

Moses Formalism Creation - Tutorial

Kim Mason

Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
kim@tik.ee.ethz.ch

February 9, 2000

Contents

1 Introduction	3
1.1 Additional SDF Concepts	3
1.2 Requirements	4
1.3 Moses Concepts	4
2 Extending the Editor	6
2.1 The Structure of a Graph	6
2.2 Editing the GTDL File	7
2.3 Parts of a GTDL File	8
2.3.1 Edges and Vertices	8
2.3.2 Syntax Predicates	9
2.3.3 Semantic Hooks	9
2.4 The Properties File	10
3 Extending the Hades Simulation Architecture	11
3.1 Simplifying Compilers	11
3.2 The Model Interface	12
3.3 Compiler Output	13
3.4 The Discrete Event Component	13
3.4.1 Inputs and Outputs	14
3.4.2 Scheduling	14
3.5 Running a Component	15
4 Animation	16
4.1 Visibility into the Compiled DEC	16
4.2 Modifying the ModelInterface and DEC	16
4.2.1 Modifications to the Internal DEC	17
A The SDF GTDL File	18
B SDF Editor Properties	22

1 Introduction

Moses is a rich and powerful modeling and simulation environment. Systems in Moses are modeled as cooperating components that can be written in a number of different formalisms. The standard release of Moses supports the following three formalisms:

Time Petri nets A high level Petri net augmented with *time* and *container places*.

Process Networks A very generic form of Kahn's process networks where nodes can contain instances of any component.

Harel State Charts This hierarchical state machine formalism allows control oriented (sub) systems to be defined naturally.

Moses provides interfaces enabling the description and implementation of additional formalisms. Hooks are provided into the Moses editing, simulation and animation environments for this purpose.

This tutorial describes how to implement the Synchronous Data Flow formalism [LM86] (black token) using Moses, and it is assumed that the reader is familiar with the syntax and semantics of an SDF graph before attempting this tutorial. An implementation of SDF including computation is beyond the scope of this tutorial.

This tutorial proceeds in 2 main sections. The first of these is the extension of the editor and simulation architecture to allow the graphical creation and non-graphical simulation of SDF components, but not the animation of these components.

The second section of the tutorial will extend the simulation component to allow animation, and present the additional components required by the Moses animation architecture.

1.1 Additional SDF Concepts

SDF [LM86] implicitly assumes that each arc connecting functional units together contains a FIFO buffer. This tutorial adds the concept of a *Buffer* to SDF. An SDF Arc is made up of a buffer and 2 arcs (in & out). Each arc has a single *weighting*, representing the number of tokens produced or consumed by the function that the arc is attached to (see Figure 1).

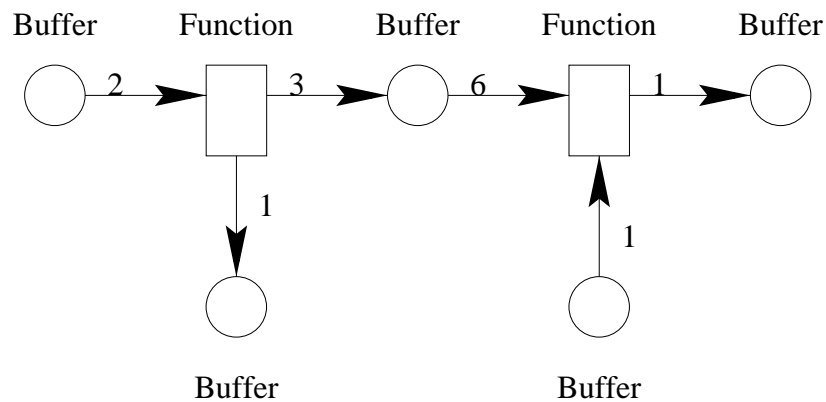


Figure 1: SDF Buffers and Functions

Additionally, the Moses SDF formalism requires inputs and outputs, so that SDF *components* can be created, and composed with components based on other formalisms. An input can produce tokens at any time, and so must be connected exclusively to a single buffer. An output can consume tokens at any time, and must be connected exclusively to a single function. The ability of an input to produce a token at any time is at odds with SDF. Inputs are not scheduled or executed at specific times like functions are, and this makes the weight of the arc coming from the input meaningless.

