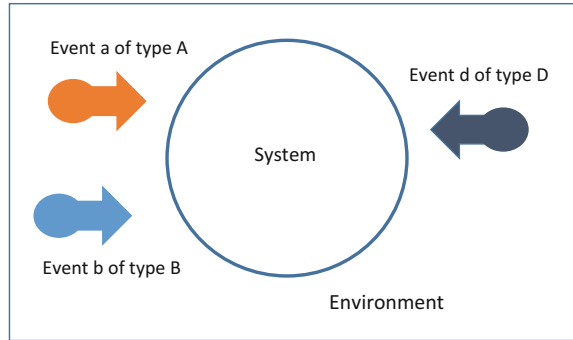


Fig. 2.6 Transducer

Fig. 2.7 Acceptor

There is no notion of environment. A transition can be labeled with both an input event and an output event. Sign $?e$ means an input event e . Sign $!e$ stands for an output event e .

If a transducer specifies a transition $(s_i, !e, s_j)$, then, being in state s_i , it can send event e and transit from state s_i to state s_j . If a transducer has a transition $(s_i, ?e, s_j)$, then, being in state s_i , it can receive event e and transit from state s_i to state s_j .

A transducer model represents message-based communication. Both sides of communication are expressed as events: sent or received. The states of the model fix the facts of sending or receiving and do not provide another information. The modeler can only start the transducer model and observe its progress. The modeler is not able to interact with the model.

The models called *Acceptors* (Fig. 2.7) consider all events only as the system *inputs* coming from its environment. Everything outside an acceptor (Fig. 2.7) model is the environment submitting events. The environment is not included into the model. The only role of the environment is to submit events to the model.

If an acceptor specifies a transition (s_i, e, s_j) , then, being in state s_i it can react to event e , submitted by environment. If event e is accepted, the model can transit from state s_i to state s_j .

Acceptors form the basis for Interactive Modeling and Simulation. An event represents one side of interaction. A modeler plays the role of environment and submits events. Another side of the interaction is the result of the system reaction represented by the state of the model. The state should be visible for the modeler involved into interaction.

Semantics of States

Abstract States Versus States with Variables

A state represents a snapshot of a system. This snapshot can be as abstract as just having only a name. The states of a model in Fig. 2.2 are abstract names “*Google Screen*”, “*Lucky Links*”.

Along with its name, a state definition may be expressed with variables of different types called model attributes. Variables may belong to the whole model or to a submodel.

When the model is simulated, the variables get their values depending on event instances and transitions. For example, the model Fig. 2.2 may have a boolean variable “A User is signed” which can be “true” or “false”.

If the abstract event type “Sign In” is replaced with the event type “Sign In User XXX with Password YYYYYY”, then the attribute “Signed User Name: String” may get its value “XXX”. The model can also have an attribute “Saved Password: String”. The value of “Saved Password” may be compared with the password “YYYYYY” in the event instance.

Attributes and their values are the necessary elements of Interactive Modeling and Simulation. Humans interpret the semantics and values of attributes in different states of the model in order to

- evaluate the correspondence of the model behavior and the required behavior (during interactive modeling);
- make decisions about the next steps of the interaction (during the interactive simulation).

Buffers for Events

Among all types of variables that have ever been defined for behavior models, the queues of events or the bags of events have the greatest influence on the model interpretation. Queues and bags in a behavior model are used as storages of events. Together with the events coming from the environment or subsystems, the events in queues introduce race conditions and nondeterministic behavior into the model.

The UML state machines are the examples of such a semantics [24]. If a recognized event arrives when a UML state machine is not ready to handle it, this event is stored. When the machine is ready to handle this event, another event can arrive. Which of those events will be handled is random; it is nondeterministic.

The nondeterministic semantics is confusing for Interactive Modeling and Simulation. A human, interacting with the model, expects a predicted type of system state to be reached. This means that the queues (and bags) of events should not be used for interactive models.

Semantics of Transitions

Can Versus Must

In most behavior modeling approaches, only the “can-transit” semantics of transitions is used, i.e., a machine can transit from one state to another if the input state of a transition is reached.

There is also semantics of “must-transit”, i.e., if the input state of a transition is reached, this transition must fire and the output state must be reached. This semantics is often used for internal actions of the system.

As far as the interaction is concerned, not all systems are adequately modeled with the “can-” and/or “must-” semantics. The additional semantics for motivation for interactions is often needed. The motivation semantics will be explained further in this book in Chap. 7.

Transitions that Update States and Variables

If the states of a behavior model include memory variables, and its events have the corresponding data structures, then a transition may be accompanied with functions that make calculations of new values for the state variables, memory variables, and/or events. In other words, if a behavior model transforms data, carried by events or stored in the states and attributes of the model, a transition has functions that perform these transformations.

A function that corresponds to a transition (s_i, e, s_j) can be associated with each of the three elements of the transition: with the initial state, with the event type and with the final state. For example, the label of a transition in the UML state machines has the following structure *[precondition]/event/[postcondition]*. Both the precondition and postcondition are functions associated with this transition.

A function that corresponds to a transition (s_i, e, s_j) can be associated with this transition as a whole (with all elements of the transition). For example, if an event instance has the type *Sign In User XXX with Password YYYYYY*, a function may search the user name *XXX* in the running model and compare the password *YYYYYY* with the stored password of user *XXX*. If the user name is found, and the password in the event instance is equal to the stored password, the function will assign “*Signed User Name:= XXX*”.

The functions associated with transitions and their elements may accompany any model as program files named after those elements. This form of association between functions and elements of transitions is an asset for interactive modeling.

Composition in the Calculus of Communicating Systems (CCS)

Compositional approaches separate objects. Objects proceed concurrently and use some rules to work together, i.e., objects use a composition technique.

One composition technique is described in the Calculus of Communicating Systems (CCS) by R. Milner [20].

CCS uses a binary composition operator \parallel . Two arguments of this operator are processes (or behaviors)

- process P_1 with the first possible event a : $a * P_1$ and
- process P_2 with the first possible event b : $b * P_2$.

