SwarmScript 3.0

Algorithmic Specification

Revision, January 2, 2011

Sebastian von Mammen, Scott Steil, and Christian Jacob

University of Calgary, Canada

1 SwarmScript—A High-level Description

SwarmScript provides a powerful programming approach based on situation-action pairs which allow the modeller to express agent behaviours and which equally well translate into computational processes. It merges a standardized representation for modelling agent behaviours with an algorithmic interpretation for simulating the resulting interactions.

In SwarmScript, situation-action pairs are built bottom-up from selection and action operators. The basic idea is that selection operators pass situational information to actions which change the simulation state. The information flow is determined through connections that are hooked up to plugs exposed by the operators. Information about the simulation state flows through chains of selection operators that feed into action operators.

On the one hand, chains of selection operators serve as sources of information for an action, e.g. they trigger whether or not an action should be executed and provide it with various parameters. These selection operators are classified as *source* selections. On the other hand, chains of selection operators direct an action toward specific targets. These selection operators are classified as *target* selections. Given that sources and targets are clearly separated, sources encompass all the information that motivates and informs an action, whereas targets clearly denote those programmatic objects that are changed based on the execution of an action. This distinction highlights the relationship between cause and effect, the two fundamental, interwoven constituents of an interaction.

The next section will introduce the data model of SwarmScript v3.0, whereas Sections 3 to 5 present pseudo code for creating, modifying and running SwarmScript simulations. Section 3 introduces some technical methods for setting, updating, and querying the connectivity of SwarmScript operators. Section 4 sheds light on routines necessary for creating and editing SwarmScript behaviours, including the creation of high-level operators. Finally, the pseudo code in Section 5 shows how SwarmScript behaviours can be simulated.

Some methods and functions that occur in the presented pseudo code are typically provided by high-level programming libraries and either access the properties of abstract data structures (getters/setters and type checking), create or delete instances (constructors receive their properties as parameters in the

order shown in Figure 1), or they are generic operators that work on collections. In particular, the following list-related routines are assumed to be commonly understood and available without further explanations:

– $list.append(element)$, $list.append(list)$,
– $list.remove(element)$,
– $list.hasNext()$ and $list.next()$,
– $list.hasElement(element)$, $list.hasElements(list)$,
– $list.first()$, $list.last()$, $list.get(index)$,
– $list.size()$,
– $list.copy()$, and
– $list.sort()$.

Regarding the usage of basic control flow routines, for and $while$ loops are used whenever the iteration order is important. Otherwise, $for\ each$ statements are deployed. $For\ all$ is only used once, in a case in which the most efficient way to iterate to a set of elements depends on the implementation framework.

2 Language Elements

In this section, the role of the various elements of the SwarmScript language is presented in detail and an according data model is introduced. Figure 1 shows a schematic diagram of the involved object classes and their dependencies in UML/Martin notation (Unified Modelling Language). Different Operator subclasses, i.e. Selections, Actions, and BehaviourFrames, have Plugs that are interconnected by means of Connection instances.

Figure 2 depicts a schematic visualization of a potential agent behaviour expressed by intertwined SwarmScript language elements. Connections are represented as thin black lines, small circles denote plugs, the wrapping boxes are operators, small numbers in the upper right corner of an operator indicate its hierarchical level, the $p_{trigger}$ plug of a behaviour frame is depicted as an on/off switch in the upper left corner of a behaviour frame. A hierarchical operator can be expanded to reveal its underlying operators. In such an open state, the higher-level operator is reduced to a framing rectangle enclosing its underlying operators with a button in the upper right corner. This button indicates that it can be wrapped, reduced to its compact representation at any point in time.

Actions and behaviour frames are placed on the grey strip, whereas source selections are arranged on the left-hand side and target selections on the right-hand side of the strip. Information flows through chains of selection operators from their output plugs to the connected input plugs.

Actions and behaviour frames only have input plugs. They receive information from source selections from the left and from target selections from the right. The spatial separation of selections in in accordance with their semantics is also reflected in the arrangement of input plugs of actions and behaviour frames—source input comes in from the left, target input from the right.

The following paragraphs detail the concepts underlying the individual language elements and how they work together.

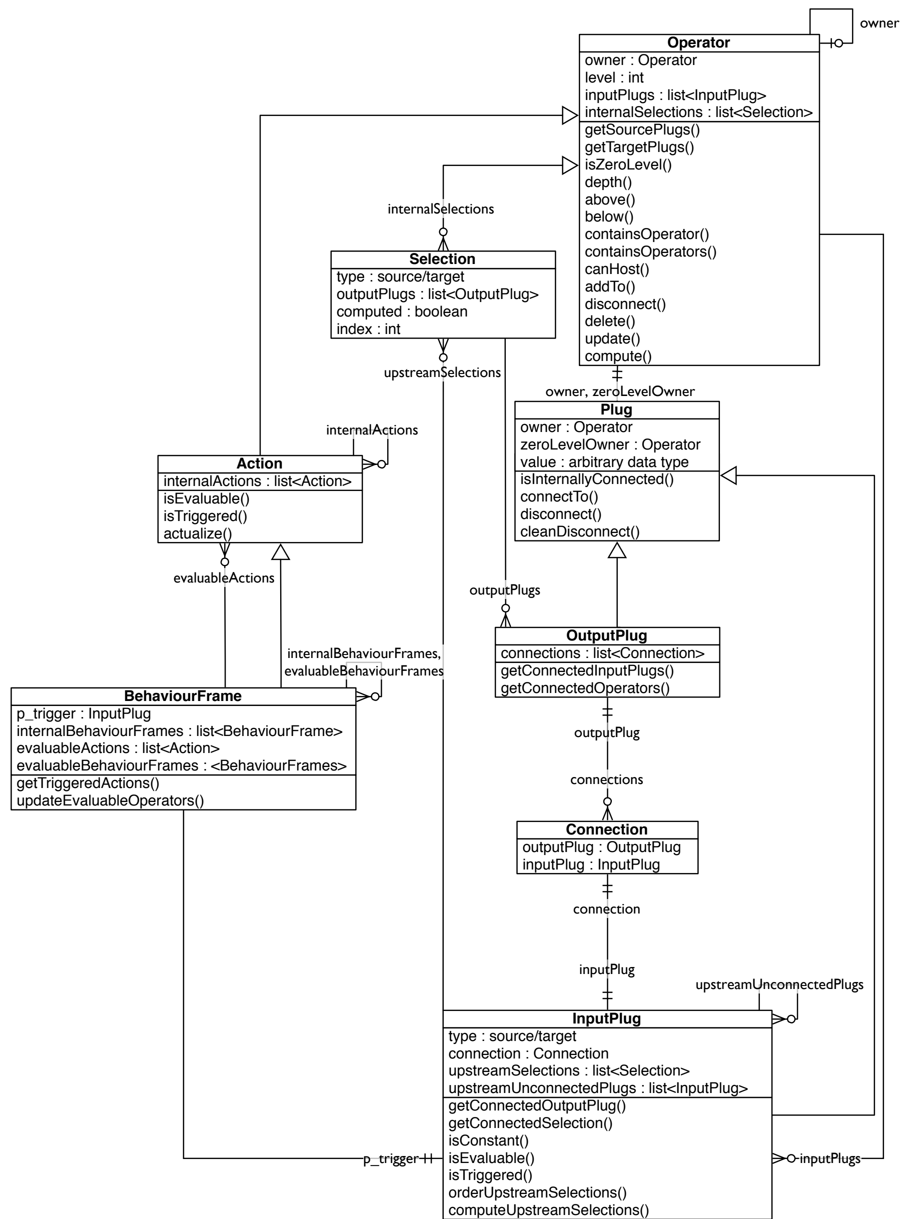

Fig. 1. The classes of the SwarmScript language elements in UML/Martin notation.

2.1 Connection

Connection instances link two operators by referencing one **input** and one **output plug**, respectively.

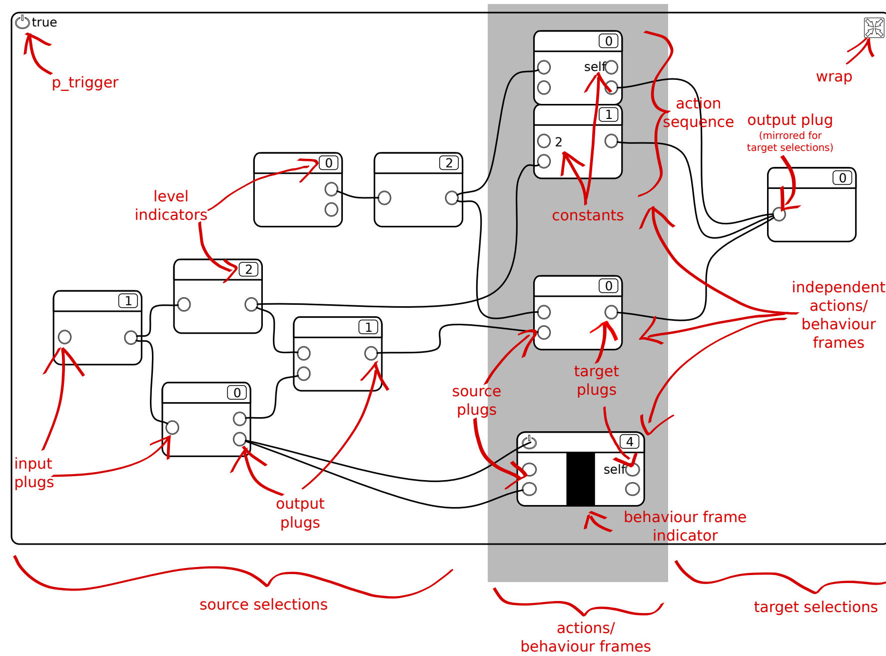

Fig. 2. A schematic visualization of SwarmScript language elements as they might be intertwined to express agent behaviour. The red arrows and labels denote the translation of the visuals into SwarmScript elements.

