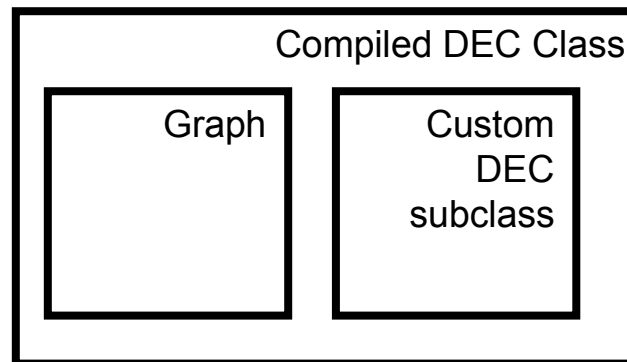

Implementing SDF in the Moses Simulation and Animation Architectures.

A 2 Stage Introduction

Parameterised DECs.



- ◆ We wish to create `DiscreteEventComponents` using a Graph (produced by the Editor) as a parameter.
- ◆ Options:
 - Create a subclass of DEC that takes a Graph as a parameter when instantiated.
 - » This means that our DEC is dependant on the Graph object.
 - » We wish to store a different DEC *class* for each SDF component.
- ◆ Solution



The SDF GTDL File



- ◆ The SDF GTDL file has a *Semantic Hooks* section.
 - Several hooks into Moses are outlined here.
- ◆ **Compiler hook**
 - `const Compiler="moses.models.translator.sdf.SDFCompiler"`
- ◆ **AnimationAdapter hook**
 - `const AnimationAdapter="moses animator.sdf.SDFAnimationAdapter"`
 - We will ignore the AnimationAdapter hook for now.
 - » We only wish to create simulation components first.
 - » Animation capability can wait.
- ◆ The Compiler hook specifies a class can compile Graph objects into Java classes implementing `DiscreteEventComponent`.
 - `DiscreteEventComponent` is the basis of the simulation.

Compiler and Animator Hook



SimpleExample

- two_a_plus_b_sqr
- two_to_one
- testbed
- sdfest
- sdfgene
- onetoor

Edit Properties

View

Edit

Unlock

Compile

Animate

Import internal format

Import RoOD

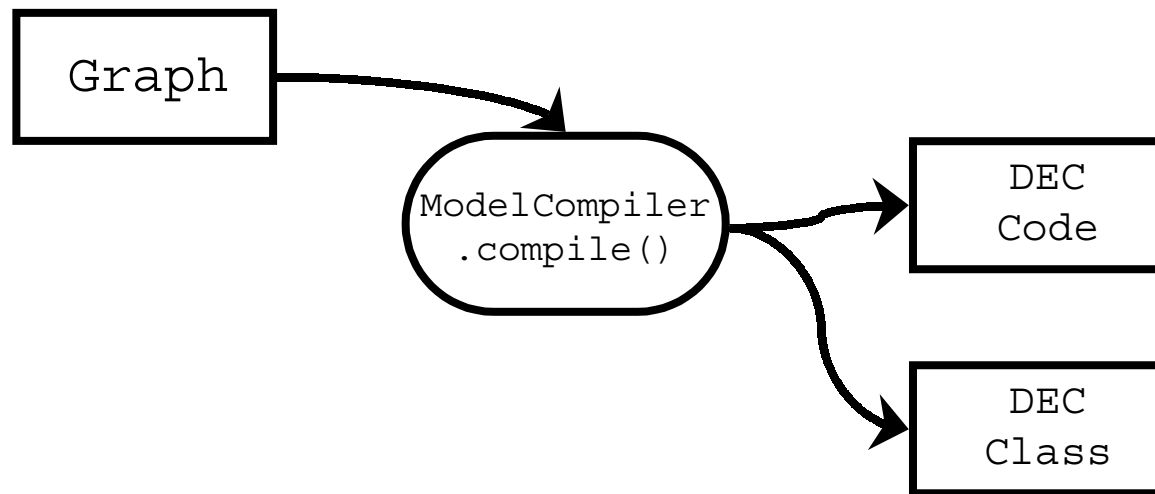
Uses Compiler Hook

Uses Animator Hook

The Compiler



- ◆ The Compiler must implement
 - `moses.models.translator.ModelCompiler`
- ◆ This interface has the method
 - `public void compile(Graph g, ModelCompilerParameters pars);`
- ◆ `ModelCompilerParameters` specifies the directories to place the resulting DEC's Java code, and compiled class files.