The result of this operator is the process of the CCS parallel composition of initial processes:

$$a * P_1 \parallel b * P_2.$$

The composition can choose to execute the transition labeled with an event a of P_1 or the transition labeled with an event b of P_2 in the interleaving manner, i.e., one after another. If the processes do not interact, they are merged and do not influence one another

$$a * P_1 \parallel b * P_2 = P_1 \parallel b * P_2 \text{ or}$$

$$a * P_1 \parallel b * P_2 = a * P_1 \parallel P_2.$$

Moller [21] proved that the complete finite axiomatisation of this composition needs the left-merge operator. The left-merge operator takes its initial transition from the process P_1 .

If the concurrent processes interact (they only interact one with another), then the send–receive semantics is used for process merging or composition. The interaction takes place if two corresponding events take place: the sender process sends event and the receiver process is in a state to receive this event:

- The sent event is labeled with the exclamation symbol $!e$.
- The received event is labeled with the question mark $?e$.
- The send–receive composition takes place if the sent event with same name is followed by the received event with the same name: $(!e * ?e)$.
- The state of both processes is updated in the case of communication:

$$a P_1 ||| b * P_2 = P_1 ||| P_2; \text{ where } a = !e; b = ?e.$$

Bergstra and Klop [3] introduced a communication merge that combines the send and receive actions with the same name in one action.

The consequence of the CCS parallel composition semantics is the race conditions when two events have been sent to the same receiver. Which of the events will arrive first? This fact depends on the factors that do not belong to the process model: the time needed for the signal transfer, the path chosen to signal transfer, etc.

Composition in the Calculus of Communicating Sequential Processes (CSP)

Another composition technique is the Communicating Sequential Processes (CSP) by A. Hoare [10].

In this calculus, any process is a set of observable sequences of events. An observable sequence of events is also called a trace. For example, a trace $(x * y * z)$ consists of three events, x followed by y , followed by z .

The calculus of Communicating Sequential Processes (CSP) defines an operator for composition of concurrent processes $a * P$ and $b * Q$. This operator uses the synchronization of the event acceptance.

The alphabets of events recognized by the acceptors P and Q are denoted as αP and αQ .

The intersection of sets of events of two processes (alphabets of events) is not empty.

Informally, the operator of CSP-parallel composition can be explained as follows,

- if an event is recognized by only one process, and this process is ready to accept it, the event is proceeded by this process. If the process is not ready to accept the event, the event is refused.
- If an event is recognized by both processes, and both processes are able to accept it, the event is accepted. If at least one of the processes is not able to accept the event, the event is refused.
- If an event is not recognized by both processes, the event is rejected.

There is a semantic difference between “refusing” and “rejection.” The rejection of an event means that event does not belong to the domain of the model. The refusing means that the event is in the domain of the model, but the model is not in the right state to accept it.

The operator of CSP parallel composition was initially defined only for the models with events that do not use data structures and with states that do not use data variables.

The models that use the CSP parallel synchronous composition semantics have such a property as observational consistency, meaning the preservation of traces of parts in the traces of the whole [8, 15].

After A. Hoare, other authors used the terms “abstraction operator” [4] or “hiding operator” [2] in order to explain the “restriction operator.” The abstraction operator can be applied to any symbol of a trace iteratively

- Let $\langle a \rangle$ be a sequence of one symbol a .
- Let A be a set to abstract from it.
- $a \in A \Rightarrow \delta_A(a) = \langle \rangle$
- $a \notin A \Rightarrow \delta_A(a) = \langle a \rangle$.

Using of a restriction operator or a hiding operator is a matter of taste.

In the next chapters of this book, the property of observational consistency will be applied for

- relating requirements and business rules to design models;
- local reasoning on models, i.e., reasoning on parts about behavior of the whole;
- model evolution when the models of different concerns and requirements are added to the model or deleted from the model.

2.2 Protocol Modeling

Protocol Modeling combines the semantic elements and properties that are necessary for Interactive Modeling and Simulation.

Briefly, a protocol model is a set of deterministic acceptors (protocol machines) synchronized by the CSP parallel composition. In order to support interactive simulation and its interpretation, these acceptors contain data structures for events and the state space extended with attributes. We already mentioned that the CSP parallel composition operator was initially defined for the processes without data. The unique feature of protocol models is the extended version of the operator of the CSP parallel composition that includes the composition of behavior models and events with data.

Now, we will give the definition of protocol models. Then, we will show that protocol models with the extended version of the CSP parallel composition possess the property of observational consistency since this is the basis for interactivity of modeling and simulation.

2.2.1 Semantics of Protocol Modeling

Each protocol model splits the universe into a system and its environment. The environment is presented by events submitted to the system. The system may change its state only by reacting to events.

A protocol model PM is a CSP parallel composition of a finite number of *protocol machines*:

$$PM = \parallel PM_i, i \in N.$$

The building blocks of a protocol model [19] are protocol machines and events. They are instances of, correspondingly, *protocol machine types* and *event types*.

An *event type* is a tuple $e = (A^e, CB^e)$, where

- A^e is a finite and non-empty set attributes of the event.
- CB^e is a set of callback functions corresponding to this event. The set can be empty or contain only one function.

All elements of the protocol model PM : the event type instances with their attributes and the protocol machine type instances with their elements, may be the inputs and the outputs of a callback function.

A protocol machine type is a Labeled Transition System extended with attributes and callback functions.

A *Protocol machine type* is a tuple $pm = (S_{pm}, E_{pm}, T_{pm}, A_{pm}, CB_{pm})$

- S_{pm} is a non-empty infinite set of states.
- E_{pm} is a finite set of event types e_{pm} . The set can be empty.
- $T_{pm} \subseteq S_{pm} \times E_{pm} \times S_{pm}$ a finite set of transitions:
 $t = (s_x, e, s_y), s_x, s_y \in S_{pm}, e \in E_{pm}$. The set of transitions can be empty.
- A_{pm} is a finite set of attributes of the specified types. The set can be empty. The standard data types such as *String*, *Integer*, *Currency*, *Date*, etc., and the types of protocol machines can be used for specification of attributes. The attributes are the data containers of a protocol machine.
- CB_{pm} is a callback function of this protocol machine type. The set can be empty or contain only one function. All elements of the protocol model PM : the event type instances with their attributes and the protocol machine type instances with their elements, may be the inputs and the outputs of the callback function.