2.2 Plug

A Plug instance belongs to an operator, its **owner**. Additionally, it has a value assigned—by default it is *nil*.

2.3 OutputPlug

Output plugs can share any number of **connections** to input plugs.

2.4 InputPlug

Based on the intended use of the information received from the selection operators, input plugs are considered either of **type** source or target. An input plug may have one **connection** to an output plug. The sequence of selections that feeds into an input plug is stored in **upstreamSelections**, unconnected input plugs that occur in this sequence are stored in **upstreamUnconnectedPlugs**.

2.5 Operator

Selection, Action, and BehaviourFrame are directly or indirectly derived from the Operator class. An operator may be owned by another operator, its **owner**. The level of recursion of an operator is reflected by its **level** property. It is determined by the maximal recursion level of its underlying operators plus one. Any operator has a (possibly empty) list of input plugs (**inputPlugs**) which provide information to the operator, and a list of Selection instances which provide or refine the incoming information. An operator can be built hierarchically and host multiple underlying selection operators. As there might be dependencies regarding the execution order, these selections are stored in the ordered list of **internalSelections**. Unconnected input and output plugs of the internal operators become the input and output plugs of the higher level operator.

2.6 Selection

Selection operators provide situational information that informs or triggers, i.e. instigates, an agent's interactions. Selection operators can be joined into chains by connecting their output plugs to other operators' input plugs. Although the propagation of information across chains of selection operators can result in complex computations, selections do not change the simulation. Selections only pass information to actions which in turn change the simulation state based on the provided information.

A selection operator stores values in its output plugs based on information received from its input. It can, for example, filter a list of agents by their name (one input and one output plug) or merge two lists of agents and pass it on (two input plugs, one output plug). Selection operators without inputs can output information that is independent of preceding selection operators. Such terminal selection operators could, for instance, provide a constant number, the outcome of a random experiment, or forward information retrieved from external computational engines, e.g. objects that have collided with a given agent determined by a physics engine.

If a selection operator fails to compute, for instance because it is lacking important information, it **might** store a *nil* value in its output plug(s). An action operator that receives one or more *nil* values, will not be executed.

There are two **types** of selections. Source selections provide information to trigger and/or configure an action, whereas target selections identify those programmatic objects that will be changed when the action is executed. This distinction is mainly semantic at the model level but might be reflected at the user-interface level as shown in Figure 2.

Individual selection operators might feed their information into multiple action operators—either because the actions target the same programmatic objects or because they retrieve information from the same sources. Instead of evaluating shared selection operators multiple times, they are computed only once per iteration and their **computed** properties are set to true.

A selection's **index** property is utilized when it is sorted to resolve the execution interdependencies among operators.

2.7 Action

An action operator only has input plugs. These input plugs are classified as source and target plugs in accordance with their intended usage. An action retrieves information from its source plugs and applies changes to selections connected to its target plugs.

If one or more input plugs of a connected selection chain are unconnected or set to *nil*, an action is not evaluable, and thus, not considered for execution. All evaluable actions, however, are considered for execution at each simulation step. If their associated selection operators provide valid return values (non-*nil*), the actions get actualized.

An action can be comprised of a sequence of several underlying actions. Naturally, the whole sequence is only executed, if (1) all of the underlying actions are evaluable, and (2) all of the associated selections provide valid return values.

An action or an action sequence can be wrapped into high-level actions together with (parts of) the connected selection chains. Unconnected source or target plugs of the wrapped **internalActions** are adopted by the new operator as such, whereas unconnected input plugs of the associated selections are added to its source or target plugs depending on their types.

2.8 BehaviourFrame

Behaviour frames can wrap sets[1] of **internal** selections, actions, and **behaviour frames**. Similar to actions, behaviour frames only expose input plugs of types source and target. However, different from actions, the exposed plugs of a behaviour frame do not need to be connected in order to consider the encapsulated actions for execution. Any evaluable, encapsulated actions are still considered for execution.

However, the execution of a behaviour frame may be skipped altogether, in case its activation plug $\mathbf{p_{trigger}}$ is not evaluable [2]. Therefore, as a first optimization step, evaluable actions and lower-level behaviour frames whose $p_{trigger}$ plugs are evaluable are stored in a behaviour frame's **evaluableActions** and **evaluableBehaviourFrames** lists. In a second step, only these evaluable behaviour frames and actions will be considered for execution.

2.9 Schematic Instance Representation

The series of Figures 3 to 7 depicts the hierarchical design of a possible SwarmScript behaviour. In Figure 3, a sequence of two action operators is shown at

[1] In numerous cases, *sets* would be mathematically more adequate as containers for SwarmScript elements than *lists*. However, due to their topological relationships, it is important to store most SwarmScript operators in ordered *lists*. Using lists facilitates the maintenance of an expedient, consistent relationship between the syntactical arrangement and the semantical interpretation of their symbolical representation.

[2] Due to the design of SwarmScript, especially because actions only execute, if their input plugs have or receive valid information, the return value *nil* generally implies no execution.

the top of the grey strip, followed by another single action in the centre, and a behaviour frame at the bottom.

The sequence of two action operators at the top will execute in top-down order, only if (a) both operators are fully connected and (b) all their plugs receive non-*nil* values. Due to their interdependency, sequences of actions can be wrapped into a higher order action operator. As one can see, the given action sequence will not be considered for execution, since its second plug on the left-hand side is not set.

The action operator in the centre of the grey strip, it seems, is fully connected. However, upon closer inspection, one will find that the left-most selection operator, which this action operator also depends on, is not fully connected. Therefore, this action operator, too, will not be considered for execution in the given configuration. The same is true for behaviour frame operator at the bottom of the grey strip[3].

In Figure 4, the level-2 selection operator to the left is opened up for inspection and modification. It shows that its output is generated by passing information through four internal selection operators, one of which is hierarchically designed as well (level-1). The second action of the action sequence at the top of the grey strip is opened up in Figure 5, revealing that it consists of another sequence of two action operators. In Figure 6, the behaviour frame at the bottom of the grey strip is opened. It consists of two behaviour frames itself and four associated selection operators. Diving into the bottom-most behaviour frame one more time in Figure 7 reveals the combination of one action sequence and one further, independent action, depending on another nine internal selection operators.

3 Basic Routines

This section introduces routines that revolve around the connectivity, i.e. retrieving, setting, and changing the state of connectivity of SwarmScript operators. These rather technical routines are fundamental for creating and editing a SwarmScript behaviour (Section 4) and for processing it as part of a simulation (Section 5).

3.1 Additional Getter Methods

By means of their respective connections, plugs can be queried for their linked plugs or linked operators. In particular, an OutputPlug instance can be queried

[3] The behaviour frame is visualized with a black bar at its centre, which not only resembles the grey strip of the behaviour frame that hosts the complete behaviour of an agent. But it also indicates that the source information on the left-hand side of the operator might not be related to the changes induced to the targets connected on the right-hand side. In actions, on the other hand, the internal logic between sources and targets is intertwined to such a degree that they can be considered completely dependent. Therefore, actions are not shown with a bar between the two kinds of input plugs.

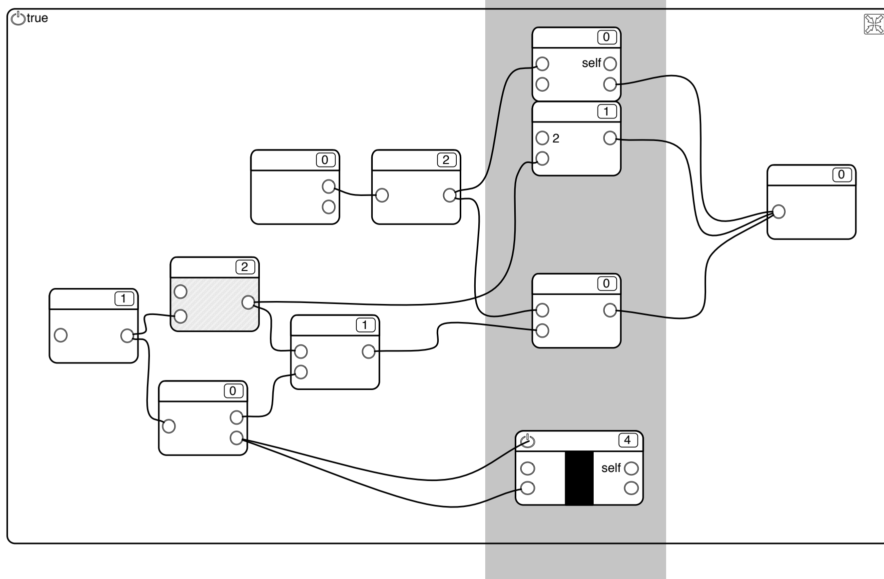

Fig. 3. A possible SwarmScript behaviour in the works—all actions and the depicted behaviour frame are not fully connected, yet.

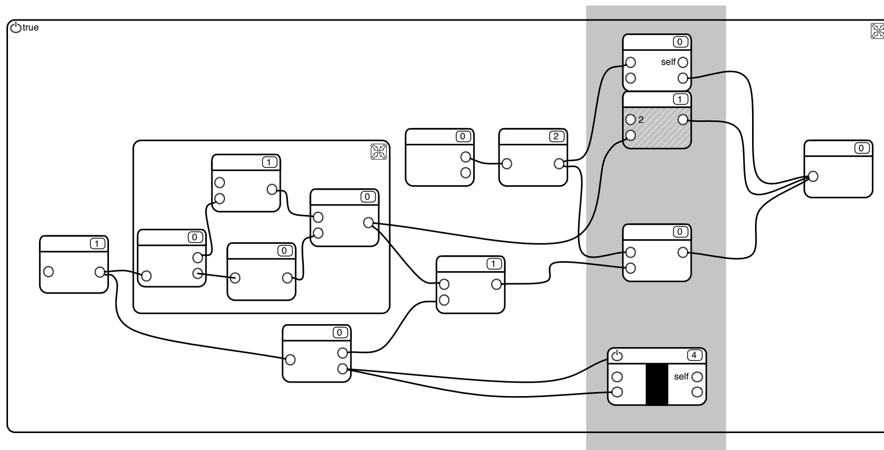

Fig. 4. The selection operator at the left centre has been opened up for inspection/modification.