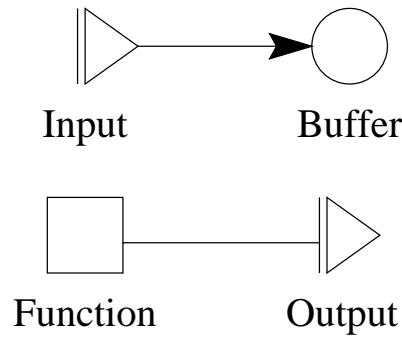


Figure 2: SDF Inputs and Outputs

1.2 Requirements

This tutorial assumes that the user is able to run Moses (this tutorial is based on Moses Alpha 3), and create and simulate simple components, as described in [EJN99]. Additionally, the Moses Alpha 3 source code (or a set of the interfaces required to create all Java components outlined in this tutorial), and a Java development environment supporting the Java 2 platform are required. All source code created for this tutorial is available in the archive `sdfbtutecode.tar.gz`.

1.3 Moses Concepts

The addition of a formalism to Moses requires an understanding of the Moses architecture. The three major parts of the Moses framework that concern the formalism implementor are:

The Editor The Moses editor must be provided with sufficient information to create *graphs* of components defined in the SDF formalism. A *graph* is a generic structure (represented by the class `graph.Graph`) that contains structural and attribute information about the graph, its vertices and edges. Note that the *graph* has no knowledge of the semantics of the graph - it simply contains the structure in a generic way.

The Hades Simulation Architecture The Hades Simulation Architecture [Jan98b] provides a set of interfaces that allow the interrogation and creation of simulation *components* within Moses. An implementation of SDF in the Hades architecture requires the implementation of a *compiler* interface that can produce compiled `DiscreteEventComponent` classes when provided with a `Graph` structure

(possibly created by the Editor). The `DiscreteEventComponent` is the interface that allows execution the simulation, provides input and output methods, and methods for initialising the component.

The Animation Framework The Moses Animation Framework allows Simulation Components (`DiscreteEventComponents`) that are used by the Hades Simulation Architecture to be animated. The formalism implementor must create an `AnimationAdapter` that can receive information from the `DiscreteEventComponent` about its internal state changes.

2 Extending the Editor

We shall start by extending the Moses editor, so that it can create SDF Graphs. This extension is relatively trivial, and requires no Java coding. The editor only requires sufficient information to be able to create a generic graph structure (a `graph.Graph` object), ensure that it is syntactically correct, and display it on screen. When components are composed together (i.e. one component contains another component), the editor must be able to match connectors without instantiating the components.

The editor requires the following information to create and manipulate graphs:

1. A list of attributes for the graph type.
2. A list of each vertex and edge types. This list includes the name of the vertex/edge type, any *attributes* (attribute-type, attribute-name) that the vertex/edge may possess, and a visual description of the vertex/edge (used to render the graph).
3. A set of syntax predicates to be used to check the syntactic correctness of a graph.
4. A way of extracting the input and output connectors of the component. This is only necessary to compose components within the editor.
5. A list of actions to be placed into the editor menu-bar and button bar (e.g. an action to add an input connector). Other information is required such as the images to place on buttons and in menus, and the tool-tips to display on a mouse-over of a button.

All of this information is provided in 2 places - a GTDL [Jan98a] description of the graph, and a corresponding editor properties file (the name of the properties file is given in the GTDL file). The GTDL file is compiled into a `graph.graphType` - `GraphType` object before being used by the Editor for graph creation. The editor inputs and outputs can be seen in Figure 3.

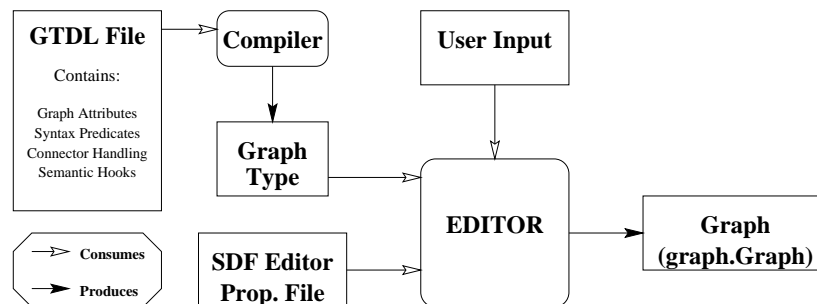


Figure 3: Editor Inputs & Outputs

2.1 The Structure of a Graph

A `graph.Graph` consists of a set of *vertices* and *edges*, each with user specified attributes - (name, value) pairs. Additionally, edges have attributes specifying their source and destination vertices (Figure 4).

