Model Abstractions in SeSAm

Franziska Klügl, Örebro University

September 2020

1 Introduction

Metaphors and abstractions are essential for modelling and simulation. They provide a framework of perspectives that a modeller uses to look at a target system, to structure and to eventually formulate the model. It is important that tools that aim at supporting modellers provide appropriate structures – that means appropriate abstractions matching what a modeller expects.

2 Basic Abstractions

Here, we want to explain the basic abstractions available in SeSAm guiding modellers probably more than visual programming. There are three categories of model elements in SeSAm: Agents, Resources and World/Environment:

- The agents are active, they have a state - consisting of a set of variables with values - and behaviour that is described using a kind-of extended activity graph.
- Resources are the inactive parts that are contained in the environmental model. They have a state, but no behaviour. Agents can modify the state of resources.
- The world describes the container for the complete environment, with or without a map. A world is actually a specific agent. That means, one can associate not only global state, but also global dynamics with that world-agent. It is the only agent that is accessible for all "normal" agents.

Figure 1 shows, how those base element categories relate to each other. In fact, "Agent", "Resource", and "World" are generic classes. When the user creates her own agent, she actually creates an agent class from a template as described in the figure.

The base for all those concepts is the so called "SimObject". That is a type of entity that has a body, that can be used to store its current state. In SeSAm, we follow a classical system theoretic approach, assuming that the state of a system (here "SimObject") can be described by values of a collection of variables. Each of those variables has a particular type - it may contain a number (integer or real number - called "double"), a boolean value, a string, a reference to another entity (type "SimObject"). SeSAm also offers complex data types such as lists of hashtables. Later we will shortly discuss composed data types and enumerations. If a variable contains a reference to another entity, that means that the entity that has this variable knows how to reach the other entity. This is for example a way to create links between agents of show that agents "own" or at least know where to find resources. As a variable can be seen as a container for information, the modeller needs to specify, whether an agent can access or even change

CivIng/ABS Course/SeSAm Introduction/abstraction/figs/SeSAmClasses.pdf

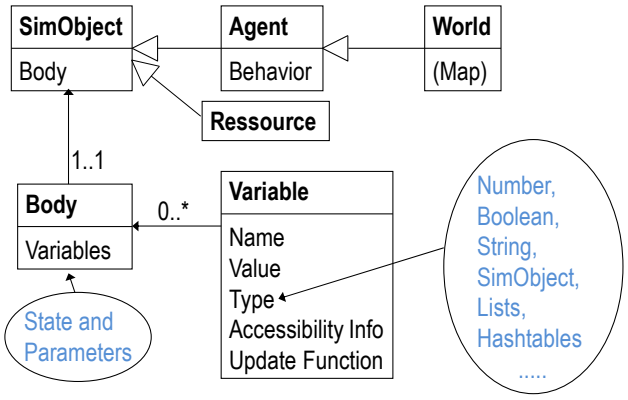

Fig. 1. Basic SeSAm Classes

the values in its variables - the modeller needs to determine whether the variable is accessible from external or whether the variable is dynamic ("writable"). One can also distinguish between variables that the agent can change consciously by its actions (e.g. an energy variable whichs value is increased when the agent is eating) and variables that are changing without action (e.g. an energy value that unconsciously decreases in every update cycle).

As written above and visible in Figure 1, a Resource is nothing more than such a basic entity that just consists of a body. An agent in addition to its state has behaviour. This behaviour is organized in form of an activity graph, like the example shown in figure 2. When an agent is created, it starts its behaviour at the small black circle - the "entry activity". Other nodes represent an activity - kind of "behaviour state". The agent is always in one activity at one time and executes a sequence of actions associated with the activity. Transitions that go away from the activity are based on conditions (if the condition is true, the current activity is stopped and the next one is started.). The world is an agent that in addition to body and behaviour also may possess a map. If this is the case, then agents and resources – for being locate-able on the map – need to possess coordinates, orientation, shape, etc. that fits to the particular map representation.

3 Primitives and JAVA programming

In the behaviour definition (within the nodes of the activity diagrams or on the links between nodes) and many other places (calculating unconscious update of variables, etc), the modeller needs to define function calls that specify actions, conditions or calculate values. We call these functions to be used in the calls "primitives". There are primitives for changing variables, for perceiving the environment, for moving, etc. The effects of those primitives – the return values or that the actions actually change something, are defined by JAVA functions that are connected to these primitives. Each of the primitives