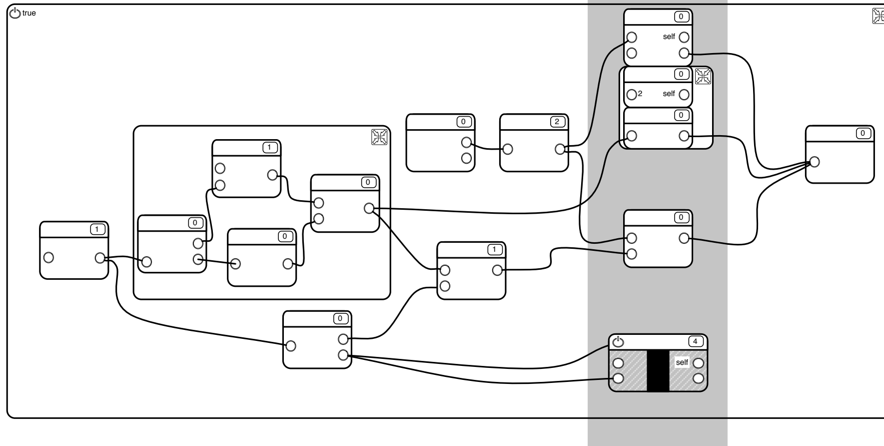

Fig. 5. The second action operator of the action sequence on the grey strip is itself composed of a sequence of two action operators.

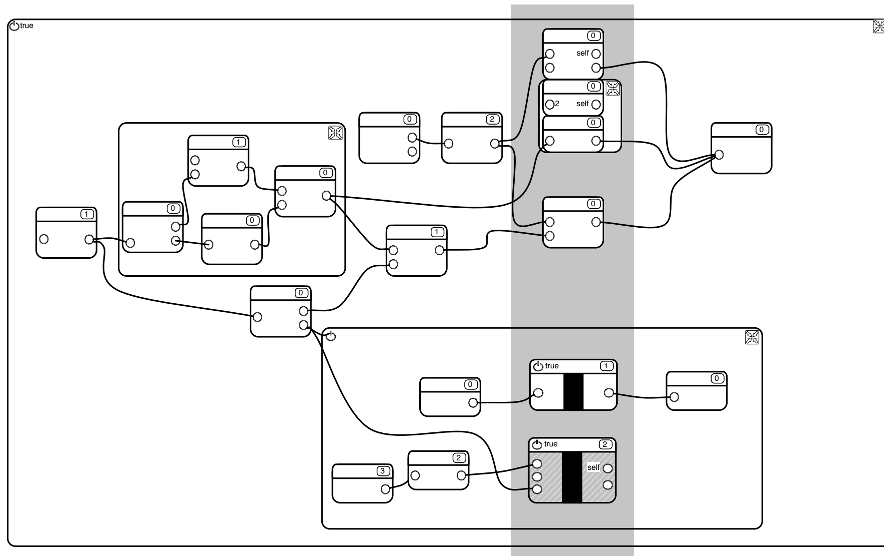

Fig. 6. The behaviour frame at the bottom consists of two lower-level behaviour frames and four selection operators.

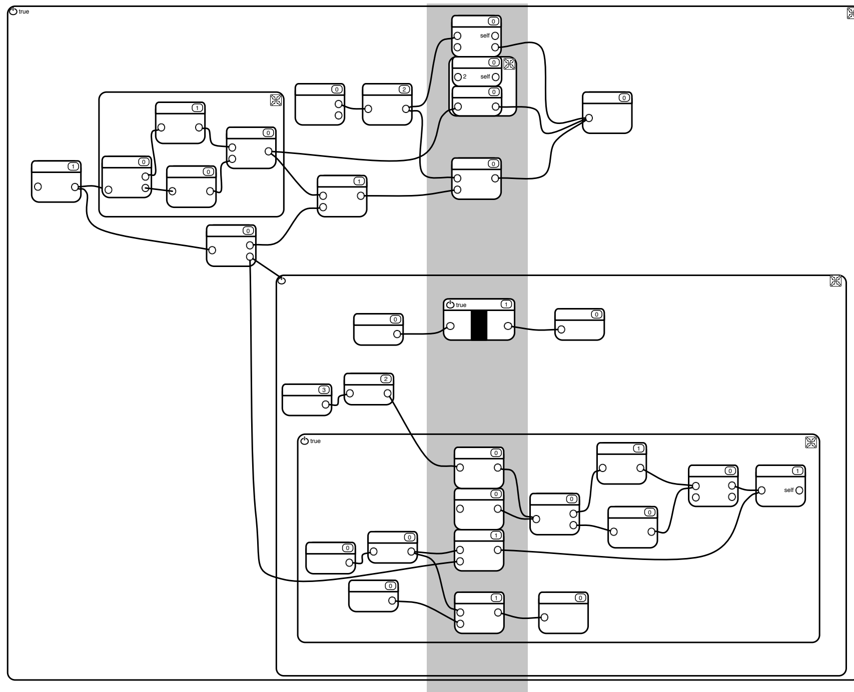

Fig. 7. Nine more selection operators, an action sequence and another independent action were hidden in the bottom-most behaviour frame.

for its linked input plugs (**getConnectedInputPlugs()**, Alg. 1) or linked operators (**getConnectedOperators()**, Alg. 2). An InputPlug instance can be queried for its associated output plug (**getConnectedOutputPlug()**, Alg. 3) or selection operator (**getConnectedSelection()**, Alg. 4).

Output: $list\langle InputPlug\rangle$
$plugs_{connected} := \{\}$;
for $i := 0,\ i < self.connections.size(),\ i++$ **do**
$plugs_{connected}.append(self.connections.get(i).inputPlug)$;
return $plugs_{connected}$;

Algorithm 1: $OutputPlug.getConnectedInputPlugs()$

Output: $list\langle Operator\rangle$
$operators_{connected} := \{\}$;
$plugs_{connected} :=$ self.$getConnectedInputPlugs()$;
for $i := 0,\ i < plugs_{connected}.size(),\ i++$ **do**
$operators_{connected}.append(plugs_{connected}.get(i).owner)$;
return $operators_{connected}$;

Algorithm 2: $OutputPlug.getConnectedOperators()$

Output: OutputPlug
return self.connection.outputPlug;

Algorithm 3: $InputPlug.getConnectedOutputPlug()$

Output: Selection
connectedOutputPlug := self.$getConnectedOutputPlug()$;
if *connectedOutputPlug* == *nil* **then**
| **return** *nil*
end
else
| **return** connectedOutputPlug.owner;
end

Algorithm 4: $InputPlug.getConnectedSelection()$

It is important to know which action plugs and $p_{trigger}$ plugs a specific input plug affects based on the given connection topology. Any topological changes

require these plugs to update their dependencies. **getAffectedInputPlugs()** (Alg. 5) finds an according list of affected, non-Selection plugs. The function tests whether the given input plug belongs to a non-Selection operator, otherwise it recursively steps through all the output plugs of its owner and recursively considers their linked input plugs. Figure 8 depicts the algorithmic process effected by the $getAffectedInputPlugs()$ routine. Starting with the input plug of the left-most selection operator, it forward traverses all linked selection operators until it connects to some of the action operators located on the grey strip, to the $p_{trigger}$ plug of the outer, opened behaviour frame, and to one of its selection's input plugs, which is exposed as one of the behaviour frame's exposed input plugs.

The latter case deserves some attention: When building the hierarchical behaviour frame operators, the Selection instance's input plug became a source plug of the inner behaviour frame first, and of outer behaviour frame afterward. Therefore, it is not considered a plug of a Selection instance any longer. As high-level operators become the owners of their exposed plugs (implemented by the updating routines, Algs. 40 and 41), the internal selection operators of the opened selection operator are not traversed.

```
Output: list⟨InputPlug⟩
if class of self.owner is not Selection then return {self};
downstreamAffectedInputPlugs := {};
foreach p_out ∈ self.owner.outputPlugs do
    foreach p_in ∈ p_out.getConnectedInputPlugs() do
        downstreamAffectedInputPlugs.append(p_in.getAffectedInputPlugs());
    end
end
return downstreamAffectedInputPlugs;
```

Algorithm 5: $InputPlug.getAffectedInputPlugs()$

The Operator base class provides input and output plugs and according, inferable setter and getter methods. The input plugs of action operators and behaviour frames are divided into source and target plugs. The Operator methods **getSourcePlugs()** (Alg. 6) and **getTargetPlugs()** (Alg. 7) provide easy access to these two distinct types of input plugs.