Easier Compilation



- ◆ We don't wish to write our own compiler.
 - We have a `GenericModelCompiler`.
- ◆ `GenericModelCompiler` uses another formalism specific class.
 - This class implements `moses.runtime.models.ModelInterface`
- ◆ The compiler is instantiated by (constructor):
 - `ModelCompiler(ModelInterface mi)`
- ◆ After compiler instantiation, the `compile` method can be called.
 - No other Compiler setup!

Why Specify the Compiler?



- ◆ Question!
 - Why don't we specify the `ModelInterface` in the semantic hooks section of the GTDL file instead of the `Compiler`?
 - » We could then use our `GenericModelCompiler` automatically
 - » Oops! This is coming. We *should* be doing this.
 - » However, you may wish to re-code the compiler (efficiency reasons).
- ◆ There will be an option of providing a `Compiler` or a `ModelInterface` in the semantic hooks section.

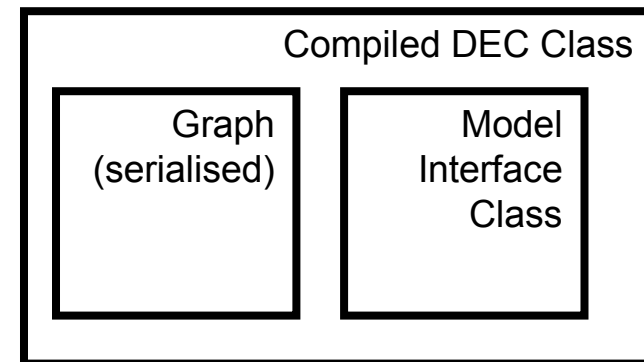
Full SDF Compiler Code!



```
package moses.models.translator.sdf;
import moses.models.translator.generic.GenericModelCompiler;

public class SDFModelCompiler extends GenericModelCompiler {
    public SDFCompiler() {
        super(new moses.runtime.models.lib.SDFModelInterface());
    }
}
```

- ◆ The compiler really produces



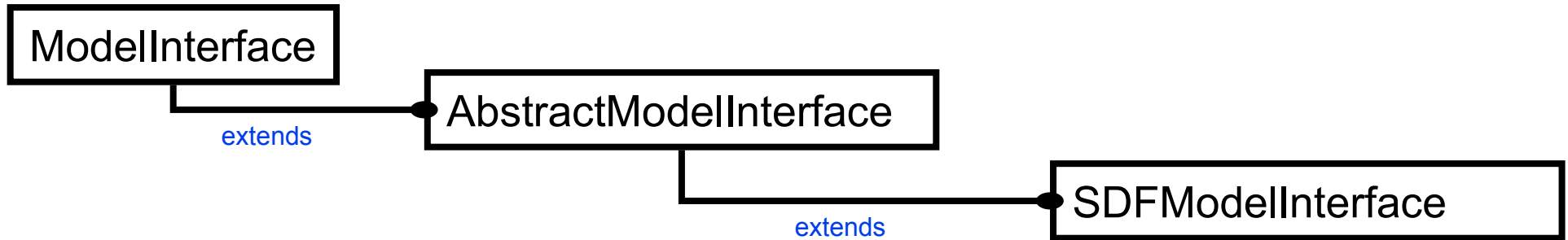


- ◆ We can now ignore the compiler.
 - ModelInterface is our primary interest.
- ◆ The ModelInterface is a factory for DiscreteEventComponents.
 - Almost all methods take a graph.Graph as a parameter.
 - Other functionality:
 - » Also provides some standard information (I.E. graph name, graph superclass, graph ComponentSpecification etc..).
 - ◆ ComponentSpecification contains information about input guards, input abstractstatefunctions, and inputfunctions.
 - » All of these things are trivially created from the graph.

AbstractModelInterface



- ◆ We have an AbstractModelInterface that implements all of the standard (trivial) functionality.



Non Trivial Methods (ModelInterface)



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

- » This method creates a locationMap from the graph. A location map maps each vertex and edge to a list of *locations*.
- » A *location* is simply a unique value within the entire graph (an Integer).
- » A non-pointer based method of discriminating between vertices and edges.
- » Mainly used by the Animation framework.

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

- » This is the method that produces a DiscreteEventComponent from the graph.
- » Note that the location map has been filled out before being passed to this method.

Non-Trivial Methods (ModelInterface)



- ◆ `public Set getAtomicConnectors(Graph g)`
 - Gets a set of all input and output connectors from `g`.
 - » A set of `AtomicConnectorDescriptor`.
 - Trivial for SDF (get inputs and outputs, and put their name into a descriptor).
- ◆ `public Set getInterfaces(Graph g) { return new HashSet(); }`
 - Used for aggregated interfaces (Jörn's Centronics example).
 - » SDF doesn't support this.
- ◆ `public boolean supportsAnimation() { return false; }`
 - returns true if this component supports animation
 - » we don't want to (yet).

