

In this chapter, we discuss basic DEVS and SES concepts and tools to support working with these concepts in the context of an actual modeling and simulation environment, the MS4 Modeling Environment. To address the different perspectives that stakeholders bring to the modeling and simulation world, we provided three different introductions aimed at three different types of users. For the general M&S user, we provided a description of the concepts supported by MS4 Me™ through the immediate application of its most basic tools. For the M&S Developer, we provided a more advanced introduction to MS4 Me™'s underlying DEVS concepts and theory and the tools that support them. For the M&S Expert Professional, we offered a glimpse into MS4 Me™'s features in more depth as well as the theory that supports them. This book is divided into three parts. In the first part, we discuss basic DEVS and SES concepts and tools to support working with these concepts in the context of an actual modeling and simulation environment, called MS4 Modeling Environment (MS4 Me™). Then in Part II, we discuss more advanced concepts that such tools can support, and in Part III, we discuss some actual applications that throw light on the kinds of System of Systems problems that can be addressed with such concepts and tools.

---

## 2.1 The MS4 Me Is a Modeling and Simulation (M&S) Environment

MS4 Me is a modeling and simulation (M&S) environment developed as the first in a commercial line of DEVS products ([ms4systems.com](http://ms4systems.com)). MS4 Me is aimed at a variety of users such as managers, modelers, developers, and programmers enabling them to work at the level for which they are most comfortable and productive. With this variety in mind, this chapter offers three different introductions aimed at three different types of users. Let's call these types of users Drivers, Designers, and Racing Pros. Drivers want to know what basic things cars can do and how they can

make cars do those things. Drivers are not interested in the car per se—only how well it gets them to where they want to go. Designers want to know how to make better cars—so they need to know what things are under the hood and how those things work together to make a car do what it does. Racing Pros want to exploit the car to its extreme. They want to get familiar with all its features and how to use them to the fullest extent. So like a Driver, you might want to know what MS4 Me does as a tool in your tool box. Let's call you the M&S User in this case. Or like a Designer, you might be an M&S Developer and want to open the hood so as to get into MS4 Me's underlying DEVS concepts and theory. Or like a Racing Pro, you are an M&S Expert Professional and want to know MS4 Me's features in depth as well as the theory that supports them.

You can skip to the introduction that best characterizes your needs and roles. However, since each introduction is written with a different perspective, you might be better served if you skimmed each one looking for nuggets that might help you understand your own focus and also how it interacts with others.

### 2.1.1 Introduction for the M&S User

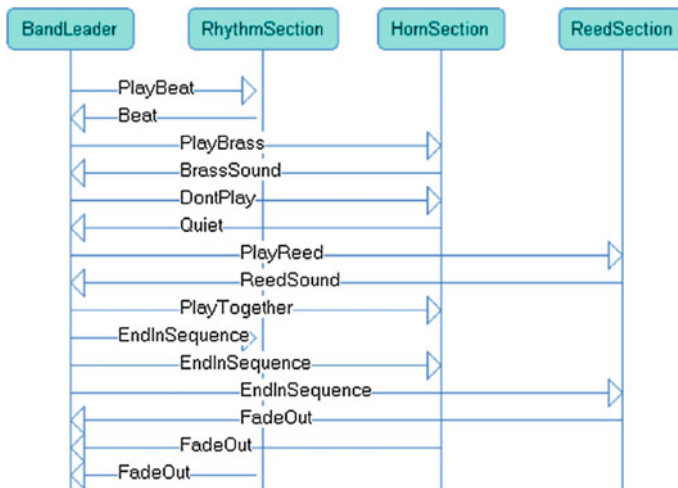
For the modeling and simulation general user, the MS4 Me modeling environment offers a restricted English language interface to generate models and then simulate the behavior graphically in real time. With minimal training, an M&S user such as a systems engineer or manager can take a need expressed in common English into a restricted, but clearly stated, set of English statements that are checked and automatically transformed into graphical models on the fly; then these individual models can be coupled to other models and presented to the stakeholder to ensure that the need is expressed as intended and that it fits a standardized process.

If the need expressed in the model is deficient or incorrect, the user and the stakeholder can negotiate immediately where intent and model have diverged iteratively arriving at a very precise statement that is formal and adheres to a standard process and language.

MS4 Me is capable of expressing very simple processes, such as workflows, as well as extremely complex, and precise timing and mathematical functions, including complex functions required to coordinate activities of components.

**Coordination Example: Jazz Band Leader** Much of today's work is done in teams, and team coordination is becoming ever more required. Coordination can be very scripted in the way a playwright determines the flow and actions of the actors. Or it can be very loose as in a Jazz combo where individual players have a large role in determining the outcome. And as we will see in many examples through the book, coordination may have to be implemented at many levels of organization of systems to enable loosely coupled or semi-autonomous components to work toward common goals.