3.2 State Querying Routines

As shown in Figure 9, a behaviour frame B_{root} hosts a set of operators that may in turn be composed of lower-level operators. The leaves of the operator tree are referred to as $0 - level$ operators; this state is determined by checking whether an operator has any internal operators (Alg. 8). There are several methods that provide information about an operator's state in the hierarchy. $depth()$ yields the distance of an operator to the root (Alg. 9). Two operators may lie on the same

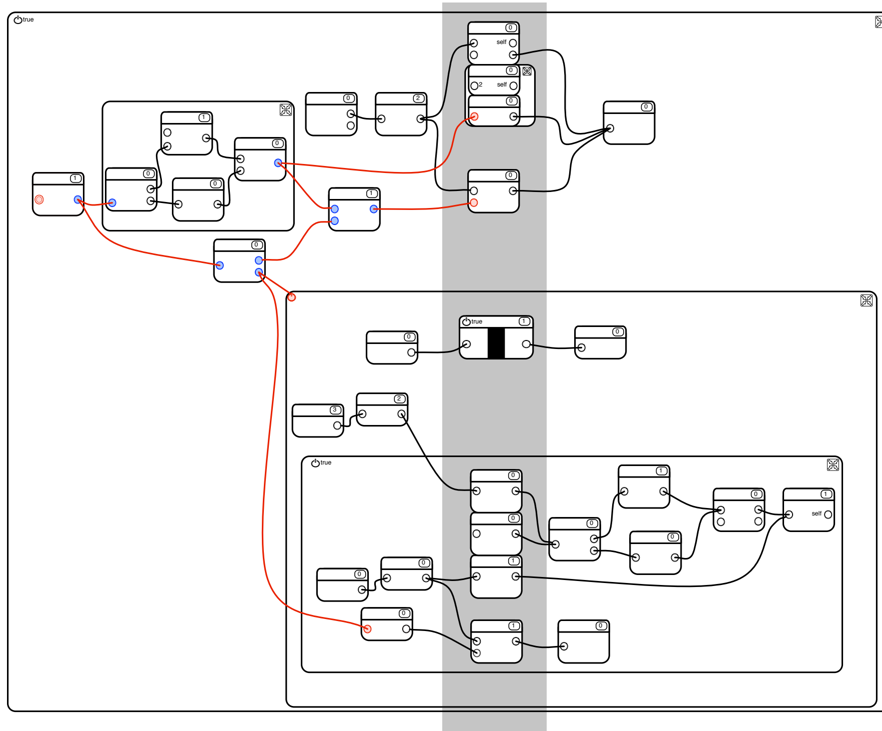

Fig. 8. Starting from the left-most selection operator, all affected non-Selection input plugs are discovered by the function $getAffectedInputPlugs()$. The result are the red coloured plugs of operators on the grey strip as well as the single source and the $p_{trigger}$ plug of the opened, outer behaviour frame.

Output: $list\langle InputPlug\rangle$
plugs := {};
for $i := 0,\ i < self.inputPlugs.size(),\ i{+}{+}$ **do**
  if *self.inputPlugs.get*(i)*.type* == *source* **then**
  plugs.$append(self.inputPlugs.get(i))$;
end
return plugs;

Algorithm 6: $Operator.getSourcePlugs()$

Output: $list\langle Plug\rangle$
plugs := {};
for $i := 0,\ i < self.inputPlugs.size(),\ i{+}{+}$ **do**
  if *self.inputPlugs.get*(i)*.type* == *target* **then**
  plugs.$append(self.inputPlugs.get(i))$;
end
return plugs;

Algorithm 7: $Operator.getTargetPlugs()$

path of the tree, one being above the other one being below (Algs. 10 and 11). Whether an operator contains other operators as (direct) children is revealed by the *containsOperator*() and *containsOperators*() methods (Algs. 12 to 15).

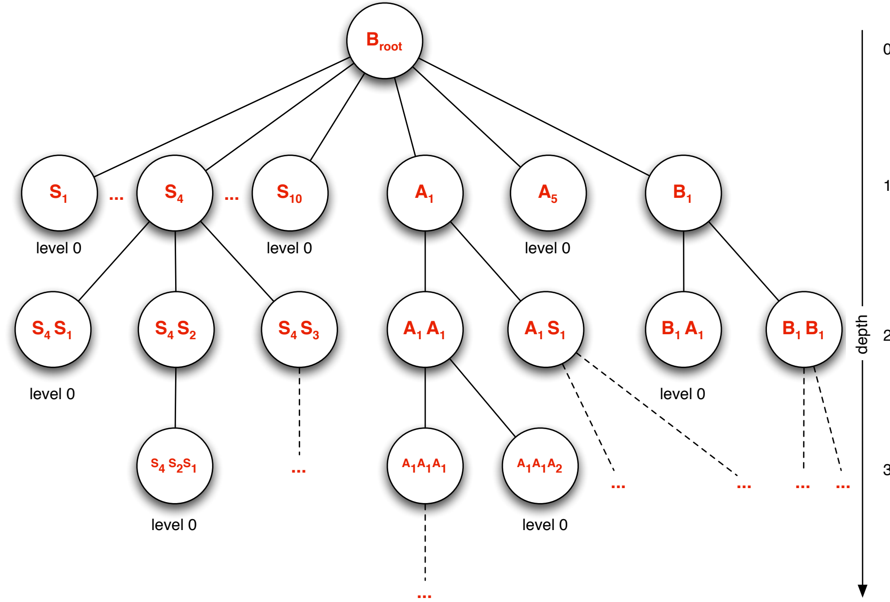

Fig. 9. The operator hierarchy; High-level operators are built bottom-up from zero-level operators. The steps needed to get to the tree's root B_{root} denotes an operator's depth in the hierarchy.

A plug can either be connected or not. The according state querying function **isConnected()** tests whether the list of connections of an output plug is initialized and not empty (Alg. 16), its implementation for an input plug simply tests whether or not its connection reference is initialized at all (Alg. 17).

The internal operators of a high-level wrapper may be connected with one another, as for example in the expanded selection wrapper in Figure 4. Here, four internal selection operators are interconnected. Some of them are also connected to the outside of the wrapper. The method**isInternallyConnected()**, Algs. 18 and 19, checks whether a plug is only connected within a given wrapper ($true$), or whether it is connected beyond its boundaries ($false$). It first queries whether a plug is connected at all, and second whether its connected operators are children of the given operator.

While the value of an output plug is updated as part of the execution of the internal logic of the respective operator, input plugs pull in information, allowing for its propagation across chained operators. It is important to know whether

Output: boolean
if *self.internalSelections.size*() > 0 **then return** false;
if *class of self is Selection* **then return** true;
if *self.internalActions.size*() > 0 **then return** false;
if *class of self is Action* **then return** true;
if *self.internalBehaviourFrames.size*() > 0 **then return** false;
return true;

Algorithm 8: *Operator.isZeroLevel*()

Output: int
depth := 0;
ancestor := self.owner;
while *ancestor != nil* **do**
  ancestor := ancestor.owner;
  depth += 1;
end
return depth;

Algorithm 9: *Operator.depth*()

Input: o: *Operator*
Output: Boolean
return *o.below*(*self*);

Algorithm 10: *Operator.above*()

Input: o: *Operator*
Output: Boolean
if *self.depth*() <= *o.depth*() **then return** *false*;
ancestor := self.owner;
while *ancestor != o AND ancestor.owner != nil* **do**
  ancestor := ancestor.owner;
end
return *ancestor* == *o*;

Algorithm 11: *Operator.below*()

Input: o : Operator
if *class of o is* ***not*** *Selection* **then**
  return *false*;
else
  return *self.internalSelections.hasElement*(*o*);
end

Algorithm 12: *Selection.containsOperator*()

```
Input: o : Operator
if class of o is Selection then
│ return self.internalSelections.hasElement(o);
else if class of o is Action then
│ return self.internalActions.hasElement(o);
else
│ return false;
end
```

Algorithm 13: *Action.containsOperator*()

```
Input: o : Operator
if class of o is Selection then
│ return self.internalSelections.hasElement(o);
else if class of o is Action then
│ return self.internalActions.hasElement(o);
else
│ return self.internalBehaviourFrames.hasElement(o);
end
```

Algorithm 14: *BehaviourFrame.containsOperator*()

```
Input: ops : ⟨Operator⟩
foreach Operator o ∈ ops do
│ if self.containsOperator(o) == false then return false;
end
return true;
```

Algorithm 15: *Operator.containsOperators*()

Output: boolean
return self.connections $! = nil$ AND self.connections.$size() > 0$;
Algorithm 16: $OutputPlug.isConnected()$

Output: boolean
return self.connection $! = nil$;
Algorithm 17: $InputPlug.isConnected()$

it is possible for an input plug to receive a valid value or not—an invalid value (nil) at a non-selection input plug prevents its operator from execution. We say an input plug is evaluable (**InputPlut.isEvaluable()**, Alg. 21), if it is set to a constant value (**isConstant()**, Alg. 20), or it is connected and it does not depend on any unconnected (or not constant) input plugs (which are accordingly titled upstream unconnected plugs). If all the input plugs of an action operator are evaluable, the operator itself is also evaluable (**Action.isEvaluable()**, Alg. 22). A behaviour frame is considered evaluable, if its activation plug, $p_{trigger}$, is evaluable (**BehaviourFrame. isEvaluable()**, Alg. 23). The $isEvaluable()$-functions ensure that the respective plugs and operators have the potential to receive non-nil values, either because they are fully upstream connected or because they provide constant, non-nil values.

Remark: Operators at the top-most level that are not evaluable are not considered for execution since they are lacking some information. However, lower-level operators that are not evaluable are considered for execution regardless since they might be properly connected at higher levels.

A positive return value of $isEvaluable()$ does not rule out the possibility than the operator may receive an invalid nil value during the actual evaluation process. Only if it receives non-nil information, we say the plug or the operator is triggered (**Plug.isTriggered()**, **Action.isTriggered()**, and **BehaviourFrame. isTriggered()**, Algs. 24 to 26).