Instantiate (ModelInterface).



- ◆ `instantiate` is the only substantial method for black token SDF.
 - `instantiate` must create a `DiscreteEventComponent` that can simulate the SDF graph in the `Graph` parameter.
 - This DEC class must be implemented by the formalism implementor.
 - » We will call our class `SDFComponent`.

Instantiate (ModelInterface).

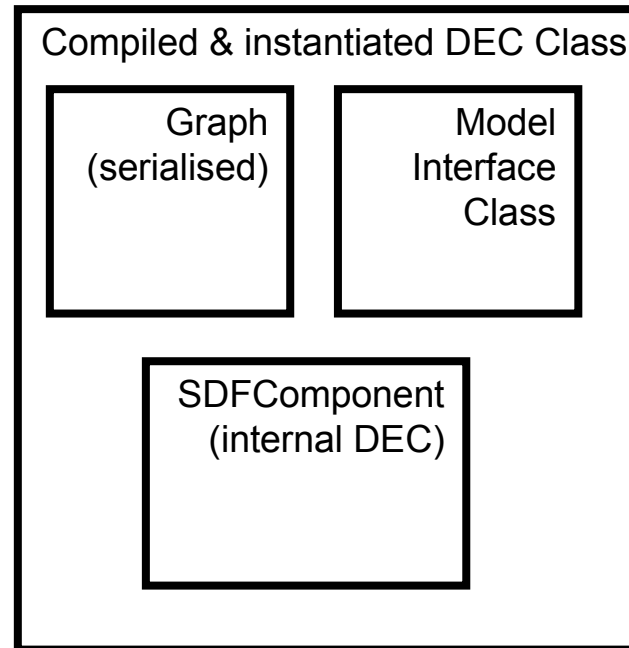


- ◆ What does `instantiate` do internally?
 - Simply creates an `SDFComponent` (`new SDFComponent ()` - an implementor of `DEC`).
 - In one pass, adds all vertices to the `SDFComponent`.
 - » Uses arbitrary methods to do this. The `SDFComponent` must create corresponding internal data structures representing the vertices and all associated vertex state.
 - In the second pass, adds all edges to the `SDFComponent`.
 - » These 2 passes are necessary, as the edges refer to the vertices.
- ◆ Our `SDFComponent` provides utility methods that the `ModelInterface` can use to add buffers, functions, arcs, etc...
- ◆ It is up to the `SDFComponent` to manage all internal data structures representing the SDF network.

Instantiation of Compiled DEC class.



- ◆ When our compiled DEC class is instantiated, an SDFComponent is created within it.
 - All calls to the compiled DEC class are passed to the internal SDFComponent.