Let's consider an example where coordination lies somewhere in between very scripted and very loose in which a band leader coordinates the sections (rhythm, woodwind, horns) of a jazz band. We will focus on how the leader starts the



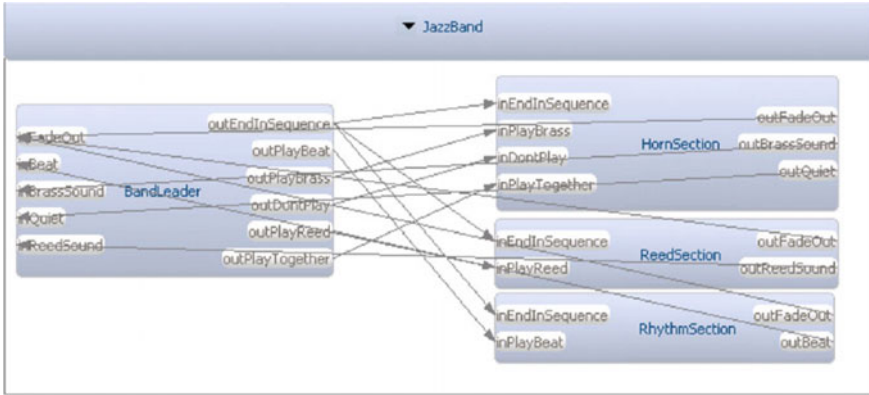
**Fig. 2.1** Sequence diagram interface example for the Jazz Band

sections playing, changes the lead from one section to another, and brings the piece to a close in which sections fade out in a sequence. Using the sequence diagram interface of MS4 Me, you can easily lay out this kind of coordinated interaction.

As illustrated in Fig. 2.1, the BandLeader, Rhythm, Horn, and Reed sections are actors in the diagram, each with its own lifeline descending down the page. Message transmissions from a sender to a receiver are shown as labeled arrows and presented in the order in which they occur as time advances down the page. For example, the BandLeader starts the piece by telling the Rhythm section to provide the beat. The Rhythm section responds by providing the beat to the BandLeader (We could complicate the diagram by also drawing arrows labeled by PlayBeat to the other sections, to indicate that they also hear the beat being played). From this input, you can automatically generate a model where you can see the actors interacting and events occurring as prescribed. This model can be viewed in the Simulation Viewer as shown in Fig. 2.2.

Watching the flow in the Simulation Viewer, you can check whether the structure and behavior are as you would like them to be, and if not, you can change the input at two levels:

- You can go into the files generated and change some of the times in the actor models, to change the times at which events occur—corresponding to what we call a change in behavior.
- You can go back and change the sequence diagram to alter the flow of events. This is a more radical change in the model—corresponding to what we call a change in structure.



**Fig. 2.2** The Jazz Band Model generated from the sequence diagram

We will show how the model generated by the sequential diagram interface leads to more advanced uses after presenting the remaining introductions.

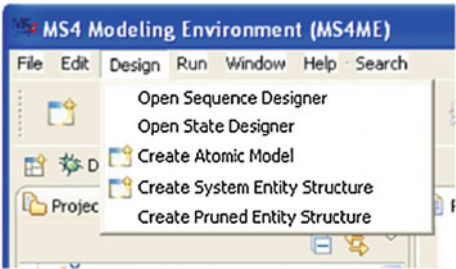
### 2.1.2 Introduction for the M&S Developer

For the modeling and simulation developer, MS4 Me offers a powerful, adaptable platform designed to develop DEVS models and simulations quickly. It includes constrained natural language (NL) interface which greatly speeds the development of models and their dynamic simulations to near real time. The graphical simulations can be run immediately upon input of the model. The NL interface and animation capability lend themselves well to capture requirements concisely, but rigorously. The tool is easy to use with only limited training. It is this combination of linguistic and dynamic graphical display of a need that will allow stakeholders and system engineers to visualize and negotiate the capabilities and behaviors expressed.

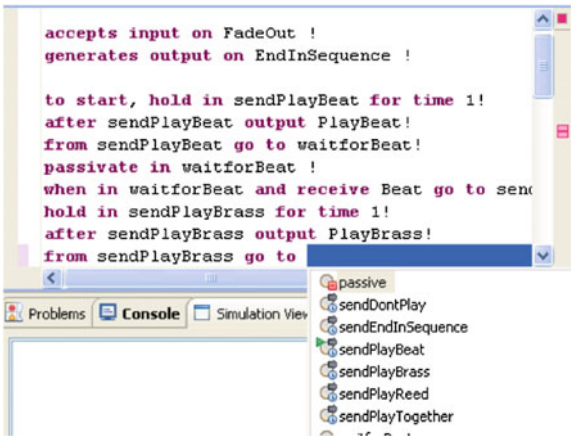
The vision of a DEVS Modeling and Simulation Environment is to provide an integrated development environment dedicated to the creation of DEVS models and their simulation. Such an environment makes developers feel they are working with a complete set of tools that are able to support all the functions needed to create DEVS models and simulate them within, or externally to, the environment. Such a vision has become feasible with the advent of the Eclipse open source community ([www.eclipse.org](http://www.eclipse.org)) and its support of extensible language development and other programming frameworks. One such framework, Xtext, provides a set of domain-specific languages and tools (parsers, type-safe abstract syntax tree, interpreter, etc.) to create a programming language and automatically generate its implementation in the Java Virtual Environment.