CivIng/ABS Course/SeSAm Introduction/abstraction/figs/ActivityDiagram.pdf

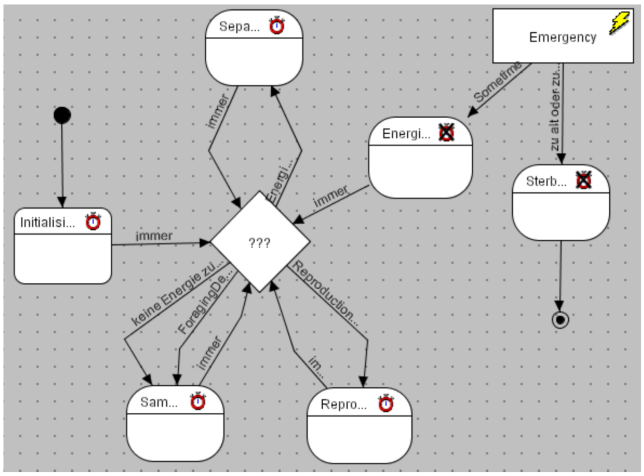

Fig. 2. Screenshot of an activity diagram. Conditions are annotated at transitions between activities. The diamond is a special node for organizing more complex conditions

have input arguments and a return value. The return value of actions is called “void” - that basically means that there is no return value. All other primitive return something. This return type defines where the primitive fits into the specifications. Therefore, the primitives form an important element that make SeSAm syntax-error-proof. A modeller can just select those primitives that make sense in the current context. If you do not find the primitive that you search for, your context may be in a way that you do not expect - click on “Edgar” to find out what context is actually needed or read the tooltips in the primitive list.

Technically seen, a primitive is a wrapper around a JAVA functions. It is quite straight forward to define new primitives that then can be added to an existing SeSAm installation by adding them into the Plugins folder. More information can be found on the SeSAm wikipage (*http://130.243.124.21/mediawiki/index.php/Main_Page*)

4 New Macros

Like in any programming language, one might want to define own functions providing a higher level of abstraction or at least a shortcut for often used combinations of function calls. SeSAm provides “User Functions”. These are kind of macro using existing primitives (or other macros). A modeller basically describes everything what would be in a primitive specification plus code that is formulated based on existing primitives (or other macros): The modeller needs to specify input arguments and output type, as well as a function call in the same manner as with defining conditions or actions in the behaviour definitions.

As with the primitives, the user functions can be used anywhere where the output type fits.

5 Model Specific Data Types

The standard data types that can be used for variables can be found in Table 1.

Activity	It is possible to store an activity in a variable. However this is not used for behaviour generation. The current activity of an agent can be accessed with the primitive `GetCurrentActivity(<agent>).`
Boolean	classical True and False
Color	Colour in RGB (in the GUI, there is a colour picker dialogue). This is only relevant, if space is relevant, that means there is a map on which entities with a particular colour can be drawn. The current color of an entity is however defined in the `SpatialInfo` of the entity.
Hashtable<K><V>	This is a classical key - value table. Values can be accessed via `GetValueForKey(<key>,<hashtable>` and set with `PutValueForKey(<key>,<value>,<hashtable>).` When defining the variable, the data type for the key and for the value has to be set.
Image	For facilitating visualization, one can store a link to a JPG file. The image that is currently used for visualization on the map, is stored in the `SpatialInfo` of the agent or resource.
List<T>	an agent can store a collection of values (same type, needs to be given when specifying)
Number	can take the form “Integer” or “Double”. The latter is the type of real numbers.
Object Class	the names of the specific model classes that the modeller has given for agent classes, resources or world classes.
Position	A variable can store a position, e.g. the current destination of the agent. The current position is represented in the `SpatialInfo` of the entity.
Shape	This is another data type that only makes sense when there is a map and the agent needs to do reasoning about spatial figures and shapes. A shape can be a specific circle, rectangle, line or polygon with specific parameters such as radius, dimensions, sequence of points... The current shape that is used for drawing the agent/resource on the map is specified in teh `SpatialInfo`
SimObject	In such a variable, a reference to another agent or resource can be stored, e.g. to how that they are related, a resource belongs to an agent, etc. The world can always be accessed using the `GetWorld()` primitive.
String	A variable may contain text.

Table 1. Data Types in SeSAm

In addition to those predefined data types, there are two categories of data types that the modeller can define to make the model more readable or to group data that belongs

together. The first type is “Enum”, which stands for Enumeration. Before using this type, it needs to be defined under “User Types”. An enumeration is a set of symbols or text that replace a given set of discrete values - such as *Monday, Tuesday, ...* or *Dead, Alive*. Instead of using integer numbers and remembering what each number means, a modeller can define those values and use them in variables defining the state of an entity.