An evaluable input plug is triggered, if it returns a constant value or if it retrieves a non-nil value from its connected output plug. In order to test whether a valid value is retrieved from a connected output plug, the method **InputPlug.computeUpstreamSelections()** (Alg. 45) needs to be called, which computes the list of upstream connected selections in the given order and updates the value of the connected output plug. This method is presented as part of the simulation routines of Section 5. Afterward, the input plug retrieves the updated value from the output plug. If it is non-nil, the input plug is considered

Input: op: Operator
Output: boolean
return self.$isConnected()$ AND
op.$containsOperators(p.getConnectedOperators())$;
Algorithm 18: $OutputPlug.isInternallyConnected()$

Input: op: Operator
Output: boolean
return self.$isConnected()$ AND
op.$containsOperator(p.getConnectedSelection())$;

Algorithm 19: $InputPlug.isInternallyConnected()$

Output: boolean
return self.$isConnected()$ == $false$ AND self.value ! = nil;

Algorithm 20: $InputPlug.isConstant()$

Output: boolean
return self.$isConstant()$ OR (self.$isConnected()$ AND
self.upstreamUnconnectedPlugs.$size()$ == 0);

Algorithm 21: $InputPlug.isEvaluable()$

Output: boolean
for $i := 0, i < self.inputPlugs.size(), i++$ **do**
  p := self.inputPlugs.$get(i)$;
  if $p.isEvaluable() == false$ **then** **return** $false$;
end
return $true$;

Algorithm 22: $Action.isEvaluable()$

Output: boolean
return self.$p_{trigger}.isEvaluable()$;

Algorithm 23: $BehaviourFrame.isEvaluable()$

"triggered". The method **InputPlug.orderUpstream Selections()** (Alg. 27) prepares an input plug's upstreamSelections list for computation. It is detailed in the following paragraphs on state updating routines.

Action.isTriggered() first tests whether all the input plugs of an action yield non-*nil* values. If it is a level-0 action, i.e. if it does not contain further internal actions, it is indeed triggered. If, however, it contains further internal actions and possibly internal selection operators, the higher-level action as a whole gets triggered only if all its internally wrapped actions are triggered. Hence, if only one contained internal action is not triggered, the high-level operator is not triggered either. The method recursively works through the action hierarchy.

```
Output: boolean
if self.isConstant() then return true;
self.computeUpstreamSelections();
self.value := self.connection.outputPlug.value;
return self.value != nil;
```

Algorithm 24: *InputPlug.isTriggered*()

```
Output: boolean
for i := 0, i < self.inputPlugs.size(), i++ do
    p := self.inputPlugs.get(i);
    if p.isTriggered() == false then return false;
end
if self.level > 0 then
    for i := 0, i < self.internalActions.size(), i++ do
        a := self.internalActions.get(i);
        if a.isTriggered() == false then return false;
    end
end
return true;
```

Algorithm 25: *Action.isTriggered*()

```
Output: boolean
return self.p_trigger.isTriggered();
```

Algorithm 26: *BehaviourFrame.isTriggered*()

3.3 State Updating Routines

The method $InputPlug.orderUpstreamSelections()$ (Alg. 27) retrieves and orders the internal selection operators that are connected to an input plug. It also aggregates upstream plugs that are neither constant nor connected. Each internally upstream-connected Selection instance is added to the list of upstreamSelections and is marked with a step-wise increased index value, starting from 0. Its input plugs are added to the list of plugs to follow up on. During the upstream traversal of connected selections, the selections are assigned index values that inversely reflect their order of access—selections with high indices need to be computed before those with lower indices. The list of upstream selections is sorted accordingly. Input plugs that are neither connected nor constant represent dependencies that cannot be internally resolved—these are added to the list of $upstreamUnconnectedPlugs$[4].

```
self.upstreamUnconnectedPlugs := {};
self.upstreamSelections := {};
index := 0;
inputPlugs := {self};
for i := 0, i < inputPlugs.size(), i++ do
    p_in := inputPlugs.get(i);
    if p_in.isInternallyConnected(self.owner.owner) then
        connectedSelection := p_in.getConnectedSelection();
        connectedSelection.index := index++;
        inputPlugs.append(connectedSelection.inputPlugs);
        self.upstreamSelections.append(connectedSelection);
    else if p_in.isConnected() == false AND p_in.isConstant() == false then
        self.upstreamUnconnectedPlugs.append(p_in);
    end
end
self.upstreamSelections.sort() with descending index;
```

Algorithm 27: $InputPlug.orderUpstreamSelections()$

Figure 10 depicts the process effected by $InputPlug.orderUpstreamSelections()$ working on B's upper input plug (double-lined, red circle). Afterward, its associated list $upstreamUnconnectedPlugs$ will contain A's upper input plug. The list $upstreamSelections$ will contain C and A, and in this order, since C needs to compute first. In Figure 11 the upstream selections of the second input plug (marked with a double red line) of an action operator are computed. Its list $upstreamConnectedPlugs$ will contain the blue coloured input plugs of operators A' and D, its $upstreamSelections$ list will be $\{D, G, A', F\}$. During

[4] External connections, i.e. those that connect to non-internal operators, need not be considered in the $orderUpstreamSelections()$ method since (1) external operators will be executed at a higher algorithmic level and (2) any upstream unconnected plugs are connected through exposed plugs of the high-level operator anyways.

the computation of the upstream selections, the index value of D was set to 3 first, then overwritten by the value 4 since two independent downstream selection operators accessed it. Overwriting indices ensures that an operator is computed as soon as necessary.

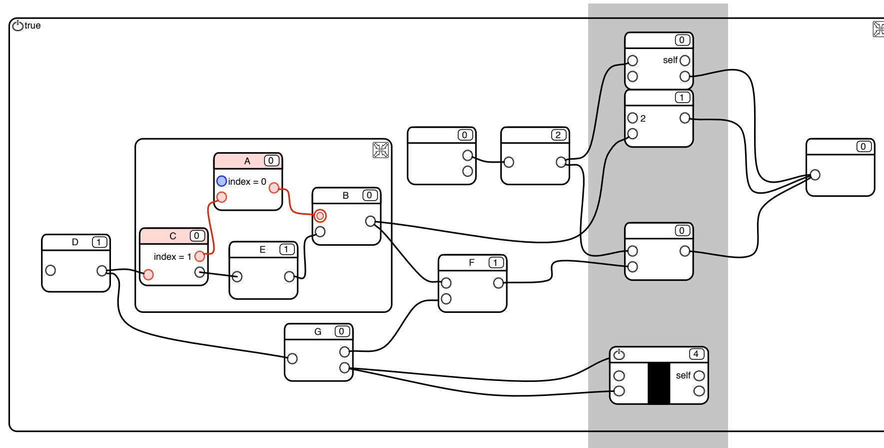

Fig. 10. Starting from the upper one of B's input plugs (the double-lined red circle), the upstream selection operators A and C are determined and ordered for execution.

4 Routines of the Modelling Phase

The routines of the modelling phase can be roughly divided into three categories: Adding and removing operators, adding and removing connections, and creating high-level operators.

4.1 Adding/Removing Operators

During the modelling phase, selections, actions and behaviour frames are combined to shape the behaviour of an agent. A single behaviour frame, also referred to as the "root" behaviour frame, serves as a workspace to host and configure SwarmScript operators. Predefined, configurable SwarmScript operators may be provided by an operator library and be added to the behaviour frame or to its lower-level operators.

Operator.canHost() (Alg. 28) tests whether an operator can wrap a given list of operators. Behaviour frames can host any kind of operators, but they cannot be added to actions, and neither actions nor behaviour frames can be added to selections.

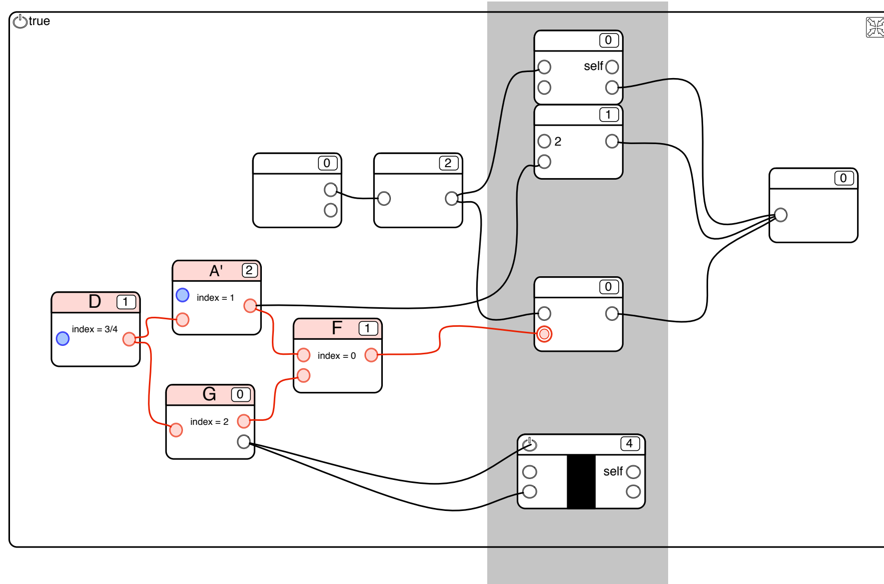

Fig. 11. Starting from an action operator's input plug (the double-lined red circle), the upstream selection operators F, A', G and D are determined and stored as the ordered list $\{D, G, A', F\}$.

If possible, **Operator.addTo()** (Alg. 29) adds an operator, which has not previously existed in the model context, to another one. Once a new operator has been introduced, its owner is updated in order to ensure that it exposes the proper plugs. The implementations of the **Operator.update()** method are presented in the paragraphs about creating high-level operators, starting with Section 4.3.