MS4 Me employs Xtext, its Extended Bachus-Naur Form (EBNF) grammar within the Eclipse Modeling Framework on the Rich Client Platform, and the

**Fig. 2.3** Part of MS4 ME initial user interface



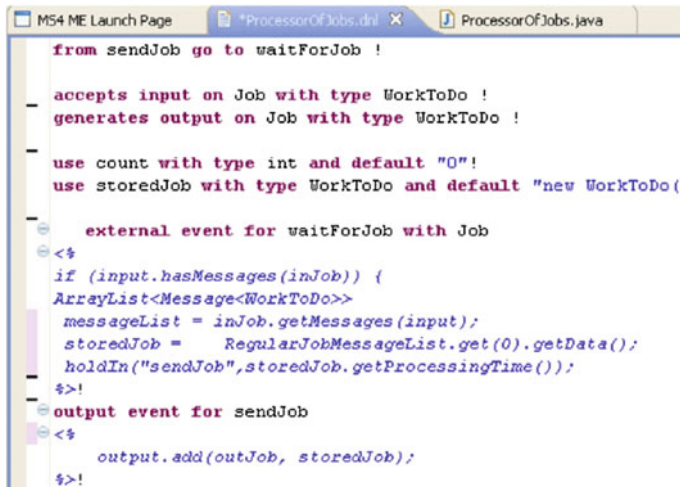
**Fig. 2.4** Constrained natural language for creating atomic models



Graphical Modeling Project to provide a full blown IDE specifically tailored to a DEVS development environment.

Figure 2.3 illustrates how the MS4 Modeling Environment user interface sets the look and feel for access to tools dedicated to DEVS modeling and simulation. The Design drop-down menu displays items that open files for the three main types of object: (1) Finite Deterministic DEVS (FDDEVS) that creates atomic models, (2) System Entity Structures that create families of hierarchical coupled models, and (3) pruned entity structure scripts that make choices from the available alternatives to specify a particular hierarchical coupled model. These objects represent and extend basic system concepts—atomic models, coupled models, and hierarchical (or nested) coupled models. The environment is concerned with providing tools to construct such models, test them for correctness, modify them until satisfactory, and simulate or animate them.

By using Eclipse’s Xtext EBNF grammar development facility, MS4 Me provides a constrained natural language to define FDDEVS models. As illustrated in Fig. 2.4, there are seven basic sentence types with variable slots that together define a FDDEVS model. These sentence types define such elements as input and output ports, states (including initial state), time advances, internal transitions, external transitions, and generated outputs. As the modeler writes the text, the editor parses it and creates an outline shown on the right of the figure that displays the structure



**Fig. 2.5** Tagged blocks for extending FDDEVS models to full-fledged DEVS models

that has been defined. Besides providing instant visualization, and click-access to source definitions, the captured information is available for model processing, as discussed earlier.

As text is entered, the parser provides syntax checking and sentence completion assistance. Such assistance is also content-based in that permissible entries are shown on request—the example in Fig. 2.4 illustrates that the parser is expecting a state and suggests states that have been entered earlier as candidates.

Although FDDEVS models have the essential properties of DEVS models, they form a subclass of all DEVS models (hence of all discrete-event systems). Figure 2.5 shows how the MS4 Me provides constructs to enable extending a FDDEVS model to become a full-fledged DEVS model implemented in Java. The linguistic support allows modelers to specify the types of DEVS messages accepted by input ports and generated by output ports, interpretation of inputs and generation of outputs, state variables and their types, new types as required, and especially operations on state variables invoked by internal and external transitions. The grammar recognizes tagged blocks for internal and external transitions in which Java code that executes the desired transition can be placed. The modeler can inspect and test the generated Java model, returning always to the FDDEVS file to make changes. Thus, consistency is always maintained between the high level specification (FDDEVS) and the implementation (Java). The approach realized by the tagged blocks also maintains traceability back from the resulting Java code to its block source.

The natural language interface for constructing System Entity Structures (SES) is illustrated in Fig. 2.6. The most fundamental statement here is the one in the modeler provides a decomposition of a system in terms of components from a certain perspective. The couplings associated with this perspective can then be defined and linked with this perspective. Hierarchical construction is done by

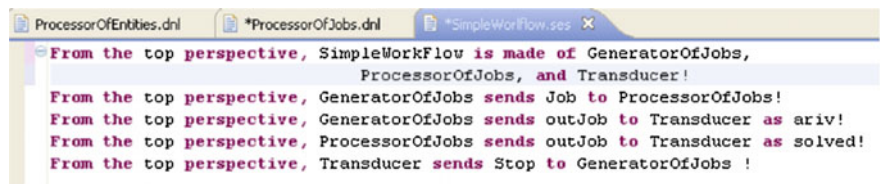


Fig. 2.6 Constrained natural language for System Entity Structure specification

recursive decomposition of a component to the depth desired. Both external and internal couplings are easily specified. The modeler may adopt any number of perspectives for decomposing a system or component according to the modeling objective and level of resolution needed.