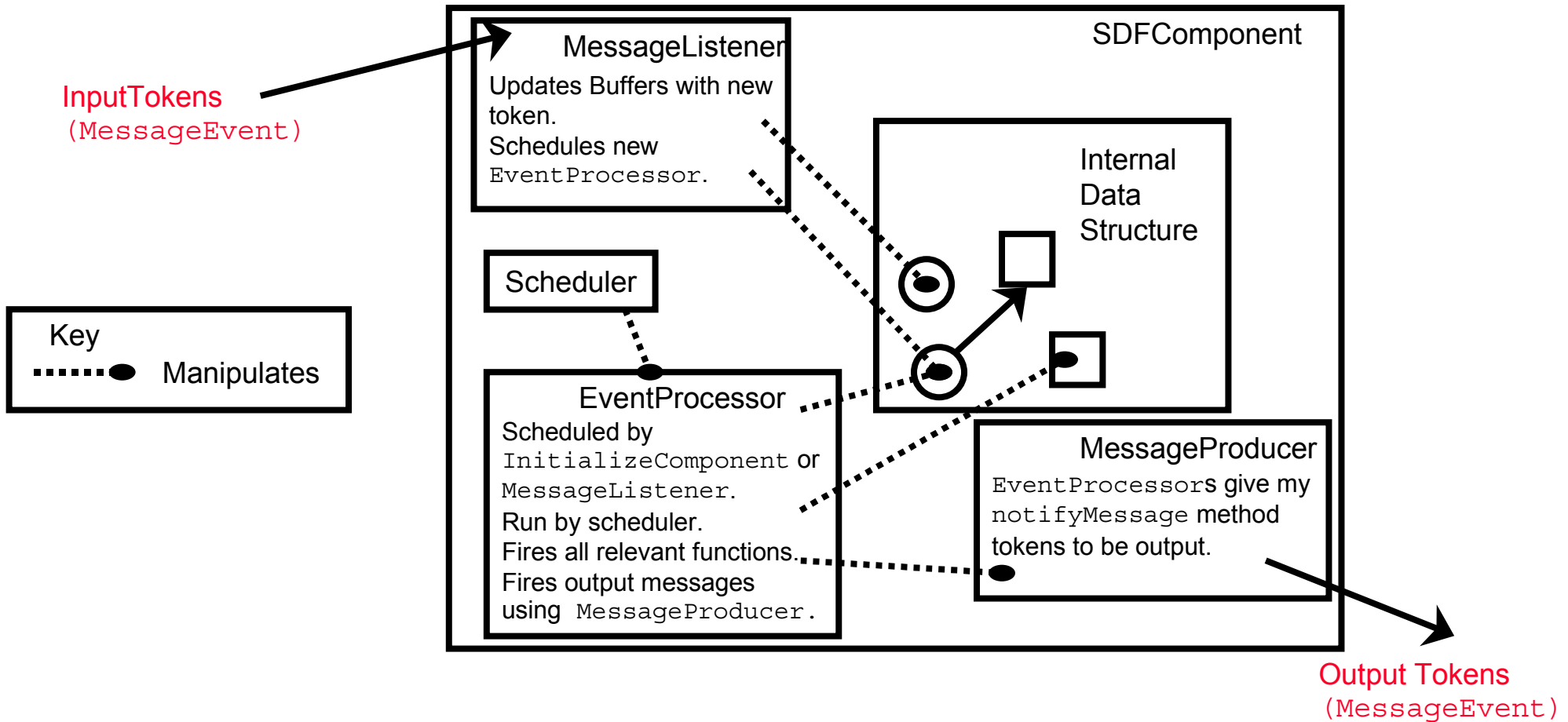


SDFComponent



- ◆ Like all DEC classes, our SDFComponent has
 - access to the scheduler (a parameter) in its initializeState function.
 - » This allows it to initially schedule an EventProcessor.
 - ◆ SDFComponent may be initialised with activated functions, and immediately be able to fire them.
 - inputs and outputs.
 - » On receipt of an input token, a MessageListener is activated.
 - ◆ This listener is able to schedule further EventProcessor objects.

SDFComponent





- ◆ Our SDFComponent implements the DiscreteEventComponent interface.
 - Important methods are
 - » `initializeState(double time, Scheduler s);`
 - » `isInitialized();`
 - » Connector handling methods (implemented by the `AbstractDiscreteEventComponent`), which `SDFComponent` subclasses.
- ◆ Also implements `EventProcessor`.
 - Not necessary. It could create `EventProcessor` classes and pass them to the scheduler (fire and forget).

DEC Building Methods



- ◆ Our SDFComponent provides additional methods to aid in its creation.
 - `public void addFunction(Object key, String name);`
 - `public void addBuffer(Object key, int numtokens, String name);`
 - etc...
- ◆ This is all that is needed to implement the compilation of DEC's from Graphs.

Stage 2 - Animation



- ◆ We wish that our `DiscreteEventComponent` be animatable.
 - Moses requires visibility into the DEC while it is running.
- ◆ `SDFComponent` has an internal set of objects representing functions and buffers.
 - The Moses animator needs to be notified when these objects change state.
 - » The animator can re-draw the corresponding graph vertex.

StateChangeProviders



- ◆ The SDFComponent internal objects can implement the `StateChangeProvider` interface.
 - Other objects can register as listeners for `StateChangeEvents`.
 - `StateChangeEvents` contain simple information:
 - » E.G. `StateChangeEvents` sent from a Buffer
 - » `String` `changeDescriptor` (e.g. “NumTokens”).
 - » `Object` `oldState` (e.g. `Integer(3)`)
 - » `Object` `newState` (e.g. `Integer(4)`)
 - » `double` `time` - the time of the state change.