A protocol model PM is a CSP parallel composition of a finite number of *protocol machines*, but it is also a protocol machine, the set of states of which is the Cartesian Product of states of all composed protocol machines [19]:

$$PM = \parallel_{i=1}^n PM_i = (S, E, T, A, CB).$$

$S = \prod_{i=1}^n S_i$ is the set of states;

$E = \bigcup_{i=1}^n E_i$ is the set of events;

$A = \bigcup_{i=1}^n A_i$ is the set attributes of all machines;

$CB = \bigcup_{i=1}^n CB^i$ is the set of callbacks of all machines.

The set of transitions T of the protocol model is defined by the rules of the CSP parallel composition:

- If an event is not recognized by the protocol model, it is *ignored*.
- If an event is recognized by the protocol model and all protocol machines, recognizing this event, are able to accept it, the event is *enabled*.
- If an event is recognized by the protocol model, but at least one protocol machine, recognizing this event, is not able to accept it, the event is *refused*.

As a result, the composition may contain the union of transitions of composed protocol machines if the sets of the recognized events of protocol machines are disjoint. If the interception of the recognized events is not empty, the transitions labeled with the events from the interception are synchronized in the correspondence with the rules of CSP parallel composition described above.

In order to facilitate reuse, there are two variants of protocol machines: objects and behaviors. Objects reflect the behavior of things existing in the universe: people, machines, phenomena, applications, services, etc. Behaviors reflect concerns, for example, security, garbage collection, initialization, collecting of items, registration, etc. Behaviors cannot be instantiated on their own, but may extend the functionality of objects. In a sense, Behaviors in protocol models are similar to mixins [6] or aspects in programming languages [17]. An object protocol machine contains at least one attribute-identifier, the name of the object. The set of attributes of a behavior protocol machine can be empty. An initial state $s0_{pm} \in S_{pm}$ is always specified for an object protocol machine. A behavior protocol machine may not have the initial state if it is instantiated with an object.

Dependent protocol machines. Derived States

Transitions T_i of a protocol machine PM_i enable the updates of its own states; namely, those in S_i .

On the other hand, protocol machines can read the states of other protocol machines, although cannot change them.

Callback functions CB_i are used to read states of specified protocol machines and update attributes and calculate derived states of the behavior protocol machines.

Callback functions create dependencies between protocol machines. A dependency means that one protocol machine (usually a behavior protocol machine) needs to read the state of other protocol machines to calculate its own state. Such calculated states are called *derived states*, which distinguishes them from the *stored states* denoted in the model [19]. A protocol machine with derived (calculated) states is called dependent.

As all protocol machines are composed by means of the CSP parallel composition rules, a dependent protocol machine specifies extra “restrictions” on the acceptance of an event by other protocol machines of the protocol model. The protocol machines, which behavior is restricted, are not necessarily the same protocol machines, states of which have been read to derive the state of the dependent protocol machine. The ability of protocol machines to read the state of other protocol machines and restrict the behavior of other protocol machines is an asset useful for modeling of crosscutting concerns.

Each transition of a dependent protocol machine contains a derived state and an event, permitted on the way-in or out this state. The derived state can be either the pre-state or the post-state of this event. The pre-state semantics is similar to guards calculated in Colored Petri Nets (CPN) [12] and the UML state machines [24]. The post-state semantics does not exist in other modeling notations. If a post-state refuses the event caused its calculation, the event is rolled back, i.e., the system sends a message about the post-state value, and it is returned into the state that preceded to the event acceptance.

2.2.2 Protocol Modeling Notation

The Protocol Modeling Notation is defined in the Modelers Guide document that belongs to the tool ModelScope [18]. All semantic elements described in the previous subsection have the corresponding keywords used for the textual description of a protocol model.

The keywords (or entries) of the Protocol Modeling notation are MODEL, OBJECT, BEHAVIOUR, EVENT, GENERIC, and ACTOR. Subentries add specific definitions to entries. Below, we present the syntax of the textual description taken from the Modelers Guide [18].⁴ Subentries are described per entry.

1. MODEL Model_Name

This keyword is used to name a Protocol Model.

The Model_Name must be the same as the Package name for the Callbacks associated with the Model.

2. OBJECT Behaviour_Name

BEHAVIOUR Behaviour_Name

⁴The British spelling is used for the entries of the Protocol Modeling notation.

An OBJECT is a special BEHAVIOUR. OBJECT is used as the owning (top) BEHAVIOUR of an assembly of BEHAVIOURs that together model an OBJECT.

- (a) Behaviour_Name must be unique in the BEHAVIOUR namespace.
If Behaviour_Name is prefixed by “!”, the BEHAVIOUR has an associated callback.
- (b) Attribute_Name for specification of the OBJECT’s Behaviour_Name is required as a subentry for an OBJECT.
It is not allowed as a subentry for BEHAVIOUR.
Attribute_Name is used to identify instances in the user interface of ModelScope.
- (c) TYPE Behaviour_Type is a subentry for an OBJECT or a BEHAVIOUR.
Behaviour_Type must be one of the following values: ESSENTIAL, ALLOWED, or DESIRED.
If the subentry is omitted, ESSENTIAL is assumed.
The ESSENTIAL BEHAVIOUR is forced by ModelScope.
The ALLOWED or DESIRED BEHAVIOURs must have derived states, and cannot have stored attributes or any Event Processing Callbacks.
- (d) INCLUDES Behaviour_Name, Behaviour_Name, ...
This subentry specifies the structure of BEHAVIOUR composition.
A BEHAVIOUR named in the INCLUDES subentry must not be an OBJECT.
A BEHAVIOUR cannot include itself either directly or indirectly.
A BEHAVIOUR may not appear more than once in an INCLUDES structure.
- (e) ATTRIBUTES Attribute_Name: Type,
Attribute_Name: Type, ...
This subentry declares the ATTRIBUTES of the described OBJECT or BEHAVIOUR.
Attribute_Name must be unique within the OBJECT or BEHAVIOUR.
Type must be either a built-in type (String, Integer, Currency, Boolean, Date) for a value attribute, or a Behaviour_Name for a reference attribute.
If Attribute_Name is prefixed by “!” the attribute has a Derived Attribute Callback.
Invisible Attributes.
Parentheses used around an attribute and its type
(Attribute_Name : Type)
indicate that the attribute is invisible, i.e., not displayed at the user interface.
Apart from not being displayed, invisible attributes are otherwise treated by ModelScope exactly like visible attributes.
- (f) STATES State_Name, State_Name, State_Name, ...
This subentry declares the possible states of the OBJECT or BEHAVIOUR.
The state specifiers (@new, @any, @old) are not declared.
@new means the initial state of an object, before any transition has taken place.
@any specifies any state of the object, apart from @new.