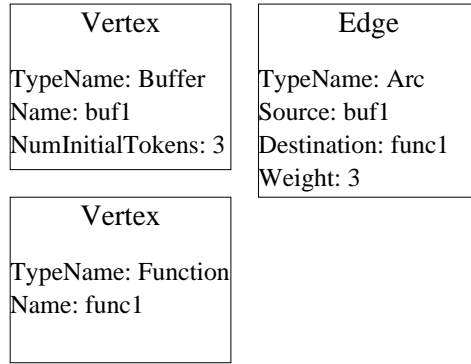


Figure 4: Edges and Vertices

2.2 Editing the GTDL File

First, we shall create the GTDL file (a complete GTDL file for black token SDF is available in Appendix A). Start the Moses tool, using the repository of your choice - “-d <filename>” selects the repository to use when provided on the java command line when starting Moses. This tutorial uses the repository `Basic.rep`.

If the Project Browser is not initially visible in the Moses tool, it can be opened from the File menu. Now select the Formalisms entry in the Project Browser, and right click it so that a contextual menu appears (Figure 5). Choose “New Object” from this menu.

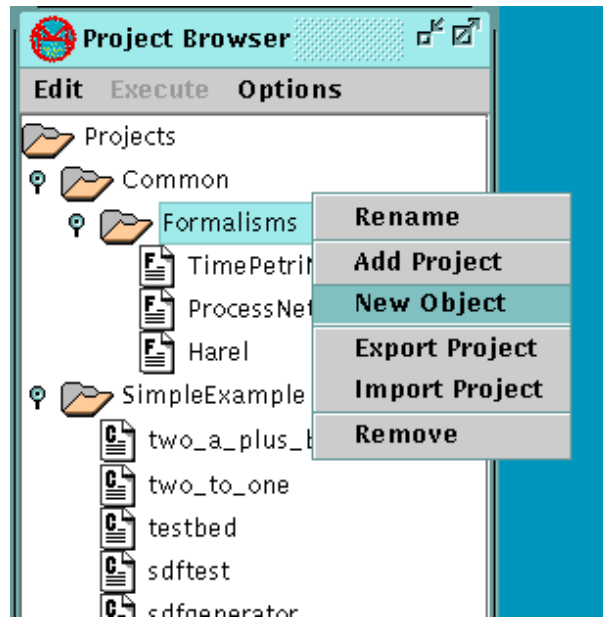


Figure 5:

A dialogue box will appear asking you to choose a formalism type - select `moses-repository.objecttypes.mosesformalism.MosesFormalismAdapter` and select OK. A second dialogue box will now appear (Figure 6). Fill in the “Name in project” field as “SDF”.



Figure 6:

After selecting OK, an SDF formalism will be visible under the Formalisms tree in the Project Browser (the tree may need expanding for the formalism to be visible). It is now time to enter the GTDL. This can be done in 2 ways.

1. Create a file in your favourite text editor with the contents provided in Appendix A. Now select the SDF element under Formalisms in the Project Manager tree. Right click on the SDF element, and choose “Import GTDL”. A file dialog will appear, and you can select your text file containing the GTDL. After the file has been imported, the GTDL can be edited as in point #2 below (however, no further editing is necessary).
2. Select the SDF element under Formalisms in the Project Manager tree. Right click on the SDF element, and choose “Edit GTDL”. Now enter the information in Appendix A. After making any modifications to the SDF GTDL file, ensure that it is checked in by choosing “Check in and Exit” from the Editor’s “File” menu.

2.3 Parts of a GTDL File

We shall now proceed downwards from the top of the SDF GTDL file presented in Appendix A.

The first part of the file (below) identifies the graph as an SDF graph.

```
graph type SDF {
```

2.3.1 Edges and Vertices

Following the graph type, are the declarations of all vertices within the graph. For example, the following declaration identifies a `Buffer` vertex. It has a string attribute called “Name”, and an integer attribute called “NumInitialTokens”. Additionally, descriptors are provided that identify the visual characteristics of a `Buffer`. The string constants provided in each of these descriptors are provided by this implementation of GTDL.

```
vertex type Buffer (string Name, integer NumInitialTokens)
  graphics (hidden string Shape = "Oval",
```

```

hidden color Color = "black",
hidden color FillColor = "yellow",
hidden integer ExtentX = 24,
hidden integer ExtentY = 24).

```

Similarly, the only edge definition in SDF defines its logical and visual attributes.

```

edge type Arc (string Name, integer Weight)
  graphics (hidden string Head = "ClosedTriangle",
            hidden color Color = "black",
            hidden color FillColor = "black").

```

2.3.2 Syntax Predicates

We now come to the syntax predicates embedded in the GTDL file. These predicates are used during editing to ensure the construction of a syntactically legal graph. Each predicate produces a boolean value - for the graph to be valid according to the current predicate, the value must be true. On production of a false value, the string after the name of the predicate is produced as an error. The syntax predicates are defined using the ELAN [Jan99] expression language.

```

predicate OneInputPerBuffer
  "Each buffer is only allowed one input"
  forall e1 in Arc, e2 in Arc :
    if (dst(e1) = dst(e2)) then
      ((e1 = e2) || ~(e1 in Buffer))
    else
      true
    end
  end
end

```

2.3.3 Semantic Hooks

The final section of the SDF GTDL file contains information that the Moses tool uses when editing, compiling, simulating and animating SDF graphs.