The SES formalism supports a powerful extension of hierarchical system concepts that we briefly touch on here (Wymore 1967). For an in-depth survey of Wymore’s system theory and its relation to model-based system engineering (Friedenthal et al. 2009), see Ören (1984), Ören and Zeigler (2012). Figure 2.7 illustrates an SES for an unmanned air vehicle testing environment, which illustrates both decompositions (single line icons) and specializations (double line icons), where a specialization offers a choice of alternatives to plug into a component slot.

For example, SensorPackage can be decomposed, from the experiment perspective, into various sensor components such as FeedBack Sensor, Observation Sensor, MotionSensor, and WeaponSensor. In addition, TestAgent has a specialization, labeled by Scenario, into alternatives such as Baseline, Observational, or Attack—selection of one will configure the TestAgent appropriately. MS4 Me provides a user interface to support pruning of choices (i.e., pruning of decompositions and specializations) with the selections recorded in the files for pruned entity structures. Automatic transformation of such structures into simulation models affords a system design environment for investigating a family of possible

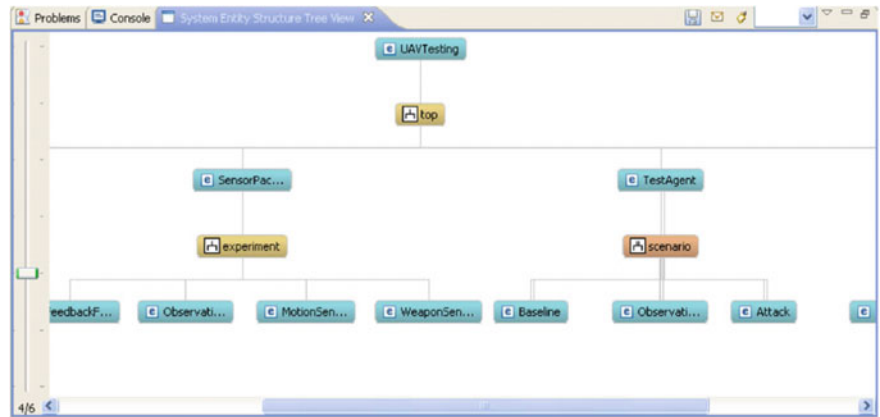


Fig. 2.7 System Entity Structure tree showing decomposition and specialization icons

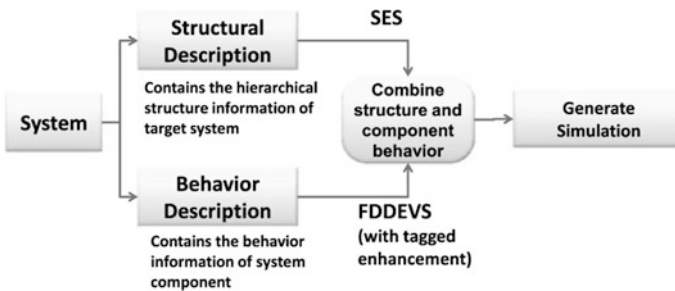
architectural models through simulation. Reaccessing such files for subsequent copying and modification as desired supports reuse and extensibility.

## 2.2 Introduction for the M&S Professional

Around the time of the emergence of DEVS as the computational basis for systems simulation, another important trend took hold. Object orientation (OO) was first introduced in simulation and later spread to programming in general. It is fair to say that OO is at the heart of current information technology, so much so, that its presence is often taken as a given. For simulation modeling, DEVS and OO formed an ideal marriage. DEVS brought the formal and conceptual framework of dynamic systems while OO provided a rapidly evolving wealth of implementation platforms for DEVS models and simulations—first in languages such as Java and C++, and later in network and Web infrastructures, and today continuing in the future toward Cloud information technologies (Wainer and Mosterman 2009). The first implementation of DEVS in object orientated form was described in Zeigler (1987), and there are currently numerous such implementations, some of which are listed in DEVS (2012). In the next section, we discuss how Wymore’s concepts take computational form in today’s information technology implementations of the DEVS formalism.

### 2.2.1 System Structure and Behavior

As illustrated in Fig. 2.8, MS4 Me employs the essential system concepts of structure and behavior to generate simulation models. The modeler provides the structural description, essentially the hierarchical coupled model, by writing the System Entity Structure in natural language form. The modeler provides the behavioral description by writing the lowest level component atomic models in natural language form. After discussing the basics of these concepts and their natural language descriptions, we will return to the sequence diagram input



**Fig. 2.8** MS4 Me’s approach to specifying structure and behavior



interface (recall the Jazz Band example) that writes the natural language coupled and atomic specifications for you.

### 2.2.2 Finite Deterministic DEVS (FDDEVS)

We begin with a brief introduction to the contained natural language and Finite Deterministic DEVS (FDDEVS).

**States** A state can either be a “hold state” or a “passive state.” A hold state is one that the model will stay in for a certain amount of time before automatically changing to another state (via an internal transition). A passive state is one that the model will remain in indefinitely (or until it receives a message that triggers an external transition).