The other user-defined data type is a composed data type. This is a set of attributes where each attribute is a kind of sub-variable with its specific name and type. There are a number of specific primitives creating, accessing specific attribute values, etc. **It is NOT recommended to change the type of attributes after the type has been used. You might end up with a inconsistent model that cannot be saved or loaded again.**

6 Classes, Instances and Simulation Runs

SeSAm follows a light-weight object-oriented approach with classes and instances, yet without inheritance. Agent, resource and World are the three meta-level classes. When creating a model specific Agent Class, the modeller actually creates a kind of template for structure of such an entity and its general behaviour program. In contrast to programming with a standard object-oriented programming, there is an intermediate step between such a template and an executable object: The “situation” is still a part of the model and contains object instance descriptions as well as a configuration of the map. From that situation an executable simulation run is created. Looking at the user interface of SeSAm, one can find additional elements that a situation can be augmented: so called “analysis”, “simulation”, etc. These are like the “situation” descriptions of what will be used in a simulation run (and will be described in Section 6.2). When a simulation run is generated, SeSAm takes all the description objects (from agent and resource instances, situation description, downwards to the selected “Experiment description” and compiles them into an executable that can be then run within SeSAm. That also means that, if there is a change in the description, the compilation has to be done again.

6.1 Situation as a Start Configuration

The “Situation” is a quite specific element of a SeSAm model. As written above, it collects descriptions of instances. That means, it is a description of an initial configuration from which a simulation run can start. All variables defined in the model specific World, now need to have a value. While one cannot see a map in the World description, the situation contains a concrete map with given dimensions. Agent instances and resource instances are fully configured including a position on the map – on the Agent or resource classes, one just can give a default position for any instance. As a consequence, a modeller can define a number of different situations for a world and select one of them for a concrete simulation run.

6.2 From Configuration to Simulation Run

The minimum requirement for a running simulation is a start configuration. But, for really doing systematic experimentation, more is needed:

- "Analysis" for generating Output Data: There are two ways of storing information that is generated during a simulation run. One can use the "File Plugin" and write any text into a file, or one can use a little bit more systematically, the predefined "Analysis" structures: A single "Analysis" consists of some settings, mainly how often the data shall be taken (every ... update cycle, or when a particular condition is true) and the destination of the output (into a given file, or visualized in a chart or a table). The second part describes the probes that access the data. The later are the "Analysis Items". Each of them is a named calculation, which - when called - returns a number. Depending on the selected output channel, there is a column in the file, a line in the line chart, a block in a block chart or a number in a table for each of these analysis items. The collection of analysis items then goes into the same chart, file or table.
- "Simulation" is a collection of one start configuration (Situation), arbitrary many suitable analysis definitions and a set of conditions that capture simulation states, in which the simulation run should end.
- "Interactive Simulations" is something that can be used for a stable model, replacing the normal simulation run GUI with a more model-specific interface with which the user can observe and interact with during the simulation run. Starting point for an interactive simulation is a fully configured "Simulation". A modeller can define visualization and control items for world, agent and resource classes. There are also some predefined interaction elements such as an animated map. Using those items, the modeller can create a GUI description that is used, once the overall interactive simulation configuration is used for running a simulation.
- "Experiments": If the modeller wants to do repeated simulation runs with systematic changes in the start configurations, etc, then one can use appropriate primitives for programming those repeated runs.

7 Plugins and extending SeSAm

As indicated above, SeSAm is implemented in Java. There are a number of entry points for extending the functionality of SeSAm. The easiest are additional primitives (for example for advanced geometric reasoning, for accessing values from sensors, for connecting to other programs...). One can also add menu items - e.g. loading in data from csv files or GIS files. There is also a plugin for high-level time representations that maps the current update cycle to calendar and hours. There are a number of extensions existing or on the SeSAm Wiki, you can find instructions how to add new functionality. The resulting jar-files need to be placed into the Plugins folder that is contained in the SeSAm folder where you installed SeSAm (and the SeSAm.jar file is). Re-starting SeSAm loads them and makes the functionality available in SeSAm.

There are two default plugins. Particularly important is the "SpatialInfo" plugin for agents and resources and the corresponding "SpatialMap" plugin for the world. This adds space to the model. The relevant information on the level of the map are it's dimensions or whether it is a torus or not. More interesting are the SpatialInfo elements of an entity: They contain the current position, the current orientation ("Direction"), the current Speed. There is a shape with extensions. The Balance Point gives information

which point within the shape is actually the one that corresponds to the position on the map. For example a Balance Point of <5,5> of a Rectangle-shaped resource with extensions of 10 pixes height and width, means that the coordinate of the entity is in the middle of the rectangle. Other information relate to how the entity is drawn (simply filled with color or just with an outline, or whether a jpg-file is drawn for the agent. The Draw Priority gives the sequence with which overlapping entities are visualized.