EditorProperties The properties file used by the editor. This file contains information such as menu-action bindings, button-action bindings, button-tooltip bindings, and button-icon bindings.

Compiler The fully qualified name of a class that implements the `moses.models.translators.ModelCompiler` interface. The creation of this class is described in Section 3. This class is used by Moses to compile a `graph.Graph` object (usually created by the Editor) into Java code (this code is then compiled, and the resulting class can be instantiated and used by the simulator as a component).

AnimationAdapter The fully qualified name of a class that implements the `moses.animation.AnimationAdapter` class. The creation of this class is described in Section **To Be Done**. This class is used by Moses as an interface to the component while it is being executed.

2.4 The Properties File

The `EditorProperties` file mentioned in the preceding section must now be created. This file is listed in full in Appendix B. The file must be given the name associated with the `EditorProperties` attribute in the semantic hooks section of the `GTDL` file, but with `.properties` appended. This properties file must be placed within the class-path. A description of the entries within this file are available within the comments in the file. The file simply associates menu items and toolbar buttons with actions, tooltips and icons.

3 Extending the Hades Simulation Architecture

The implementation of SDF using the Hades Simulation Architecture requires the creation of 1 entity to implement a formalism. This is the *Compiler*, referred to in the Semantic Hooks section of the SDF GTDL file. The compiler implements the `moses.models.translator.ModelCompiler` interface, and solely provides the following method:

```
public void compile(Graph ingraph,  
                    ModelCompilerParameters pars);
```

This method simply compiles `ingraph` into Java source code, and then compiles this source (using the Java Compiler). The class produced implements the `hades.DES.DiscreteEventComponent` interface (see Figure 7. As described in [Jan98b], this interface is the basis for all simulation. The `ModelCompilerParameters` parameter only contains simple options (e.g. deletion of Java source).

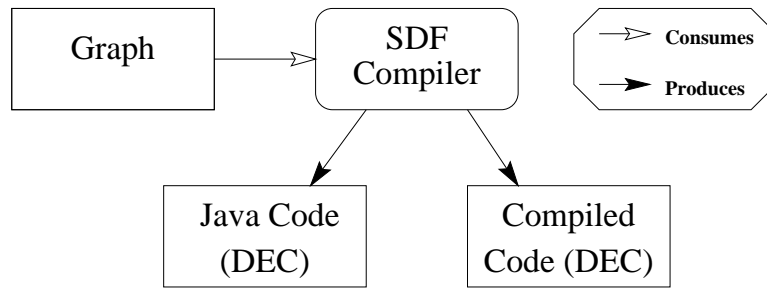


Figure 7: Compiler Inputs & Outputs

3.1 Simplifying Compilers

The above requires the formalism implementor create a compiler class that will completely perform the compilation of a `graph.Graph` into a `DiscreteEventComponent`. We obviously do not wish to implement the full functionality of this compiler for every formalism that we implement in Moses. For this reason, the `moses.models.translator.generic.GenericModelCompiler` class has been supplied. We use this class as a superclass for our compiler implementation, and it does much of the work for us. It provides the following simple constructor.

```
public GenericModelCompiler(  
    moses.runtime.models.ModelInterface mi)
```

When provided with a `ModelInterface` as its constructor parameter, the `GenericModelCompiler` provides complete compiler functionality. All that we must do is subclass `GenericModelCompiler`, and provide its constructor with a `ModelInterface`. Our new subclass is the compiler class referred to in the Semantic Hooks section of the SDF GTDL file.

We now wish to create a class that implements `moses.runtime.models.ModelInterface`, and to instantiate and use this class as a parameter for the `GenericModelCompiler` constructor.

3.2 The Model Interface

The `ModelInterface` is a factory for Hades `DiscreteEventComponents`, and other formalism specific information related to Graphs. The primary information produced by `ModelInterfaces` is represented in Figure 8.