**Passive States** To define a passive state, use the following syntax:

```
passivate in STATE_NAME!
```

**Hold States** To define a hold state, use the following syntax:

```
hold in STATE_NAME for time 5.7!
```

**Initial States** One state in the model must be designated as the initial state. To do this, the state description must start with “to start.” For example, if we wanted to make the previous state the initial state, we would use this syntax:

```
to start passivate in STATE_NAME!
```

or

```
to start hold in STATE_NAME for time 5.7!
```

**Internal Transitions** Every hold state in the model must have one and only one internal transition defined in order to specify the state to which the model should transition after the specified amount of time. Internal transitions use the following syntax:

```
from CURRENT_STATE go to NEXT_STATE!
```

```
<add extra line>
```

**Output** Any state that has an internal transition can also have one output message that is generated before that internal transition occurs. The syntax for this is:

```
after STATE_NAME output OUTPUT_MESSAGE!
```

```
<add extra line>
```

**External Transitions** Any state can have one or more external transitions defined. An external transition defines an input message that the model might receive when in a given state and the state to which the model should transition in reaction to that input message. The syntax is:

```
when in STATE_NAME and receive INPUT_MESSAGE go to NEXT_STATE!
```

### 2.2.3 System Entity Structure (SES)

Let's continue with the second concept, the System Entity Structure (SES). One of the powerful capabilities of the MS4 Me tool is the ability to couple multiple models into a larger and more complete system. The SES language is used to describe how a system is decomposed into subsystems when viewed from a certain perspective, different specializations of a system that might occur, messages sent from one system to another, and variables that a system might have. A SES is made up of:

**Aspects** describing subsystems that make up a system when that system is viewed a certain way.

*Example:* A car has an engine, a transmission, and a chassis when one considers the structural components of the car, but it also has a manufacturer, model, and license plate when one considers the physical description of the car.

**Specializations** that describe different subsystems that perform the duties of some system.

*Example:* Continuing the car example from above, the engine might be an electrical engine, a gasoline engine, or a natural gas engine.

**Couplings** that describe how systems interact with each other.

*Example:* The car's engine can send rotation to the transmission, and the transmission can send motion to the chassis (by actually turning the wheels).

**Similarities** that can be used to indicate that one system is similar to another in some way.

*Example:* When considering the structural components of a truck, it's easier to say that a truck is like a car instead of describing the same components.

**Variables** that a system might have which affect its behavior.

*Example:* An engine might have a variable called "HoursRun" that keeps track of the total number of hours that the engine has been operating, and this variable might affect the performance or reliability of the engine.

The SES and FDDEVS are specified in logical and mathematical form (see Mittal and Douglass 2011, for background on FDDEVS and Zeigler and Hammonds 2007 for in-depth discussion of the SES). A complete theory of DEVS is given in Zeigler et al. (2000) with key formal properties of well definition, closure under coupling, universality and uniqueness summarized in the Appendix. While the details of the mathematics are transparent to users, it is important to point out that the tool is based on a rigorous mathematical specification with more than thirty years of scrutiny and application. It is this rigor which will provide confidence to stakeholders and engineers that a need expressed in this format is syntactically and formally correct. The formal properties summarized in the Appendix give DEVS checks and balances that allow other models created in the specification to be coupled together correctly. The tool actually prevents users from making logical and syntactical mistakes that might otherwise propagate through to requirements.

In the DEVS formalism, *atomic* DEVS captures the system behavior, while *coupled* DEVS describes the structure of system. The MS4 ME natural language interface automatically generates the mathematics demonstrated here, freeing the system engineer to capture needs quickly yet rigorously. The specification forces you to extract information from the stakeholder in a very efficient manner and distills any need into its fundamental components. In practically, any behavior or function the elements of the specification are required. A well defined need of any kind will contain these atomic elements. In essence, to capture a DEVS model forces you to ask the questions:

- What are the inputs?
- If nothing external happens, what does the system do and when?
- If there is an external input, what does the system do?
- What are the outputs?

In addition, creating a DEVS' **atomic** model forces you to ask questions like:

- What are the possible states?
- In the absence of input, how long does the system stay in each of its states?
- When an input event occurs, how does the system change state?
- After the system finishes its time in a state, what output does it produce and what state does it go to?

Creating a DEVS' **coupled** model forces you to ask questions like:

- What are the components?
- How are the components connected internally?
- How are the components connected externally?
- What are the sub-components?
- What are the interfaces?

---

## 2.3 Jazz Band Continued

The model generated by the Jazz Band sequence diagram of Fig. 2.1 is a coupled model that has as components the actors appearing in the diagram, namely, the BandLeader, Rhythm, Horn, and Reed sections. The SES that is generated is given in natural language form:

```
From the music perspective, JazzBand is made of BandLeader,  
RhythmSection, HornSection, and ReedSection!
```

```
From the music perspective, BandLeader sends PlayBeat to  
RhythmSection!
```

From the music perspective, RhythmSection sends Beat to BandLeader!

From the music perspective, BandLeader sends PlayBrass to HornSection!

From the music perspective, HornSection sends BrassSound to BandLeader!