A Note on Compilation



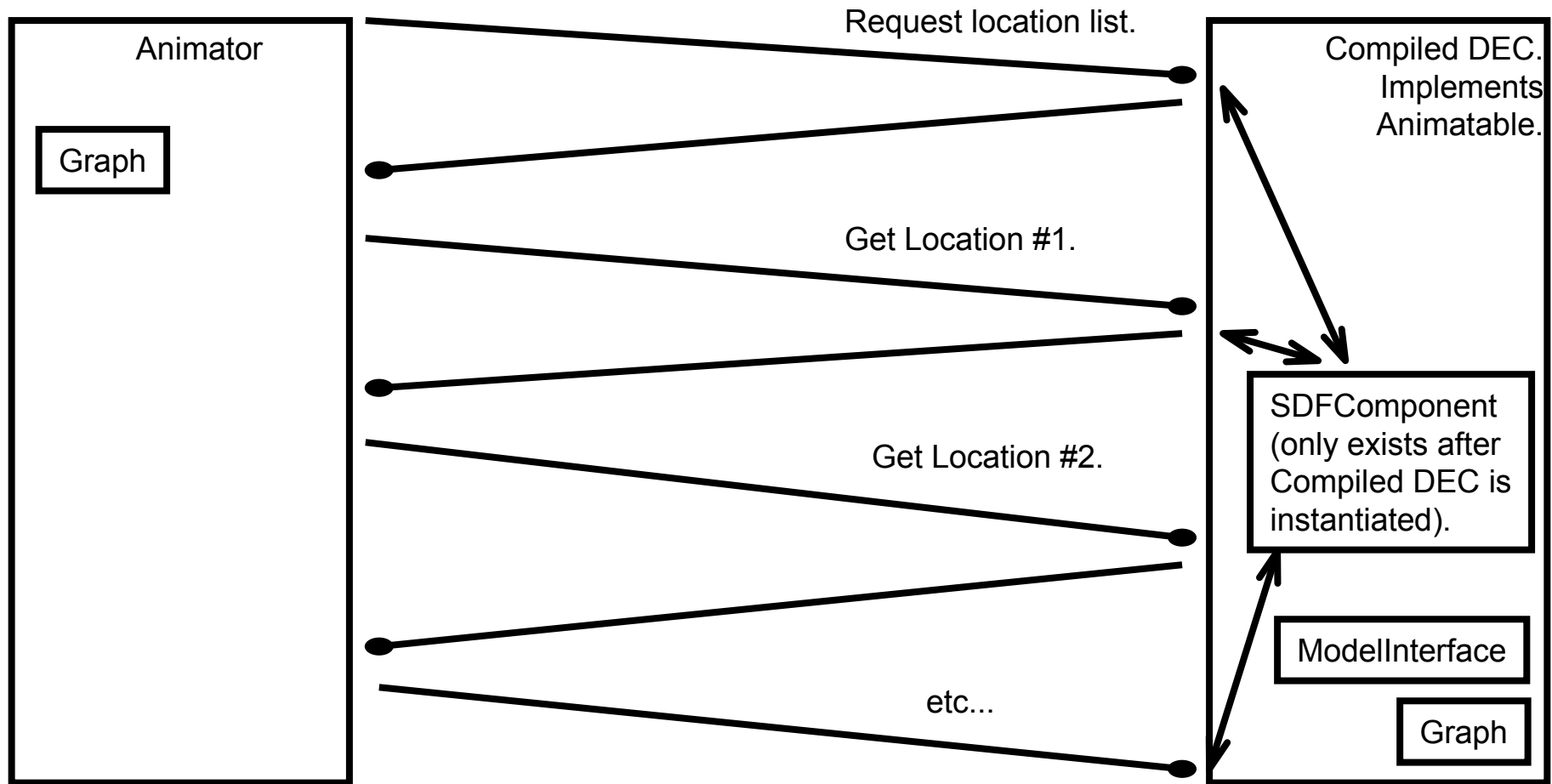
- ◆ When we set `supportsAnimation()` in the `SDFModelInterface` to return true:
 - The compiled DEC implements the `Animatable` interface as well as `DiscreteEventComponent`.
 - » This interface allows the animator to extract the DEC's `locationMap`.

Accessing StateChangeProviders



- ◆ We now need to access the `StateChangeProviders` in the `SDFComponent`.
 - The Moses animator has the `Graph` being animated, and the `locationMap` of the current `SDFComponent`.
 - » A `locationMap` maps edges and vertices to integers.
- ◆ We now modify `SDFComponent` to implement the interface `LocationMap`.
 - Different from the `locationMap` that is extracted from the `ModelInterface`.
 - Allows us to get the internal object representing the location (function, buffer, etc...).
 - The animator can extract these internal objects, and cast them to `StateChangeProviders` if necessary.

Animator Architecture



Interpreting StateChangeEvents



- ◆ We now have a way for the Animation module to access the StateChangeProviders for all locations, and receive StateChangeEvents.
- ◆ Something must interpret these events.
 - The AnimationAdapter provided in the semantic hooks section of the SDF GTDL file.
 - The AnimationAdapter creates an AnimationVisual.
 - » This can modify the vertex and edge graphics.
 - How?
 - » This explanation requires another presentation.