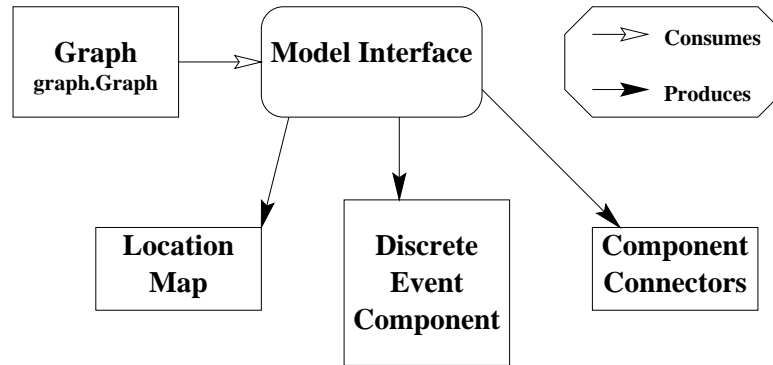


Figure 8: Model Interface Inputs & Outputs

Some methods for the extraction of this information are implemented in the `moses.runtime.models.AbstractModelInterface` class, which should be extended when implementing a formalism specific `ModelInterface`. This abstract class provides support for standard functionality including:

- Extracting information from a graph involving its corresponding components identity, interfaces, and other trivially extracted information.
- Describing the capabilities of the model - by default, only the most simple capabilities are described.
- Utility methods to create a *location map*. A location map is a Java `Map` that maps vertices and edges in the graph to a list of unique integer identifiers. Each integer identifier represents another *location* in the graph. This is useful for identifying individual “locations” within a component, and to map the location to a vertex or an edge.

The methods that remain to be implemented in `ModelInterface` are as follows:

```
public DiscreteEventComponent instantiate(Graph g,
                                         Environment env, Map locMap);
```

This method instantiates a `DiscreteEventComponent` from information usually created by the editor (`Graph`, `Environment` and `LocationMap`). In the case of the `SDFBTModelInterface`, the `DiscreteEventComponent` returned will be an instance of a custom DEC class (created by the formalism implementor). The creation of this class is discussed in Section 3.4.

```
public Set getAtomicConnectors(Graph g);
```

Simply returns a set of `AtomicConnectorDescriptors` describing the input and output connectors of the component specified by `g`.

```
public Set getInterfaces(Graph g);
```

Returns a set of compound interfaces (currently un-implemented functionality - this should return an empty Set).

```
public Map createLocationMap(Graph g);
```

Creates a map of locations, as mentioned above.

3.3 Compiler Output

Having seen that the Compiler depends only upon the Graph that represents a component, and its formalism's ModelInterface, we may now ask how the DiscreteEventComponent class that is created by the GenericModelCompiler is produced.

The compiled DiscreteEventComponent class simply contains a serialised copy of the graph.Graph that describes the component, and constructor code that will create a ModelInterface, and use its createLocationMap and instantiate methods to create the DEC that is stored internally. All calls to the enclosing DEC are then passed through to the internally stored DEC (see Figure 9).

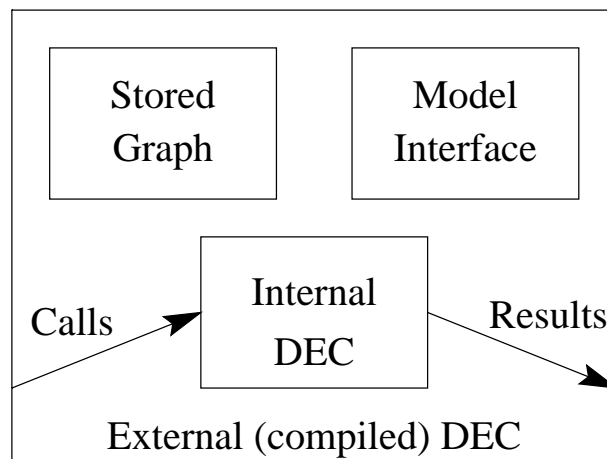


Figure 9: Compiled DEC

Future versions of Moses will store an XML representation of the Graph instead of a serialised instance. This will allow extension of the Graph class, without breaking existing DiscreteEventComponents.

3.4 The Discrete Event Component

All of the methods that we must implement from ModelInterface are fairly simple, with the exception of the instantiate method, which must create a DiscreteEventComponent.

It is up to the formalism implementor to create this class, and instantiate it using the methods of their choice. Essentially, this DiscreteEventComponent must

be provided with all relevant information contained within the `graph.Graph` that is required for the execution of the component, including the internal structure of the component, and all associated state.

An `AbstractDiscreteEventComponent` is provided to simplify connector handling, and implements some of the DEC methods. Our DEC must implement the remaining 2 methods:

```
public void initializeState(double t, Scheduler s)
```

Initialises the component at time `t`. The component is able to schedule any `EventProcessors` with the `Scheduler`.

```
public boolean isInitialized();
```

Simply returns true if this component has been initialised.