From the music perspective, BandLeader sends DontPlay to HornSection!

From the music perspective, HornSection sends Quiet to BandLeader!

From the music perspective, BandLeader sends PlayReed to ReedSection!

From the music perspective, ReedSection sends ReedSound to BandLeader!

From the music perspective, BandLeader sends PlayTogether to HornSection!

From the music perspective, BandLeader sends EndInSequence to RhythmSection!

From the music perspective, BandLeader sends EndInSequence to HornSection!

From the music perspective, BandLeader sends EndInSequence to ReedSection!

From the music perspective, ReedSection sends FadeOut to BandLeader!

From the music perspective, HornSection sends FadeOut to BandLeader!

From the music perspective, RhythmSection sends FadeOut to BandLeader!

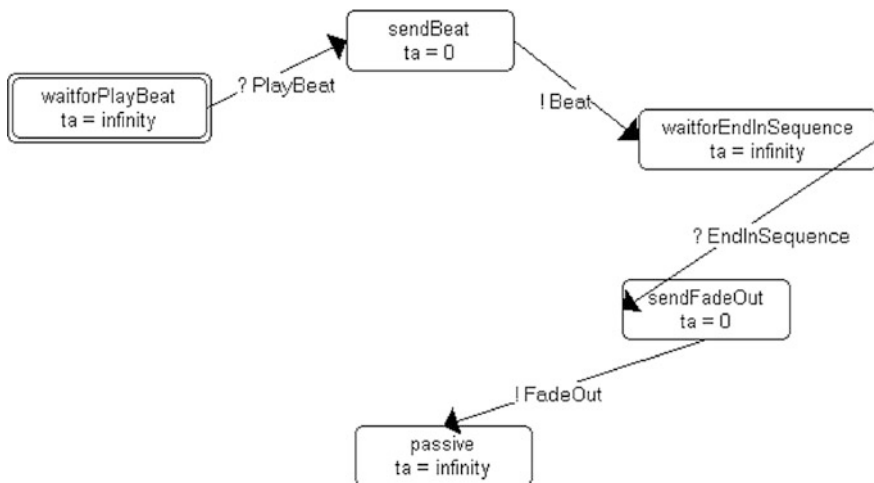
Note that the first sentence lists the components of the model while the remaining sentences describe the message flow broken down into a set of coupling specifications. Each such specification sets up the possibility for a message transmission. For example, the second sentence sets up a coupling of the output port `outBeat` of `BandLeader` to the input port `inBeat` of `RhythmSection`. There is no intrinsic order to the coupling statements of an SES—any permutation will result in the same set of couplings. These couplings are shown as grey lines in the simulation view of Fig. 2.2. In contrast to the sequence of message transmissions specified by the sequence diagram, they describe routing patterns. Indeed, this places the burden on the component behaviors to enact a sequence of transmission events, accordingly, each of the four components, `BandLeader`, `RhythmSection`, `HornSection`, and `ReedSection` in the Jazz Band needs an atomic model to provide the behavior in the manner shown in Fig. 2.2. The FDDEVS natural language forms for these atomic models are automatically generated. That of the `RhythmSection` is shown here:

```

to start,passivate in waitForPlayBeat!
when in waitForPlayBeat and receive PlayBeat go to sendBeat!
hold in sendBeat for time 0!
after sendBeat output Beat!
from sendBeat go to waitForEndInSequence!
passivate in waitForEndInSequence!
when in waitForEndInSequence and receive EndInSequence go to
sendFadeOut!
hold in sendFadeOut for time 0!
after sendFadeOut output FadeOut!
from sendFadeOut go to passive!
passivate in passive!

```

FDDEVS models expressed in natural language have an alternative description in the form of state diagrams. The state diagram for RhythmSection is shown in Fig. 2.9. In this graphical depiction, states are shown as rectangles, each state has a time advance and may have external transitions (input arrows with “?”) and internal transitions with or without outputs (arrows with “!”). The modeler can work in either of the natural language and state diagram equivalent representations and switch between them at will.



**Fig. 2.9** State Diagram view of RhythmSection

The SES is automatically generated as a \*.ses file and deposited in the Models.ses folder while the FDDEVS models are generated as \*.dnl files and deposited in the Models.dnl folder. This makes them available to work with further as you wish to continue to develop the model. For example, the standard hold time for the sendFadeOut state is 0. But we can change the value in the RhythmSection.dnl file to the duration that the component should hold for while fading out, as in:

```
hold in sendFadeOut for time 20!
```