@old is used as a post-state in a transition to indicate that the post-state is the same as the pre-state.

- (g) TRANSITIONS Transition, Transition,
Transition, ...

This notation element is used to specify transitions.

The form of a Transition is:

Pre_State * Event_Specifier = Post_State.

- i. Pre_State and Post_State are either a State_Name declared in the STATES subentry, or one of the state specifiers @new, @any, @old.
- ii. Event_Specifier is either an Event_Name from an EVENT entry, or a Generic_Name from a GENERIC entry.

3. EVENT Event_Name

ATTRIBUTES Attribute_Name : Type,
Attribute_Name : Type, ...

This entry declares an EVENT.

- (a) If Event_Name is prefixed by “!”, the Event has an Event Handling Callback.

Event_Name must be unique in the BHAVIOUR/EVENT namespace.

- (b) ATTRIBUTES subentry defines the attributes of the EVENT.

If Attribute_Name is prefixed by “!” the attribute has an Attribute Handling Callback.

- (c) Attribute_Name must be unique within the EVENT.

- (d) Type must be either an built-in type for a value attribute, or a Behaviour_Name for a reference attribute.

4. GENERIC Generic_Name

MATCHES Event_Specifier,
Event_Specifier,
Event_Specifier, ...

This entry specifies an alias for a set of events.

- (a) Generic_Name must be unique in the BEHAVIOUR/EVENT namespace.

- (b) MATCHES Event_Specifier,
Event_Specifier,
Event_Specifier, ...

This subentry specifies the membership of a GENERIC.

Event_Specifier is either an Event_Name from an EVENT entry, or a Generic_Name from a GENERIC entry.

The Event_Specifier can be subscripted to an OBJECT of BEHAVIOUR. In this case, it takes the form Event_Name [Attribute_Name] or Generic_Name [Attribute_Name].

A `Generic` cannot have itself as a member, directly or indirectly.

Subscripted and unsubscripted use of the same `Event` cannot be mixed in a `Generic`, either directly or indirectly.

```
5. ACTOR Actor_Name
   BEHAVIOURS Behaviour_Name, Behaviour_Name,
   Behaviour_Name, ...
   EVENTS Event_Name,
   Event_Name,
   Event_Name, ...
```

This entry declares an `Actor` of the model.

```
(a) BEHAVIOURS Behaviour_Name,
    Behaviour_Name,
    Behaviour_Name, ...
```

This subentry specifies the `Behaviours` in a `Actor`.

```
(b) EVENTS Event_Name,
    Event_Name,
    Event_Name, ...
```

This subentry specifies the `Events` in an `Actor`.

`GENERICs` cannot be used.

If no `ACTOR` is specified, `ModelScope` creates an `Actor All` in which all `Objects` and `Events` are visible.

2.2.3 *Example of a Protocol Model*

Figure. 2.8 renders the example “Google Screen with the Security aspect and the Email access” as a protocol model. There are many protocol models that can present this case with the same behavior. The protocol model in Fig. 2.8 has been built to make it comparable with the CPN model of the same case from Fig. 2.5. All the protocol machines, presented in Fig. 2.8, are described below in the textual protocol model.