3.4.1 Inputs and Outputs

`DiscreteEventComponents` use message listeners and message producers for their token input/output. Every input has an associated listener (a `MessageListener`), and every output an associated producer (a `MessageProducer`). When components are composed together in Moses, input listeners register themselves with the corresponding output producers (see Figure 10). It is up to the formalism implementor to create custom `MessageListener` subclasses that execute the necessary behaviour on receipt of a token from a producer.

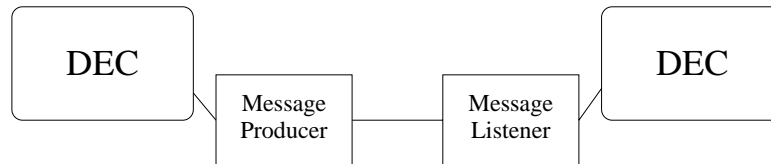


Figure 10: Producers and Listeners

3.4.2 Scheduling

The `Scheduler` allows the continued execution of our DEC, and a reference to it should be stored by the DEC. The `Scheduler` allows `EventProcessors` to be scheduled at any time. `EventProcessors` contain a `processEvent(double time)` method that is called by the scheduler when it is time for them to execute, and this method must be over-ridden by the formalism implementor. `EventProcessors` are then scheduled whenever some execution of the DEC is required. This can include:

- The initialisation of the DEC (in SDF, some functions may be able to fire).
- The receipt of a token in a `MessageListener`. For example, in SDF, the receipt of a token may enable some functions, and so `EventProcessors` that fire the functions can be scheduled.
- The execution of an existing `EventProcessor` may change the state of the component such that a new `EventProcessor` needs to be scheduled.

There are 2 major ways to schedule EventProcessors. The first is to have 1 major EventProcessor, and to schedule it when any execution is required. This EventProcessor needs to be able to modify the state of the DEC appropriately under any valid conditions. A second method is to use small, task specific EventProcessors that are scheduled to execute 1 small task (such as the firing of a single function). The interactions between the DEC, Scheduler, EventProcessors, and inputs and outputs can be seen in Figure 11.

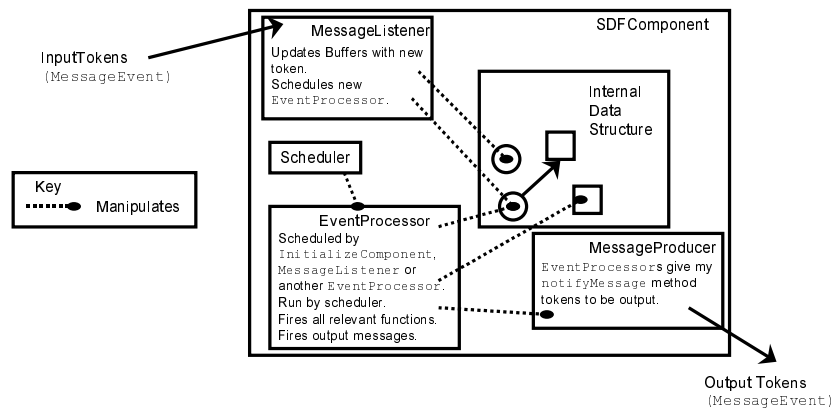


Figure 11: DEC Interactions

3.5 Running a Component

After your SDF Black Token formalism (without Animation) has been added to mooses, it is possible to edit and save SDF graphs, and also compile them. Running them can be achieved simply by embedding them in a component that is built using an animatable formalism (i.e. time petri nets), and then running the enclosing component. Although you cannot view your SDF components running, you can view the inputs to and outputs from your SDF components.

4 Animation

We now wish to Animate our SDF DEC. The class that handles animation is named by the `AnimationAdapter` item in the SDF GTDL file, which must implement the `moses. animator. AnimationAdapter` interface.

The `AnimationAdapter` must be able to extract the `graph. Graph` from the compiled DEC, and receive some kind of event for each vertex/edge whenever the state of that vertex/edge changes within the component. Specifically, we want to extract a `hades. DES. util. StateChangeListener` for each vertex and edge that can undergo an animatable state change. The `StateChangeListener` interface allows `StateChangeListeners` to register to receive `StateChangeEvents`.

4.1 Visibility into the Compiled DEC

Visibility into the compiled `DiscreteEventComponent` is only enabled by the `GenericModelCompiler` when the corresponding `ModelInterface` used by the compiler returns `true` from its `supportsAnimation()` method.

When the `supportsAnimation()` method returns `true`, the compiled DEC class is made to implement the `moses. animator. Animatable` interface, as well as the `DiscreteEventComponent` interface. The `Animatable` interface implements the `hades. DES. util. LocationMap` interface, and the `moses. components. SourceProvider` interface.