---

## 2.4 Summary

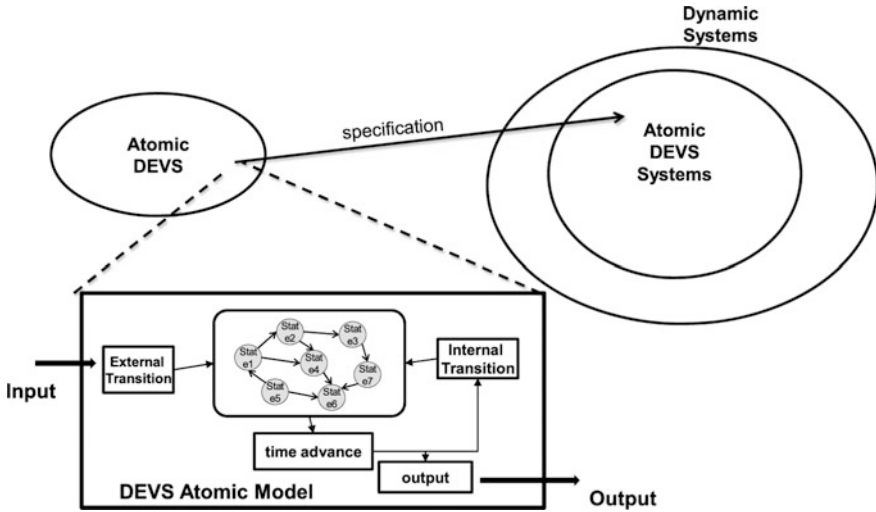
In this chapter, we discussed basic DEVS and SES concepts and tools to support working with these concepts in the context of an actual modeling and simulation environment, the MS4 Modeling Environment (MS4 Me) ([ms4systems.com](http://ms4systems.com)). To address the different perspectives that stakeholders bring to the modeling and simulation world, we provided three different introductions aimed at three different types of users. For the general M&S user, we provided a description of the concepts supported by MS4 Me through the immediate application of its most basic tools. For the M&S Developer, we provided a more advanced introduction to MS4 Me's underlying DEVS concepts and theory and the tools that support them. For the M&S Expert Professional, we offered a glimpse into MS4 Me's features in more depth as well as the theory that supports them.

To summarize, there are two main pillars to the DEVS-based modeling and simulation for Systems of Systems (SoS), the DEVS formalism itself and the SES that enables composition of DEVS models as components. For the composition of components required in the SoS context, the most relevant pillar to start with is the SES. Thus, the next chapter will start the exposition of the SES and its features mentioned above. We will then return to consider the FDDEVS formalism in its natural language form and the enhancements that can be made to be incorporated into Java models in Chap. 4.

### Appendix: Key Formal Properties of DEVS

This appendix summarizes some key formal properties of DEVS as given in Zeigler et al. (2000). These include well definition, closure under coupling, universality and uniqueness. The fact that DEVS stands for Discrete-Event System Specification becomes more apparent from examining Fig. 2.10.

Here, we see an apparent distinction between Atomic DEVS and Dynamic Systems. The set of all Dynamic Systems is taken as a well-defined class in which each system has a set of input time segments, states, state transitions, and output time segments (Zeigler et al. 2000). Although the class of Dynamic Systems is well defined, it is too encompassing and mathematical a concept to allow directly constructing a particular member system. A DEVS atomic model contains the sets (input, states, output) and functions (transition and output) that take the right form