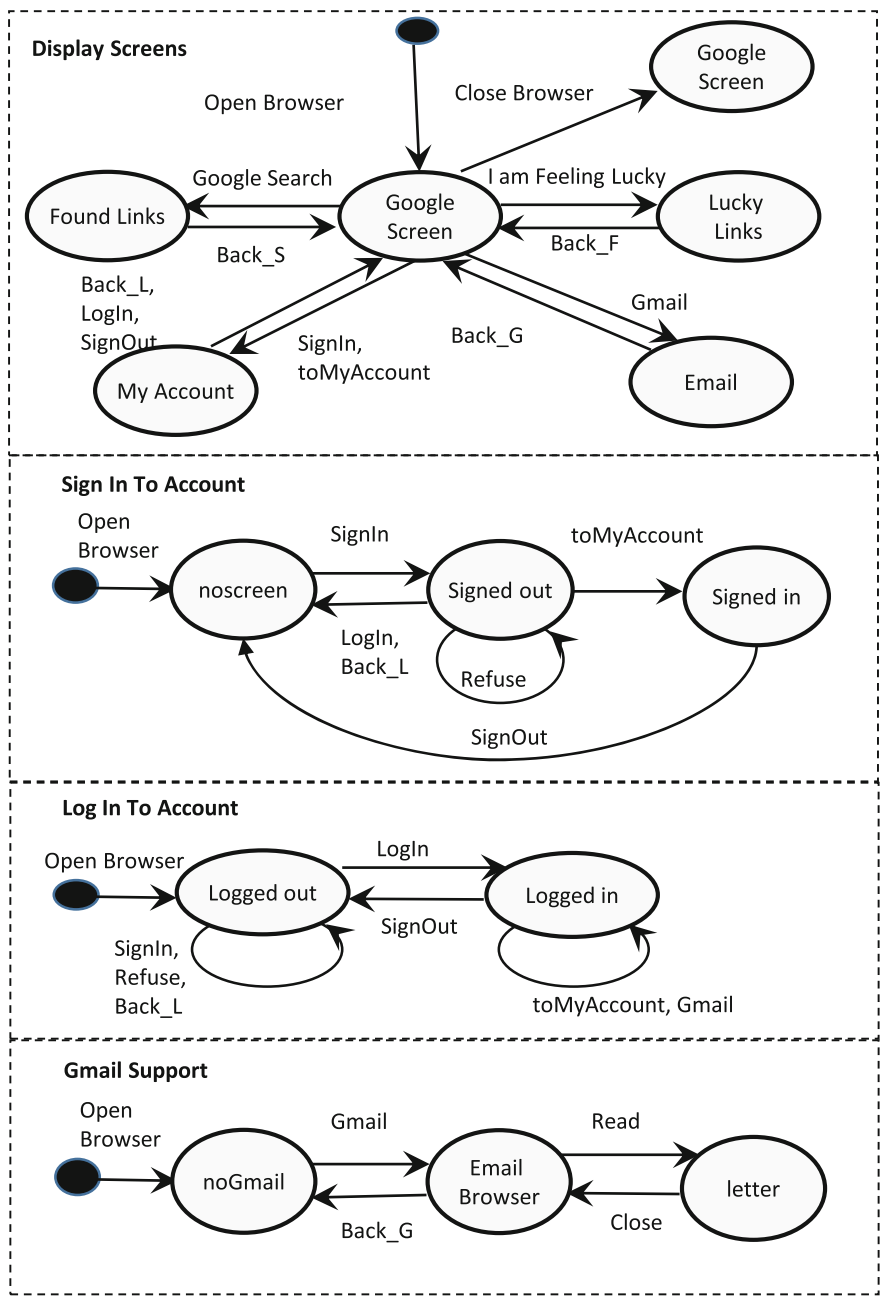


Fig. 2.8 A protocol model of the Google Screens example

OBJECT DisplayScreens

The protocol model contains the OBJECT protocol machine DisplayScreens and INCLUDES three BEHAVIOURS: LogInToAccount, SignInToAccount, and GmailSupport. The set of events recognized by a protocol machine can be found in the set of its transitions. For example, the OBJECT DisplayScreens presents the transitions available on the main Google Screen: OpenBrowser, GoogleSearch, IamfeelingLucky, SignIn. The SignIn event is synchronized with the security aspect implemented as two protocol machines: SignInToAccount and LogInToAccount.

Listing 2.1 OBJECT DisplayScreens

```

MODEL GoogleScreens
# OBJECT definitions
OBJECT DisplayScreens
    NAME DisplayScreensName
    INCLUDES LogInToAccount , SignInToAccount , GmailSupport ,
    ATTRIBUTES DisplayScreensName : String , ! Status : String ,
    STATES GoogleScreen , Closed ,
        LuckyLinks , FoundLinks , MyAccount , Email
    TRANSITIONS
        @new*OpenBrowser=GoogleScreen ,
        GoogleScreen*CloseBrowser=Closed ,
        GoogleScreen*GoogleSearch=FoundLinks ,
        FoundLinks*Back_S=GoogleScreen ,
        GoogleScreen*IamFeelingLucky=LuckyLinks ,
        LuckyLinks*Back_F=GoogleScreen ,
        GoogleScreen*SignIn=MyAccount ,
        MyAccount*LogIn=GoogleScreen ,
        MyAccount*Back_L=GoogleScreen ,
        GoogleScreen*toMyAccount=MyAccount ,
        MyAccount*SignOut=GoogleScreen ,
        GoogleScreen*Gmail=Email ,
        Email*Back_G=GoogleScreen

```

BEHAVIOUR SignInToAccount

This protocol machine specifies the screen that allows one to type a password to sign in, but also presents a menu for signing out. The set of events contains OpenBrowser, SignIn, LogIn, Refuse, Back_L, toMyAccount, SignOut.

Listing 2.2 BEHAVIOUR SignInToAccount

```

BEHAVIOUR SignInToAccount
    STATES noScreen , Signed in , Signed out
    TRANSITIONS
        @new*OpenBrowser=noScreen ,

```



```

noScreen*SignIn=Signed out ,
Signed out*LogIn=noScreen ,
Signed out*Refuse=Signed out ,
Signed out*Back_L= noScreen ,
noScreen*toMyAccount=Signed in ,
Signed in*SignOut=noScreen

```

BEHAVIOUR LogInToAccount

This protocol machine accepts or refuses the password and keeps the state of the user: logged in or logged out. The set of events contains OpenBrowser, SignIn, LogIn, Refuse, Back_L, toMyAccount, Gmail, SignOut, Refuse.

Listing 2.3 BEHAVIOUR LogInToAccount

```

BEHAVIOUR LogInToAccount
STATES  logged in , logged out
TRANSITIONS
    @new*OpenBrowser=logged out ,
    logged out*SignIn=logged out ,
    logged out*LogIn=logged in ,
    logged in*toMyAccount=logged in ,
    logged in*Gmail=logged in ,
    logged in*SignOut=logged out ,
    logged out*Refuse=logged out ,
    logged out*Back_L=logged out

```

BEHAVIOUR GmailSupport.

The GmailSupport functionality becomes enabled only when the user is logged in. If enabled, it supports opening Open Browser of the Email Browser, reading a letter (event Read), closing it (Close), closing the EmailBrowser (event Back_G). When the user is logged out (BEHAVIOUR LogInToAccount is in state LogInToAccount), the GmailSupport is unavailable.

Listing 2.4 BEHAVIOUR GmailSupport

```

BEHAVIOUR GmailSupport
STATES  EmailBrowser , letter , noGmail
TRANSITIONS
    @new* OpenBrowser=noGmail ,
    noGmail*Gmail=EmailBrowser ,
    EmailBrowser*Read=letter ,
    letter*Close=EmailBrowser ,
    EmailBrowser*Back_G=noGmail

```

Event types

The event types are specified by their names and attributes.