Section 3 introduced the *location map*, which is a mapping of `Edges` and `Vertices` to *locations* (usually `Integers`). The `SourceProvider` interface allows the extraction of the source (a `graph. Graph`, in this case), and the *location map* from the DEC. The `LocationMap` interface allows each of the *locations* with the *location map* to map to an `Object`. Note the distinction between the *location map* (a Java `Map` of `Edges` and `Vertices` to *locations*), and the `LocationMap` interface, which allows *locations* to be mapped to `Objects` (see Figure 12).

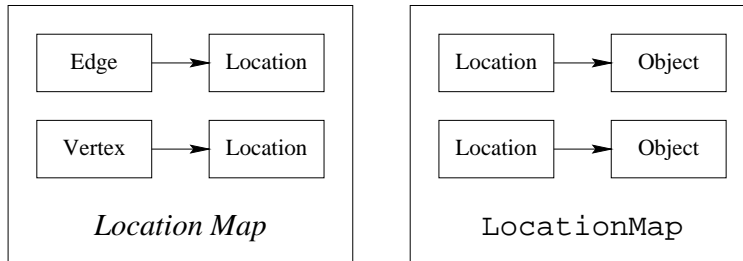


Figure 12: Location Maps

The `Objects` returned by the `LocationMap` are `StateChangeProviders`, which our `AnimationAdapter` can use.

4.2 Modifying the ModelInterface and DEC

Modification of our `SDFBTModelInterface` is trivial. We must over-ride the `supportsAnimation()` method to return `true`.

4.2.1 Modifications to the Internal DEC

We must now modify the `DiscreteEventComponent` that is returned by the `instantiate` method.

Moses allows us to associate a `StateChangeProvider` with each *location* derived from a `Graph`. The association of a `StateChangeProvider` with each *location* in the `Graph` is realised by making our `DiscreteEventComponent` implement the interface `hades.DES.util.LocationMap`. `LocationMap` provides the following methods:

```
public Object getContent(Object location);
```

```
public Set locations();
```

Our internal DEC doesn't need to store the *location map* or the `Graph`. All of this is taken care of by the surrounding compiled DEC.

Animation Adapter - TOBEDONE.

A The SDF GTDL File

```
//
// Synchronous Data Flow Graphs
// Initially, simply uses black-token (no computation).
//
// authors: Kim Mason (kim@tik.ee.ethz.ch)
//

graph type SDFBlackToken {

//-----
// Graph Attributes
//-----

attribute text Comment.

//-----
// Vertex types
//-----

vertex type Buffer (string Name, integer NumInitialTokens)
    graphics (hidden string Shape = "Oval",
              hidden color Color = "black",
              hidden color FillColor = "yellow",
              hidden integer ExtentX = 24,
              hidden integer ExtentY = 24).

vertex type Function (string Name)
    graphics (hidden string Shape = "Rectangle",
              hidden color Color = "black",
              hidden color FillColor = "orange",
              hidden integer ExtentX = 24,
              hidden integer ExtentY = 24).

vertex type InputConnector (string Name)
    graphics (hidden string Shape = "InputTriangle",
              hidden color Color = "black",
              hidden color FillColor = "lightGray",
              hidden integer ExtentX = 24,
              hidden integer ExtentY = 24).

vertex type OutputConnector (string Name)
    graphics (hidden string Shape = "OutputTriangle",
              hidden color Color = "black",
              hidden color FillColor = "lightGray",
              hidden integer ExtentX = 24,
              hidden integer ExtentY = 24).
```

```

vertex type Comment ()
  graphics (text Text = "Comment",
           hidden color TextColor = "black").

//-----
// Edge types
//-----

edge type Arc (string Name, integer Weight)
  graphics (hidden string Head = "ClosedTriangle",
           hidden color Color = "black",
           hidden color FillColor = "black").

//-----
// Syntax Predicates
//-----

predicate NamedInputs "Inputs must have a name."
  forall v in InputConnector :
    let nm = v("Name"):
      (nm <> null) && (nm <> "")
    end
  end

predicate NamedOutputs "Outputs must have a name."
  forall v in OutputConnector :
    let nm = v("Name"):
      (nm <> null) && (nm <> "")
    end
  end

predicate WeightedArcs "Arcs must have a weight."
  forall e in Arc:
    let wt = e("Weight"):
      (wt<>null)
    end
  end

predicate InputConnectorToBuffer
  "Input Connectors must be connected to buffers"
  forall e in Arc:
    if (src(e) in InputConnector) then
      dst(e) in Buffer
    else
      true
    end
  end

predicate FunctionToBufferOrOutputConnector

```

```

"Functions must be connected to buffers or output connectors"
forall e in Arc:
  if (src(e) in Function) then
    ((dst(e) in Buffer) || (dst(e) in OutputConnector))
  else
    true
  end
end
end