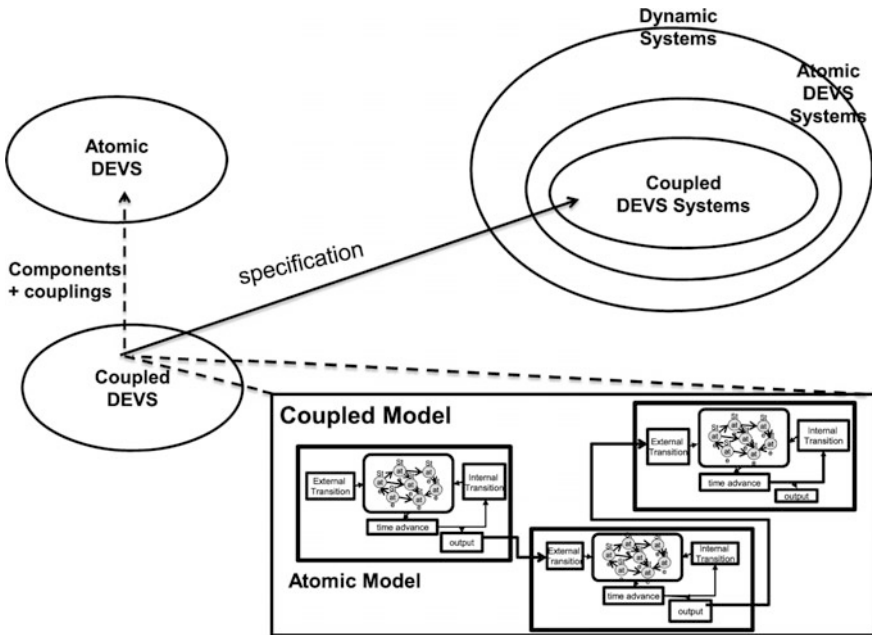


**Fig. 2.10** DEVS Atomic Models as system specifications

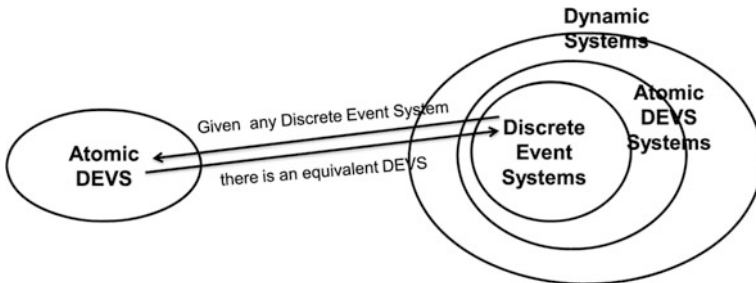
to provide such a construction. The theory shows how the sets and functions should be interpreted to specify a dynamic system and establishes the conditions under which such a specification is well defined, i.e., where there is one, and only one, dynamic system that can be constructed from an Atomic model. In Fig. 2.11, the set of Atomic DEVS Dynamic Systems is the subset of Dynamic Systems that are specified by the set of Atomic Models.

Indeed, the theory shows how DEVS provides a computational framework for working with Dynamic Systems as computational models of real world Systems of Systems. This is further clarified in Fig. 2.11 which shows that DEVS coupled models also define a subclass of Dynamic Systems.

A DEVS Coupled Model constructs a Dynamic System by specifying its components and couplings. The theory shows how the components and couplings should be interpreted to specify a well-defined system. In Fig. 2.11, the set of Coupled DEVS Dynamic Systems is the subset of Dynamic Systems that are specified by the set of Coupled Models. Actually, the theory shows that the subset of Coupled DEVS Systems is contained within the subset of Atomic DEVS Systems. This property is called closure under coupling and states that the dynamic system specified by a coupled model can be represented as (more technically, is behaviorally equivalent to) an Atomic DEVS System. Closure under coupling is important for two reasons: (1) it provides the basis for the Abstract DEVS Simulator, i.e., a simulator is the computational device that carries out the rules by which the components carry out state transitions and send messages to each other through the couplings. (2) It justifies hierarchical composition in which a coupled model (treated as its behaviorally equivalent atomic model) can become components themselves in larger coupled models.



**Fig. 2.11** DEVS Coupled Models as system specifications closed under coupling



**Fig. 2.12** DEVS Universality and Uniqueness for Discrete-Event Systems

The properties of well definition and closure under coupling give you confidence that the models that you construct using a DEVS Modeling Environment are backed up by a solid mathematical and logical foundation. The properties of universality and uniqueness, illustrated in Fig. 2.12, support the claim that any discrete-event model you are likely to want to build, can be done in a DEVS Modeling Environment. First, let's define a Discrete-Event Dynamic System as a Dynamic System with discrete-event input and output segments. The theory shows that DEVS is universal in the sense that any such Discrete-Event Dynamic System is behaviorally equivalent to a DEVS Dynamic System. Moreover, uniqueness says, that there is a



DEVS equivalent system which has the smallest number of states and is essentially contained within any other such equivalent. This means that you are not limited in the range of discrete-event models that you build in a DEVS Modeling Environment. Indeed, you can build any discrete-event model you could build in some other environment. Moreover, if you do not include extraneous and redundant features in it, then it will be the most efficient representative of all the models that could give the same behavior.

---

## References

- DEVS. (2012). DEVS Standardization Group. <http://cell-devs.sce.carleton.ca/devsgroup/?q=node/8>.
- Friedenthal, S., Moore, A., & Steiner, R. (2009). *A practical guide to SysML: the systems modeling language* (1st ed.). San Mateo: Morgan Kaufmann.
- Mittal, S., & Douglass, S. A. (2011). From domain specific languages to DEVS components: application to cognitive M&S. *SpringSim (TMS-DEVS)*, pp. 256–265.
- Ören, T. I. (1984). GEST—a modelling and simulation language based on system theoretic concepts. In T. I. Ören, B. P. Zeigler, & M. S. Elzas (Eds.), *Simulation and model-based methodologies: An integrative view* (pp. 281–335). Heidelberg: Springer.
- Ören, T. I., & Zeigler, B. P. (2012). System theoretic foundations of modeling and simulation: A historic perspective and the legacy of A. Wayne Wymore. *Simulation*.
- Wainer, G. A., & Mosterman, P. J. (2009). *Discrete-event modeling and simulation: Theory and applications*. London: Taylor & Francis.
- Wymore, A. W. (1967). *A mathematical theory of systems engineering: The elements*. New York: Wiley.
- Zeigler, B. P. (1987). Hierarchical, modular discrete-event models in an object oriented environment. *Simulation J.*, 49(5), 219–230.
- Zeigler, B. P., Kim, T. G., & Praehofer, H. (2000). *Theory of modeling and simulation: integrating discrete-event and continuous complex dynamic systems* (2nd ed.). Boston: Academic Press.
- Zeigler, B. P., & Hammonds, P. (2007). *Modeling & simulation-based data engineering: Introducing pragmatics into ontologies for net-centric information exchange*. Boston: Academic Press, 448 pages.

Guide to Modeling and Simulation of Systems of  
Systems

Zeigler, B.; Sarjoughian, H.S.

2017, XIV, 400 p. 216 illus., 141 illus. in color.,

Hardcover

ISBN: 978-3-319-64133-1