The transfer of a list of already existing operators into an enclosing operator is realized by the **transfer** procedure (Alg. 30); The operation is cancelled, if the selected operators cannot be introduced into the target operator. In case the target operator is lower in the hierarchy, all the connections of the transferred operators can be maintained. Otherwise, all those plugs of the transferred operators need to be disconnected that were internally connected to operators that are **not** being transferred as well. Otherwise, we could not ensure that a consistent state is achieved, without illegal connections (for instance from on action to another one). Next, the transfer procedure detaches the operators from their previous owners, appends them to the respective lists of their new owner, and adjusts the ownership reference. After all the operators have been transferred, the previous owners as well as the target that received the operators are updated to re-establish consistent connectivity and information flow across the hierarchies.

Operator.delete() (Alg. 32) completely removes an operator—it detaches itself from its owner, deletes its connections, updates its previous owner, and finally removes itself.

```
Input: operators: list < Operator >
Output: Boolean
if class of self is BehaviourFrame then return true;
atLeastOneAction := false;
foreach Operator o ∈ operators do
    if class of o is BehaviourFrame then
        return false;
    else if class of o is Action then
        atLeastOneAction := true;
    end
end
if class of self is Action then return true;
if atLeastOneAction == true then return false;
return true;
```

Algorithm 28: *Operator.canHost*()

```
Input: to: Operator
if to.canHost({self}) == false then return;
if class of self is Selection then
    to.internalSelections.append(self);
else if class of self is Action AND class of to is not Selection then
    to.internalActions.append(self);
else if class of self is BehaviourFrame AND class of to is BehaviourFrame then
    to.internalBehaviourFrames.append(self);
end
self.owner := to;
self.owner.update();
```

Algorithm 29: *Operator.addTo*()

4.2 Adding/Removing Connections

Programmatic relationships are created through setting up connections from output to input plugs. Thus, selection operators are joined into chains that are eventually hooked up to the activation plugs $p_{trigger}$ of behaviour frames or to the source and target plugs of action operators. A Connection instance references an output and an input plug and is itself added to the connection list of its input and referenced as the connection of an output plug. When connecting an output plug or input plug these dependencies are updated accordingly

```
Input: operators: list < Operator >, target: Operator
if target.canHost(operators) == false then return;
previousOwners := list of copies of operators' owners;
foreach Operator o ∈ operators do
    foreach InputPlug p ∈ o.inputPlugs do
        connectedOp := p.getConnectedSelection();
        if connectedOp.owner.depth() > target.depth() AND
        operators.hasElement(connectedOp) == false then
            p.disconnect();
        end
    end
    if class of o is Selection then
        foreach OutputPlug p ∈ o.outputPlugs do
            connectedPlugs := p.getConnectedInputPlugs();
            connectedOps := {};
            connectedOpsOutOfReach := false;
            foreach InputPlug p ∈ connectedPlugs do
                connectedOps.append(p.owner);
                if p.owner.owner.depth() > target.depth() then
                    connectedOpsOutOfReach := true;
                end
            end
            if operators.hasElements(connectedOps) == false AND
            connectedOpsOutOfReach == true then
                p.disconnect();
            end
        end
    end
    if class of o is Selection then
        o.owner.internalSelections.remove(o);
        target.internalSelections.append(o);
    else if class of o is Action then
        o.owner.internalActions.remove(o);
        target.internalActions.append(o);
    else if class of o is BehaviourFrame then
        o.owner.internalBehaviourFrames.remove(o);
        target.internalBehaviourFrames.append(o);
    end
    o.owner := target;
end
previousOwners.append(target);
smartUpdate(previousOwners);
```

Algorithm 30: $transfer()$

($OutputPlug.connectTo()$, $InputPlug.connectTo()$, Algs. 34 and 33). If an output plug is connected to an already connected input plug, that input plug is disconnected first.

```
if class of self is Selection then
    foreach plug p ∈ self.outputPlugs do
        p.disconnect();
    end
end
foreach plug p ∈ self.inputPlugs do
    if p.isConnected() then p.disconnect();
end
```

Algorithm 31: *Operator.disconnect*()

```
if class of self is Selection then
    self.owner.internalSelections.remove(self);
else if class of self is Action then
    self.owner.internalActions.remove(self);
else if class of self is BehaviourFrame then
    self.owner.internalBehaviourFrames.remove(self);
end
self.disconnect();
self.owner.update();
delete(self);
```

Algorithm 32: *Operator.delete*()

When a connection is created, the enclosing host operator needs to be updated, as its internal topology has changed. In case, the connection is part of the internal topological definition of a selection operator, the execution sequence of internal operators has to be re-calculated. If the connection is encapsulated by an action operator or inside a behaviour frame, it plays a role in connecting a selection operator or a chain of selection operators to one or more actions. The affected action plugs need to update their dependencies accordingly, as well. No matter which wrapping operator hosted the connection, it might have to update its exposed plugs.

When disconnecting an input plug (*InputPlug.disconnect*(), Alg. 35), the reference to the corresponding Connection instance needs to be removed from the connected output plug. Then, the Connection instance is deleted, also setting the input plug's reference to *nil*.

Finally, all the affected non-Selection plugs need to update their upstream dependencies, since they have changed. And the associated owner's host needs to get updated as its exposed plugs might have changed as well. Disconnecting an output plug, requires the removal of all its connections (*OutputPlug.disconnect*(), Algs. 37), including updating the owners of the connected operators as well as updating any affected input plugs that belong to actions and behaviour frames[5].

[5] There is no need to update anything inside of the owners of the identified non-Selection plugs, since the non-exposed plugs do not have any external dependencies.

```
Input: to: OutputPlug
if self.isConnected() then self.disconnect();
self.connection := new Connection(to, self);
to.connections.append(self.connection);
foreach Plug p ∈ self.getAffectedInputPlugs() do
  | p.orderUpstreamSelections();
end
self.owner.owner.update();
```

Algorithm 33: $InputPlug.connectTo()$

```
Input: to: InputPlug
to.connectTo(self);
```

Algorithm 34: $OutputPlug.connectTo()$

```
self.connection.outputPlug.connections.remove(self.connection);
delete(self.connection);
self.connection := nil;
foreach Plug p ∈ self.getAffectedInputPlugs() do
  | p.orderUpstreamSelections();
end
```

Algorithm 35: $InputPlug.disconnect()$

```
self.disconnect();
self.owner.owner.update();
```

Algorithm 36: $InputPlug.cleanDisconnect()$

```
foreach InputPlug p_in ∈ self.getConnectedInputPlugs() do
  | p_in.disconnect();
end
```

Algorithm 37: $OutputPlug.disconnect()$

```
self.disconnect();
self.owner.owner.update();
```

Algorithm 38: $OutputPlug.cleanDisconnect()$

4.3 Creating High-Level Operators

In order to provide powerful programmatic building blocks to the modeller, we present routines that wrap SwarmScript "circuitry", including operators and connections, into higher-level operators. The plugs exposed by a high-level operator reference the plugs of the internal selections, actions and behaviour frames. Following this approach, the connection dependencies of newly wrapped operators and their contexts are automatically maintained.

wrapOperators() Alg. 39 wraps a set of selected operators. It first creates an empty wrapper that is added to the owner of the highest selected operator. It determines which wrapper is needed based on the diversity of selected operators. In particular, if multiple independent actions or at least one behaviour frame need to be wrapped, only a new behaviour frame can become their host. One action or multiple dependent actions can be merged into a new higher level action. In the latter case, one needs to set the method's *actionSequence* argument to *true*.

```
Input: operators: list⟨Operator⟩, boolean: actionSequence
Output: operator
if operators.size() == 0 then return;
actions := {};
selections := {};
behaviourFrames := {};
foreach Operator o ∈ operators do
    if class of o is Selection then selections.append(o);
    if class of o is Action then actions.append(o);
    if class of o is BehaviourFrame then behaviourFrames.append(o);
end
if behaviourFrames.size() > 0 OR
(actions.size() > 1 AND actionSequence == false) then
    p_trigger := new Plug(nil, true, source, nil, {}, {});
    wrapper := new BehaviourFrame(nil, 0, {}, {}, {}, p_trigger, {});
else if actions.size() == 1 OR actionSequence == true then
    wrapper := new Action(nil, 0, {}, {}, {});
else
    type := operators.first().type;
    wrapper := new Selection(nil, 0, {}, {}, type, {}, false, 0);
end
operators.sort() with ascending depth();
newOwner := operators.first().owner;
wrapper.addTo(newOwner);
transfer(operators, wrapper);
return wrapper;
```

Algorithm 39: *wrapOperators*()

Selection Wrapper The internal dependencies and ownership relationships of a hierarchical selection operator are resolved by means of the **Selection.update()** method (Alg. 40). It computes the exposed input and output plugs of the selection wrapper and takes possession of them. The level of recursion of the wrapper is determined by the maximal level of any of the internal operators plus one. In order to ensure that its output plugs will provide values, the internal dependencies need to be pre-computed. In particular, the upstream selection chains are

computed for the internal input plugs that are owned by those internal operators that deliver the output values of the high-level selection operator.