```

```

predicate BuffertoFunction
"Buffers must be connected to functions"
forall e in Arc:
  if (src(e) in Buffer) then
    dst(e) in Function
  else
    true
  end
end
end

```

```

predicate OutputConnector
"Output connectors must not be connected to anything"
forall e in Arc:
  if (src(e) in OutputConnector) then
    ~(dst(e) <> null) && (dst(e) <> "")
  else
    true
  end
end
end

```

```

predicate OneOutputPerBuffer
"Each buffer is only allowed one output"
forall e1 in Arc, e2 in Arc :
  if (src(e1) = src(e2)) then
    ((e1 = e2) || (~(e1 in Buffer)))
  else
    true
  end
end
end

```

```

predicate OneInputPerBuffer
"Each buffer is only allowed one input"
forall e1 in Arc, e2 in Arc :
  if (dst(e1) = dst(e2)) then
    ((e1 = e2) || (~(e1 in Buffer)))
  else
    true
  end
end
end

```

```
//-----  
// Semantic Hooks  
//-----  
const EditorProperties=  
  "SDFBT_Graph".  
const Compiler=  
  "moses.models.translator.sdfbt.SDFBTCompiler".  
const AnimationAdapter=  
  "moses animator.sdfbt.SDFBTAnimationAdapter".  
}
```

B SDF Editor Properties

Note - The symbol □ indicates that the next line should be appended to the current line (the line break should be removed).

```
# @(#)SDF_Graph.properties      1.0 18-12-99
#
# Resource strings for a very simple class of black
# token SDF.

# menubar definition
#
# Each of the strings that follow form a key to be
# used to the actual menu definition.

menubar=insert

# insert Menu definition
#
# Each of the strings that follow form a key to be
# used as the basis of a menu item definition.
# dashes are used to insert dividers into the menu
#

# the following line defines the menu entries
insert=comment - arc - buffer function inputconnector □
outputconnector

# return the actual string representing the menu
insertLabel=Insert

# for each menu entry three strings can be defined two of
# which are obligatory:
# xxxLabel returns the actual string representing the menu entry
# xxxAction returns the actual string representing the command.
# It is assumed that all actions are or the form of the Vertex
# or Edge name + "Action" e.g. PlaceAction
#
# xxxImage returns the actual string representing the image
# to be used to represent the menu entry (optional)

commentLabel=Comment
commentImage=images/petri-images/comment.gif
commentAction=CommentAction

inputconnectorLabel=Input Connector
inputconnectorImage=images/petri-images/inputconnector.gif
inputconnectorAction=InputConnectorAction

outputconnectorLabel=Output Connector
```

```

outputconnectorImage=images/petri-images/outputconnector.gif
outputconnectorAction=OutputConnectorAction

bufferLabel=Buffer
bufferImage=images/petri-images/place.gif
bufferAction=BufferAction

functionLabel=Function
functionImage=images/petri-images/trans.gif
functionAction=FunctionAction

arcLabel=Arc
arcImage=images/petri-images/arc.gif
arcAction=ArcAction

# toolbar definition
#
# Each of the strings that follow form a key to be
# used as the basis of the tool definition. Actions
# are of course sharable, and in this case are shared
# with the menu items.
# dashes are used to insert extra space between buttons

toolbar=^ - comment - arc - buffer function inputconnector □
outputconnector

# for each menu entry a string can be defined
# xxxTooltip returns the actual string representing
# the tooltip to be shown

commentTooltip=Insert a comment
bufferTooltip=Insert a buffer
functionTooltip=Insert a function
arcTooltip=Insert an arc
inputconnectorTooltip=Insert an input connector
outputconnectorTooltip=Insert an output connector

```

References

- [EJN99] Rob Esser, Jörn Janneck, and Martin Naedele. *The Moses Tool Suite - A Tutorial*. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092 Zurich, Switzerland, 1.01 edition, Nov 1999.
- [Jan98a] Jörn Janneck. Graph Type Definition Language. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092 Zurich, Switzerland, Oct 1998.
- [Jan98b] Jörn Janneck. HADES - Hierarchical Architecture for Discrete Event Simulation. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092 Zurich, Switzerland, Oct 1998.
- [Jan99] Jörn Janneck. Expression language elan specification 0.9. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092 Zurich, Switzerland, Mar 1999.
- [LM86] E.A Lee and D.G Messerschmitt. Synchronous Data Flow: Describing DSP Algorithms. In Kung, Owen, and Nash, editors, *VLSI Signal Processing II*, pages 373 – 384. IEEE Press, 1986.
- [LM87] E.A Lee and D.G Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, Sept 1987.