For example, the specification of the event type `OpenBrowser` tells that it is used by the object type `DisplayScreens` and the name of this object is also specified `DisplayScreensName: String`.

Listing 2.5 EVENT definitions

```
# EVENT definitions

EVENT OpenBrowser
    ATTRIBUTES DisplayScreens: DisplayScreens ,
               DisplayScreensName: String
EVENT GoogleSearch
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Back_S
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT IamFeelingLucky
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Back_L
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT SignIn
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT SignOut
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Gmail
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Back_F
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Back_G
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Close
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Read
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT LogIn
    ATTRIBUTES DisplayScreens: DisplayScreens
EVENT Refuse
    ATTRIBUTES DisplayScreens: DisplayScreens
```

Callback functions

For convenience of model simulation, the state of a `DisplayScreens` instance is a callback function visualizing the current state:

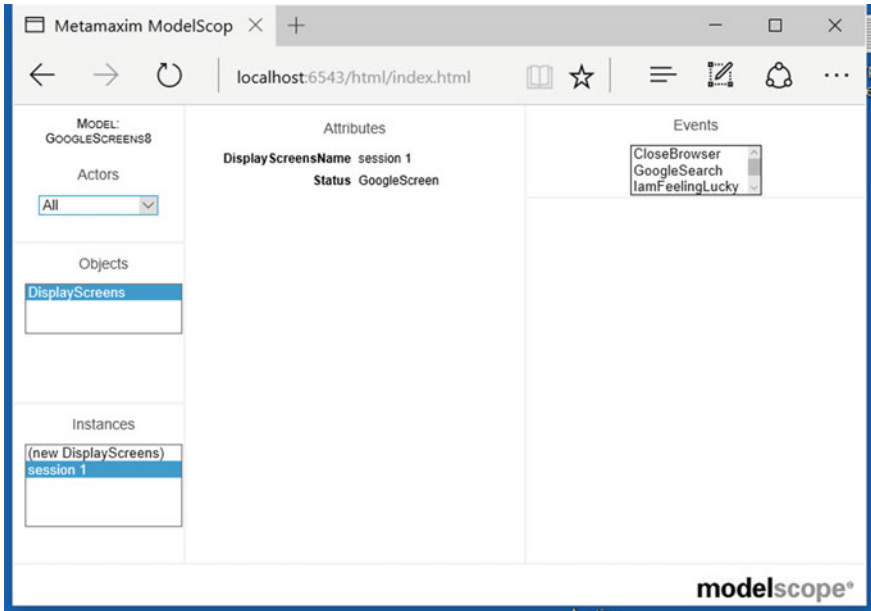


Fig. 2.9 An execution screen

Listing 2.6 Callback DisplayScreens

```
package GoogleScreens;
import com.metamaxim.modelscope.callbacks.*;
public class DisplayScreens extends Behaviour {
    public String getStatus () {
        return (this.getState("DisplayScreens"));
    }
}
```

The protocol model is executable in the Modelscope tool [18], so that the user can play with the model, interactively choose events, fill them with data, submit events, and analyze the system response. Figure 2.9 shows an execution step of the protocol model after acceptance of an event `SignIn`.

In the next chapters of this book, we shall master the language of Protocol Modeling allowing to build a model from small fragments of behavior. The Modelscope will compose fragments with preservation of observational consistency of their behavior in the whole model.

2.2.4 Observational Consistency in Protocol Models

The proof of the property of Observational Consistency for the CSP parallel composition of dependent machines with data was done by A. McNeile [15].

In order to prepare the theorem proof, let us remind that

1. A *protocol model* is a CSP parallel composition of protocol machines

$$PM = \parallel_{i=1}^n PM_i = \parallel_{i=1}^n (S_i, E_i, T_i, A_i, CB_i) = (S, E, T, A, CB).$$

E_i is the alphabet of events of PM_i .

2. *Independent and Dependent protocol machines.* Among the composed protocol machines PM_i there are both dependent and independent protocol machines.

- An *independent* machine PM_i does not read the local storage of other machines and, in particular, does not use information from the local storage of other machines to determine its own state.
- A *dependent* machine reads the local storage of other machines and uses the read information to derive its own state.

3. *A Trace of a Protocol Machine.*

If S a *trace* of a machine PM_i then

1. All events in S belong to E_i and
2.
 - i. For an independent PM_i :
If all events of S are presented in turn to PM_i , they are accepted by PM_i (i.e., no event in S is refused by PM_i)
 - ii. For a dependent PM_i :
There exists a protocol machine $\exists Q$, such that
 - $E_Q \subseteq E_i$,
 - $(PM_i \parallel Q)$ is *independent* and
 - If all events of S are presented in turn to $(PM_i \parallel Q)$, they are accepted by $(PM_i \parallel Q)$, i.e., S is a trace of $(PM_i \parallel Q)$.
 - iii. Note that, by taking Q to be a machine that *ignores all events presented to it* $E(Q) = \emptyset$; $\emptyset \subseteq E_i$, the part *ii* of the definition (for the case PM is dependent) is *also true* if PM is independent.

4. *The restriction operator* of a sequence of events S to an alphabet of events E_i of the protocol machines PM_i

$$S \upharpoonright_{E_i}$$

produces a new sequence of events by removing from S every element that does not belong to the alphabet of E_i .

Theorem 2.1 *A Protocol Machine is observationally consistent with its components:*

$$\text{traces}(PM_1 \parallel PM_2) \upharpoonright_{E_1} \subseteq \text{traces}(PM_1)$$

Proof: Suppose that we have a trace T , such that $T \in \text{traces}(PM_1 \parallel PM_2)$. There are two cases to consider

- PM_1 is independent.

In this case, we know that PM_1 must either have accepted or ignored every event in T because, had it refused an event, that event would also have been refused by $(PM_1 \parallel PM_2)$ (by the rules of CSP composition) and T would not then be a trace of $(PM_1 \parallel PM_2)$. As PM_1 is independent, it is unaffected by the presence of PM_2 in composition, and will accept/ignore events of T in the same way when executing alone. Moreover, the execution of PM_1 alone is unaffected by events not in E_1 as it ignored these. So, $(T \upharpoonright_{E_1}) \in \text{traces}(PM_1)$.