Figure 12 shows these mechanisms in detail. The input plugs coloured in blue are the ones that feed into internal operators that will provide the output plugs of the wrapper. The upstream selections of these blue plugs need to be determined. Any input and output plugs that are not internally connected and not constant are coloured in red (connections to the outside of the operator are indicated by red lines). These plugs are adopted and exposed by the new higher-level operator.

```
if self.isZeroLevel() == true then
    self.owner.update();
    return;
end
self.level := 0;
self.inputPlugs := {};
self.outputPlugs := {};
foreach Selection s ∈ self.internalSelections do
    self.level := max(self.level, s.level);
    s.owner := self;
    outputOperator := false;
    foreach OutputPlug p ∈ s.outputPlugs do
        if p.isInternallyConnected() == false then
            p.owner := self;
            self.outputPlugs.append(p);
            outputOperator := true;
        end
    end
    foreach InputPlug p ∈ s.inputPlugs do
        if outputOperator == true then  p.orderUpstreamSelections();
        if p.isConstant() == false AND
        p.isInternallyConnected(self) == false then
            p.owner := self;
            self.inputPlugs.append(p);
        end
    end
end
self.level+ = 1;
self.owner.update();
```

Algorithm 40: Selection.update()

Action Wrapper Hierarchical actions wrap a combination of selection operators and an action or an action sequence. It is important to understand that feeding multiple actions into a high-level action are dependent—they form one

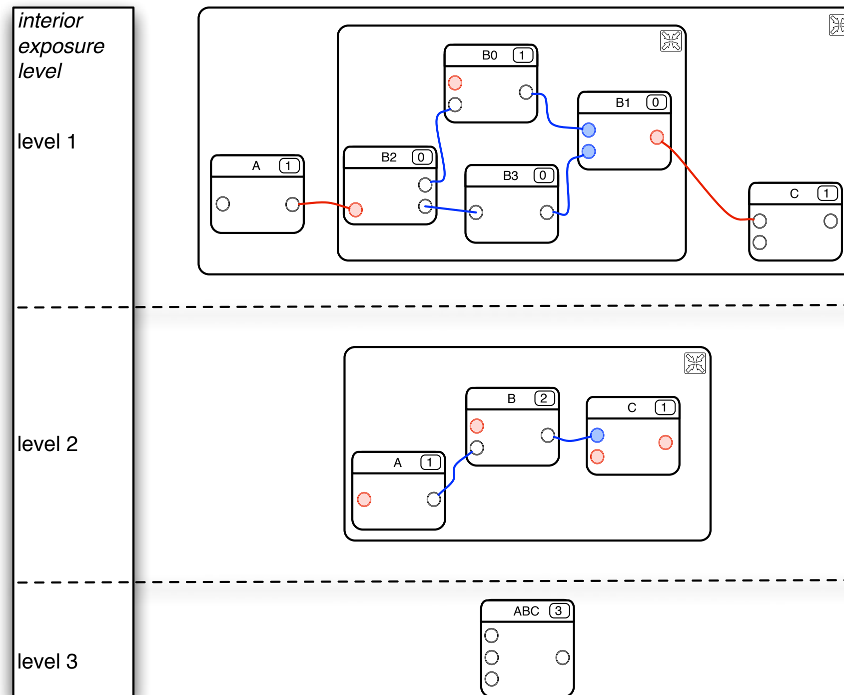

Fig. 12. First the four interior selection operators $B0 \ldots B3$ are wrapped into level-2 operator B. Next, A, B, and C are wrapped into the operator ABC. Red plugs at one level become the exposed plugs at the next higher level. Blue plugs denote those internal input plugs whose upstream dependencies need to be pre-calculated before the wrapper can be computed.

high-level action, and thus either none of them or all of them will be executed (in the given order).

An action operator does not expose any output plugs since it does not provide information. It rather creates, deletes, or changes information. Therefore, unlike *Selection.update*(), **Action.update()** (Alg. 41) does not consider the output plugs of any wrapped selections. It solely exposes a wrapped selection's input plugs, if they are neither constant nor internally connected. Depending on the Selection instance's type (source or target), any adopted input plugs become source or target plugs of the wrapper, respectively.

If the plug of an internal action is internally connected, its internal dependencies are computed by means of the method *orderUpstreamSelections*(). If it is externally connected or not connected at all, it will get exposed by the wrapper.

Updating a behaviour frame is fairly similar to wrapping an action, which is why *Action.update*() covers updating behaviour frames as well. Wrapped be-

haviour frames are treated exactly as wrapped actions, except for the additional consideration of the special activation plug $p_{trigger}$ of a behaviour frame. Since it algorithmically works like a regular source input plug, it is added to the list of input plugs and processed, i.e. ignored or adopted, like the other ones.

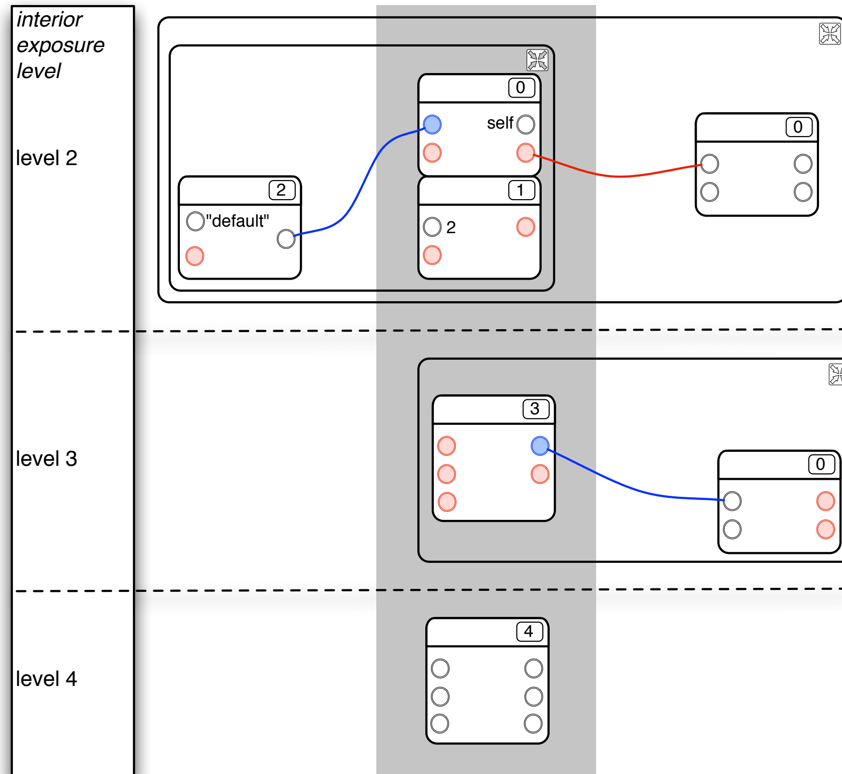

Fig. 13. The combination of interior action and selection operators is wrapped into a level-3 action which exposes five plugs (coloured red at level-2). The blue plug and blue connection on level 2 denote the only internal dependencies of the level-3 action. The process is repeated to forge a level-4 action.

BehaviourFrame Wrapper Wrapping operators into a BehaviourFrame instance is very similar to creating a higher level Action instance as described in the previous paragraph. However, there are two subtle, but important differences regarding the semantics of behaviour frames.

```
if self.isZeroLevel() == true then
    self.owner.update();
    return;
end
self.level := 0;
self.inputPlugs := {};
foreach Selection s ∈ self.internalSelections do
    self.level := max(self.level, s.level);
    s.owner := self;
    for i := 0, i < s.inputPlugs.size(), i++ do
        p := s.inputPlugs.get(i);
        if p.isConstant() OR p.isInternallyConnected() then continue;
        p.type := s.type;
        p.owner := self;
        self.inputPlugs.append(p);
    end
end
informedOperators := self.internalActions.copy();
if class of self is BehaviourFrame then
    informedOperators.append(self.internalBehaviourFrames);
end
foreach Operator o ∈ informedOperators do
    self.level := max(self.level, o.level);
    o.owner := self;
    internalInputPlugs := {};
    if class of o is BehaviourFrame then internalInputPlugs.append(o.p_trigger);
    internalInputPlugs.append(o.inputPlugs);
    for i := 0, i < internalInputPlugs.size(), i++ do
        p := internalInputPlugs.get(i);
        if p.isConstant() then continue;
        if p.isInternallyConnected(self) then
            p.orderUpstreamSelections();
        else
            p.owner := self;
            self.inputPlugs.append(p);
        end
    end
end
self.level+ = 1;
self.owner.update();
```

Algorithm 41: *Action.update*()

1. A behaviour frame can host multiple independent actions. As a consequence, a behaviour frame does not follow the same algorithmic rules as an action. As the source and target plugs exposed by a behaviour frame may stem from independent internal actions, insisting on valid input values before considering all the wrapped actions for execution would be detrimental.
2. It is the purpose of behaviour frames to allow the modeller to manage groupings of arbitrary behavioural "circuitry". For this reason, they provide an on/off switch for the consideration of the wrapped behavioural code (Section 2.8). A behaviour frame exposes an according special activation plug $p_{trigger}$ that is set to a valid value—*true* in the given case—by default which activates the behaviour frame. Like a source plug of an action, $p_{trigger}$ can be connected to an arbitrary selection, and an input value of *nil* will prevent the consideration of the behaviour frame's contents.

Accordingly, $wrapOperators()$ (Alg. 39) instantiates the Plug instance $p_{trigger}$ of type source, with its value set to *true*. Then the function passes $p_{trigger}$ to a newly created behaviour frame. The remainder of the function steps through the chosen operators, transfers them to the corresponding containers, and finally updates the new BehaviourFrame instance in order to resolve its internal dependencies and to compute its exposed input plugs. The update of a BehaviourFrame instance is performed by the *Action.update*() method discussed above.

smartUpdate() The *update*() methods work up the hierarchy. Hence, when updating several operators of one hierarchy, it makes sense to only explicitly call *update*() on those operators that hang lowest on different branches. This minimal overhead approach is implemented by the *smartUpdate*() procedure (Alg. 42).

5 Simulation Routines

Before entering the simulation phase, the swarm script behaviour can be optimized by pruning away actions and behaviour frames that are not evaluable, and thus would not impact any simulation processes.

Each simulation step can further be divided into several sub-steps: (1) Computing the chains of selections that feed into the actions and behaviour frames, (2) identifying triggered actions and behaviour frames, and (3) the actualization of the triggered actions.