- PM_1 is dependent.

It should read some storage in order to derive and update its states. In this case, from the definition of “A trace of a Protocol Machine”, there exists such a protocol machine Q , such that with $E_Q \subseteq E_1$ and $(PM_1 \parallel Q)$ is independent and has T as a trace.

We need to prove that $(PM_1 \parallel Q \parallel PM_2) \upharpoonright_{E_1} \subseteq \text{traces}(PM_1)$

Now suppose that the local storage of $(Q \parallel PM_2)$, which we will denote by Σ , has value B_i before the i^{th} event of $T \upharpoonright_{E_1}$ and A_i after the i^{th} event of $T \upharpoonright_{E_1}$. Using these storage values, we construct a new machine, Q' with storage Σ' , to simulate for PM_1 the storage environment provided by $(Q \parallel PM_2)$ through execution of the event sequence T , as follows:

1. $E_{Q'} = E_{PM_1}$ Q' has the same set of events of PM_1 , which appear in T .
2. Q' is trace independent (i.e., we construct it to make no access outside its own local storage).
3. Q' accepts all events presented to it (i.e., we construct it to make no refusals).
4. The local storage Σ' of Q' mirrors Σ , so that PM_1 cannot differentiate between accessing Σ and accessing Σ' .
5. The value of the local storage Σ' is derived (calculated on-the-fly during execution) by Q' . The algorithm used by Q' for this derivation gives Σ' the value B_j if invoked to give the pre-state of the j^{th} event of $T \upharpoonright_{E_1}$, and the value A_j if invoked to give the post-state value of this event.

With this construction, Σ' simulates *exactly* the local storage values that PM_1 would access from Σ before and after each event that PM_1 processes (i.e., before and after processing each event in $T \upharpoonright_{E_1}$). Thus we have constructed Q' so that

- $(PM_1 \parallel Q')$ is trace independent. This is because $(PM_1 \parallel PM_2 \parallel Q)$ is trace independent, so Q' , by offering the same storage environment to PM_1 , must fully resolve every access that PM_1 makes outside its own local storage. Moreover, we have constructed Q' to make no accesses outside of its own local storage.

- The set of events of PM_1 is equal to the set of events of $(PM_1 \parallel Q')$:
 $(E_1 = E_{PM_1 \parallel Q'})$.
 This is because $E_{Q'} \subseteq E_1$ by construction and $E_{PM_1 \parallel Q'} = E_1 \cup E_{Q'}$.
- $(PM_1 \parallel Q')$ accepts all the events of $T \upharpoonright_{E_1}$. This is because
 - Q' provides PM_1 with storage values (in Σ') that are identical to those it sees (in Σ) when consuming the event sequence T while composed with $(PM_2 \parallel Q)$; and
 - Q' is constructed not to refuse any event presented to it.

Thus, $T \upharpoonright_{E_1}$ is a trace of $(PM_1 \parallel Q')$ and so Q' satisfies the existence criterion required by Definition “A trace of a Protocol Machine” to establish that $T \upharpoonright_{E_1} \in \text{traces}(PM_1)$.

□

2.2.5 Local Reasoning on Protocol Models

Definition 2.2 Local reasoning on a model is an ability to make conclusions about traces of the whole model of a system by examining the model’s parts.

Local reasoning in Protocol Modeling is based on the following property of the CSP parallel composition: If we take a sequence of events, S , which is accepted by the composition $(PM_1 \parallel PM_2)$ of the two machines PM_1 and PM_2 , then the subsequence, S' , of S obtained by removing all events in S that are not in the alphabet of PM_1 , would be accepted by the machine PM_1 by itself.

In other words, composing another machine with PM_1 cannot “damage its trace behavior.” This means that we can use properties of PM_1 (or PM_2) *alone* to argue about the behavior of $PM_1 \parallel PM_2$ to support local reasoning.

In our running example of the Protocol Model of the Google Screens: the composition of protocol machines in Fig. 7.3 accepts a sequence $S = \text{OpenBrowser}, \text{SignIn}, \text{LogIn}, \text{Gmail}, \text{Read}, \text{Close}, \text{Back_G}$.

This sequence being restricted to the alphabet of events of the protocol machine *GmailSupport* is transformed into $S' = \text{OpenBrowser}, \text{Gmail}, \text{Read}, \text{Close}, \text{Back_G}$. Only by examining the protocol machine *GmailSupport*, we know the sequences of the email handling in the whole system.

2.3 Protocol Modeling and Conceptual Modeling

The modeling and simulation community actively discusses the role of conceptual models for simulation. Concepts are universal means for understanding the world in general and therefore, for modeling of all kinds of man-designed systems. Traditionally, a concept is defined statically, using its name and some attributes. Conceptual

models reflect the relations between concepts. Conceptual modeling covers ever growing area of applications, especially in modern enterprises. Conceptual models present different layers of a system from low technological, up to application, business and motivation layers. Nothing stops relating concepts of different abstraction layers together. The Model-Driven Architecture community has succeeded with creating tools for generation of executable code from conceptual models.

However, the tools are able to generate executable models from the complete conceptual model. Even local changes of a conceptual model demand complete regeneration and retesting of the complete executable code. It is because that MDA tools are based on the UML semantics of behavior of classes corresponding to concepts. The behavior modeling techniques adopted by the UML do not possess observational consistency and local reasoning.⁵ With the MDA approaches, easy changes of requirements is impossible; each new requirement demands revising of the complete model and code regeneration.