5.1 Initialization

During the simulation phase, the agents' interactions are computed. Since the modelling routines update the operators' dependencies, the simulation process does not require further initialization. However, actions and behaviour frames that are not evaluable can be excluded from the set of operators considered for execution upfront. The BehaviourFrame method **BehaviourFrame.update EvaluableOperators()** (Alg. 43) filters the operators contained in a behaviour

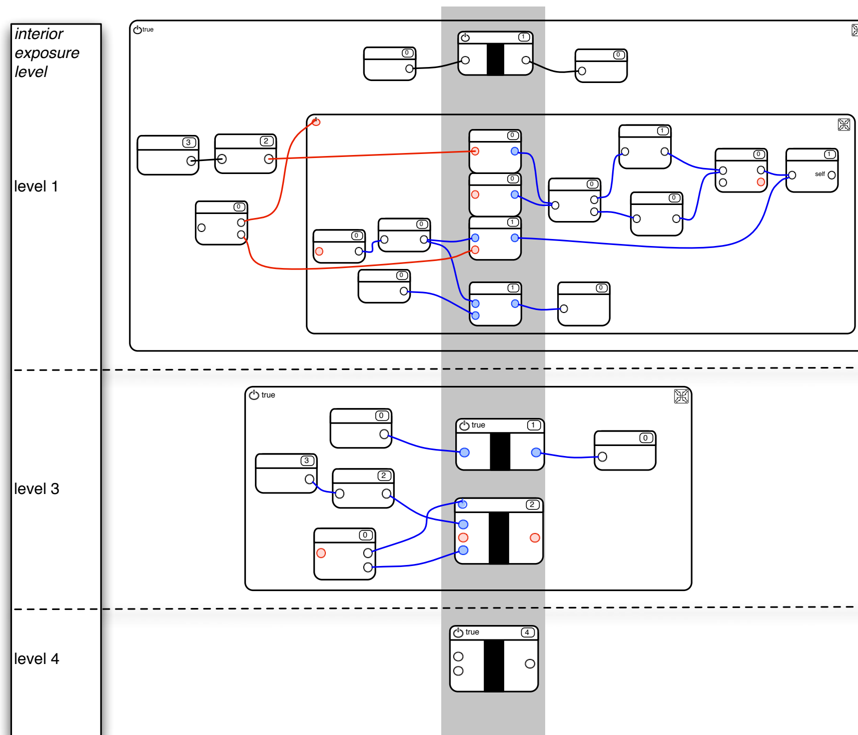

Fig. 14. The inner operators are wrapped into a level-2 behaviour frame. In addition to three source plugs and one target plug, the wrapper exposes its activation plug $p_{trigger}$. External connections are coloured in red, blue connections denote the computation of upstream dependencies of the blue coloured input plugs.

frame accordingly, adding only evaluable operators to its *evaluableActions* and *evaluableBehaviourFrames* containers, respectively. Internally hosted behaviour frames are filtered recursively, resulting in a complete optimization of the Swarm-Script code.

5.2 Compute Selections

At each simulation step, the filtered operator hierarchy is searched for actions and behaviour frames that are successfully triggered. In order to get triggered, the input plugs of actions need to receive valid values (non *nil*), as do the activation plugs $p_{trigger}$ of behaviour frames.

The computation of a selection (**Selection.compute()**, Alg. 44) is performed recursively on its hierarchical structure. At each level, the method **Selection.compute()** (Alg. 44) steps through the operator's internally ordered upstream selections that were calculated during the modelling phase by means

```
Input: operators: list < Operator >
branches := {};
foreach Operator o ∈ operators do
    branchAssigned := false;
    foreach lowestOp ∈ branches do
        if o.below(lowestOp) == true then
            lowestOp := o;
            branchAssigned := true;
            break;
        else if o.above(lowestOp) == true then
            branchAssigned := true;
            break;
        end
    end
    if branchAssigned == false then  branches.append(o);
end
foreach lowestOp ∈ branches do
    lowestOp.update();
end
```

Algorithm 42: *smartUpdate*()

of the method *orderUpstreamSelections*(). On each of those input plugs the method **InputPlug.computeUpstreamSelections()** (Alg. 45) is performed ensuring that all the necessary, upstream linked selections are computed in order and provide the selection operator's results. A non-hierarchically defined selection operator (level-0), overrides the *Selection.compute*() method to store values in its output plugs, considering the values of possible input plugs.

5.3 Identify Triggered Actions

At this point of the simulation step, all the necessary selection values are provided and it can be decided which behaviour frames and actions are triggered. As a result, the hierarchical operator structure is reduced to a list of triggered actions by means of the function **BehaviourFrame.getTriggeredActions()** (Alg. 46). It identifies and collects the triggered actions recursively contained in any considered behaviour frames.

5.4 Compute and Actualize Triggered Actions

Next, the triggered operators are executed. High-level actions recursively iterate through any internal actions. Level-0 actions, similar to level-0 selections, need to override the **Action.compute()** and the **Action.actualize()** methods (Algs. 47 and 48). First, these methods process the information the operator gathers from the source inputs and second, they manipulate the target inputs. Accordingly, **Action.compute()** computes the necessary information, and **Action.actualize()** actualizes the changes of the action.

```
self.evaluableActions := {};
self.evaluableBehaviourFrames := {};
for i := 0, i < self.internalActions.size(), i++ do
    a := self.internalActions.get(i);
    if a.isEvaluable() then self.evaluableActions.append(a);
end
for i := 0, i < self.internalBehaviourFrames.size(), i++ do
    bf := self.internalBehaviourFrames.get(i);
    if bf.p_trigger.isEvaluable() then
        self.evaluableBehaviourFrames.append(bf);
        bf.updateEvaluableOperators();
    end
end
```

Algorithm 43: *BehaviourFrame.updateEvaluableOperators()*

```
outputSelections := {};
foreach Plug p ∈ self.outputPlugs do
    if outputSelections.hasElement(p.zeroLevelOwner) == false then
    outputSelections.append(p.zeroLevelOwner);
end
foreach Selection s ∈ outputSelections do
    for i := 0, i < s.inputPlugs.size(), i++ do
        plug := s.inputPlugs.get(i);
        plug.computeUpstreamSelections();
    end
    s.compute();
end
self.computed := true;
```

Algorithm 44: *Selection.compute()*

```
index := self.upstreamSelections.size() − 1;
while index >= 0 AND self.upstreamSelections.get(index).computed == false
do
    index − −;
end
index + +;
for i := index, i < self.upstreamSelections.size(), i++ do
    self.upstreamSelections.get(i).compute();
end
```

Algorithm 45: *InputPlug.computeUpstreamSelections()*

Output: $\langle Action \rangle$
triggeredActions := {};
for $i := 0, i < self.evaluableActions.size(), i++$ **do**
  a := self.evaluableActions.$get(i)$;
  if $a.isTriggered()$ **then** triggeredActions.$append(a)$;
end
for $i := 0, i < self.evaluableBehaviourFrames.size(), i++$ **do**
  bf := self.evaluableBehaviourFrames.$get(i)$;
  if $bf.p_{trigger}.isTriggered()$ **then**
  triggeredActions.$append(bf.getTriggeredActions())$;
end
return triggeredActions;

Algorithm 46: $BehaviourFrame.getTriggeredActions()$

Asynchronous actualization (Alg. 49) iterates through the triggered actions one after another, computes their required parameters based on the provided selection values and actualizes their changes. This approach does not consider the impact of step-wise state changes on subsequent actions.

Using synchronous actualization (Alg. 50), the calculation of the actions' parameters is concluded during the *Action.compute*() method, before any changes are introduced into the system by means of the *Action.actualize*() method. Furthermore, in order to effectively simulate concurrency, the actualization of state changes needs to be ordered based on a given priority policy.

for $i := 0, i < self.internalActions.size(), i++$ **do**
  a := self.internalActions.$get(i)$;
  a.$compute()$;
end

Algorithm 47: *Action.compute*()

for $i := 0, i < self.internalActions.size(), i++$ **do**
  a := self.internalActions.$get(i)$;
  a.$actualize()$;
end

Algorithm 48: *Action.actualize*()

Input: actions: $list\langle Action\rangle$
for $i := 0,\ i < actions.size(),\ i++$ **do**
  actions.$get(i).compute()$;
  actions.$get(i).actualize()$;
end

Algorithm 49: $asynchronousActualize()$

Input: actions: $list\langle Action\rangle$
sort actions in accordance with order policy $\mathcal{P}$;
for $i := 0,\ i < actions.size(),\ i++$ **do**
  actions.$get(i).compute()$;
end
for $i := 0,\ i < actions.size(),\ i++$ **do**
  actions.$get(i).actualize()$;
end

Algorithm 50: $synchronousActualize()$

5.5 Running the Simulation

Before running the simulation, all the given behaviour frames that store all the agents' behaviours are filtered for evaluable operators by means of the $updateEvaluableOperators()$ method. Then, while the simulation is not terminated, all those optimized behaviours are considered for execution at each step (Alg. 51).

Input: behaviours: $list\langle BehaviourFrame\rangle$
foreach *BehaviourFrame bf in behaviours* **do**
  bf.$updateEvaluableOperators()$;
end
while *not terminated* **do**
  $stepSimulation(behaviours)$;
end

Algorithm 51: $runSimulation()$

The procedure **stepSimulation()** (Alg. 52) steps through a list of behaviour frames, computes the triggered actions and actualizes them. As explained in the preceding paragraphs, the selections linked to the actions' plugs need to be computed in order to verify whether an action is actually triggered or not. Therefore, all the three steps—(1) selection computation, (2) triggered action identification, and (3) triggered action actualization—are performed by the $stepSimulation()$ procedure.

```
Input: behaviours: list⟨BehaviourFrame⟩
forall the Selection instances s do
  | s.computed := false;
end
actionCache := {};
foreach BehaviourFrame bf in behaviours do
  | actionCache.append(bf.getTriggeredActions());
end
actualize(actionCache); (*synchronous or asynchronous*)
```

Algorithm 52: *stepSimulation*()