The Protocol Modeling approach possesses the observational consistency and provides the developers possibility to change locally, to build protocol machines corresponding concepts, requirements, business rules and policies. Structurally, protocol models can be seen as conceptual models. The transitions and the composition of the approach clarify the semantics of concept's relations in terms of extending of the state space of one by another, synchronization, reading, and updating of values of attributes. Many complex concepts (as for example, capability [28], readiness, etc.) can be only explained by relating several concepts via a process and communication, i.e., they become clear only when conceptual model and behavior model are related together. Thanks to the CSP parallel composition, protocol models are able to separate concepts and aspects, requirements and policies as parts of the executable model and allow for local reasoning and interaction about them.

Taking into account the expressive and compositional power of Protocol Modeling, we consider it as the first foundation of the Interactive Modeling and Simulation. In the next chapters, we will demonstrate the Interactive Modeling and Simulation with protocol models and provide more examples for mastering the local reasoning on protocol models. Local reasoning is a very powerful abstraction technique for the Interactive Modeling and Simulation as well as for understanding the computer-supported businesses around us.

Problems

2.1 What is a Finite State Machine? What are its modeling abilities and expressivity limitations?

2.2 Build a Finite State Machine of a door in your room. You may open the door and close it. Define the states, events, and transitions.

⁵More about the semantics of behaviors in UML can be found in [16].

2.3 What is a distinctive feature of holistic approaches? How to model a system state in holistic modeling approaches?

2.4 How many event semantics do you know? Which of that semantics would you use to model interactions? Why?

2.5 Which of the semantics of a state would you use to model interactions? Why?

2.6 Which of the semantics of a transition would you use to model interactions? Why?

2.7 Describe different composition semantics that you have learned throughout this chapter.

2.8 How do you understand the property of observational consistency? Does this property make sense for the transducer models? Why?

2.9 Which of the protocol machines of the Protocol Model of the Google Screens example (Fig. 7.3) should you examine in order to understand the sequences of the email access?

What are the sequences of the email access in the whole Protocol Model of the Google Screens?

What are the sequences of the security aspect in the whole Protocol Model of the Google Screens?

References

1. J.C.M. Baeten, W.P. Weijland, *Process Algebra* (Cambridge University Press, New York, 1990)
2. T. Basten, W.M. van der Aalst, Inheritance of behavior. *J. Log. Algebr. Program.* **47**(2), 47–145 (2001)
3. J. Bergstra, J. Klop, Process algebra for synchronous communication. *Inf. Control* **60**(13), 109–137 (1984)
4. J.A. Bergstra, J.W. Klop, Algebra of communicating processes with abstraction. *Theor. Comput. Sci.* **37**, 77–121 (1985)
5. E. Börger, R.F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis* (Springer, Berlin, 2003)
6. G. Bracha, W. Cook, Mixin-based inheritance, in *OOPSLA/ECOOP '90 Proceedings of the European Conference on Object-oriented Programming, Languages, and Applications*, 303–311 (1990)
7. CPN tools. <http://cpntools.org> (2015)
8. J. Ebert, G. Engels, Structural and behavioural views on omt-classes, in *Proceedings of Object-Oriented Methodologies and Systems, International Symposium ISOOMS' 94, Palermo, Italy, September 21–22, 1994*, 142–157 (1994)
9. W. Fokkink, *Introduction to Process Algebra*, 1st edn. (New York Inc, Secaucus, 2000)
10. C. Hoare, *Communicating Sequential Processes* (Prentice-Hall International, Englewood Cliffs, 1985)
11. M. Jackson, *System Development* (Prentice Hall, Englewood Cliffs, 1983)
12. K. Jensen, *Coloured Petri Nets* (Springer, Berlin, 1997)

13. A. Kent, J.G. Williams, *Encyclopedia of Computer Science and Technology: Volume 25 - Supplement 10: Applications of Artificial Intelligence to Agriculture and Natural Resource Management to Transaction Machine Architectures*. Encyclopedia of Computer Science Series. (Taylor and Francis, 1991)
14. A. McNeile, E. Roubtsova, Protocol modelling semantics for embedded systems, in *International Symposium on Industrial Embedded Systems, 2007. SIES '07*, 258–265, July 2007
15. A. McNeile, E. Roubtsova, CSP parallel composition of aspect models, in *Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, AOM '08, pp. 13–18, New York, NY, USA, (2008). ACM
16. A. McNeile, E. Roubtsova, Composition semantics for executable and evolvable behavioral modeling in mda, in *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, BM-MDA '09, pp. 3:1–3:8 (2009)
17. A. McNeile, E. Roubtsova, *Aspect-Oriented Development Using Protocol Modeling*, vol. 6210 (LNCS, 2010), pp. 115–150
18. A. McNeile, N. Simons, <http://www.metamaxim.com/> (2005)
19. A. McNeile, N. Simons, Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Softw. Syst. Model.* **5**(1), 91–107 (2006)
20. R. Milner, *A Calculus of Communicating Systems*, vol. 92, Lecture Notes in Computer Science (Springer, Berlin, 1980)
21. F. Moller, The importance of the left merge operator in process algebras, in *Automata, Languages and Programming*, vol. 443, Lecture Notes in Computer Science, ed. by M. Paterson (Springer, Berlin, 1990), pp. 752–764
22. H.G. Molter, Discrete event system specification, in *SynDEVS Co-Design Flow*, pp. 9–42 (2012)
23. OMG. Business Process Model and Notation (2011)
24. OMG. *Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03* (2003)
25. W. Reisig, Petri nets and algebraic specifications. *Theor. Comput. Sci.* **80**(1), 1–34 (1991)
26. E. Roubtsova, Chapter Two - Advances in Behavior Modeling, *Advances in Computers*, vol. 97 (Elsevier, 2015), pp. 49–109
27. E. Roubtsova, M. Aksit, Extension of Petri Nets by Aspects to Apply the Model Driven Architecture Approach, in *Preliminary Proceedings of the 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB)* (2005)
28. E. Roubtsova, V. Michell, Behaviour models clarify definitions of affordance and capability, in *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*, BM-FA '14, pp. 6:1–6:10 (2014)

Interactive Modeling and Simulation in Business System
Design

Roubtsova, E.

2016, XIV, 201 p. 63 illus., 23 illus. in color., Hardcover

ISBN: 978-3-319-15101-4