Implementation of a Visual DEVS Formalism in Both GenGED and AToM^3

Denis Dubé
Modelling, Simulation and Design Lab
McGill University
d3n14@yahoo.com

Abstract

An implementation of a visual DEVS formalism in two meta-modeling tools, GenGED and AToM^3 is described. In particular an attempt is made to exploit the specialized nature of these tools to build the implementation in as little time as possible. The need for graphical layout, an easy to use user interface, model correctness checking, and generation of DEVS simulation code push these tools to the limit. This paper can be read as an informal comparison of the approaches the two tools have taken to the problem of rapid prototyping of visual languages.

1. Introduction

A visual DEVS formalism is implemented using the the research tools GenGED and AToM^3. DEVS, the Discrete Event Specification System, is an excellent simulation formalism due to its expressivity and formal underpinnings that allow for detailed analysis. However, DEVS models can grow to be as large as 400,000 lines of code as at leas one example in [1] shows. This makes it difficult for users of the formalism to keep a mental map of what the model really represents, hence suggesting a need for a visual implementation of the formalism.

Traditionally, visual modeling tools are manually coded from the ground up for a specific formalism. Although this usually leads to an efficient and user-friendly tool, it is extremely time-intensive and cost inefficient. The alternative is meta-modeling, whereby the desired visual formalism is itself specified in the form of a model, and the tool is automatically generated from this model. Two research tools that can do this are GenGED and AToM^3.

GenGED, [7,8,10,11], is the first tool used to implement a visual DEVS formalism in this paper. The key benefit of using this system is the tight integration of a layout constraint manager PARCON. Just as letters need to be aligned horizontally in a word, visual languages require visual components to be positioned in a restricted number of ways. Thus it shall be interesting to see what the final output of this tool will be.

AToM^3, [4,5], is the second tool used. The advantage of this tool will lie in with its great flexibility with regards to the plugging in of new code in the rapid proto-typing language Python. This will be especially critical, as AToM^3 does not currently provide any native support for either hierarchical structures nor hierarchical layout.

Regardless of the meta-modeling tool used, the end result should be a useable visual DEVS modeling tool. It should be useable in the sense that small to medium sized DEVS problems can be specified with it in reasonable time and that the specification can then be outputted to a simulator so that results can be obtained.

2. Background

Prior to the implementation of a visual DEVS tool, the literature behind both meta-modeling tools is examined. Moreover, the DEVS formalism itself requires some scrutiny if a reasonable implementation is to be built.

2.1. GenGED

The GenGED meta-modeling tool, short for Generation of Graphical Environments for Design, is designed to make the visual definition and manipulation of visual languages relatively quick and easy. Also built-in are modules that make simulation and animation trivial to implement [6,9]. The most important consideration for the implementation of a visual formalism however, is the integration of a layout constraint manager called Parcon. This allows the very difficult task of positioning and re-sizing visual objects in a logical way, to be specified very easily, with no need for coding in most cases. Parcon does have a drawback however, only binaries of it are available, and they work only on very old Linux distributions or Solaris operating systems. This is a shame since GenGED is otherwise a platform independent Java program.

GenGED can be broken down into three major components. The alphabet editor, the grammar rule editor, and the simulation and animation editors. However the simulation and animation capabilities of GenGED fall outside the scope of this paper.

2.1.1. Alphabet Editor

The alphabet editor is the component of GenGED whereby the visual symbols and the relations between these symbols, are specified for a given formalism. This involves three different specification editors according to [7].

The first of these editors is the Graphical Object Editor. In this editor, visual symbols are defined for each visual entity in the formalism. This can be as simple as creating a circle for a state and a line with an arrowhead for an arrow, or can involve creating a double rectangle with a layout constraint enforcing one rectangle always remains above the other.

The second editor is the TypiEditor. Entities in the formalism to be created are assigned either a visual symbol defined in the Graphical Object Editor or are given a non-visual placeholder. Additionally, displayable attributes, such as strings for the names, are instantiated, although they are not tied to any entity at this point.

The third editor, the ConEditor, then allows you to specify the actual relationship between the various entities, including which attributes belong to which entities. Moreover, layout constraints can be set throughout this process, so attributes can appear anchored to the top of an entity, and arrows anchored to the borders of the entities it connects.

The visual formalism specification these three editors creates is sufficient for a prototype tool to be generated. The prototype tool allows for the creation of all the entities in the formalism and for testing the layout.

2.1.2. Grammar Rules Editor

The next major component of GenGED involves the specification of grammar rules for the formalism. The first type of grammar to create is the syntax grammar. The motivation for syntax grammars lies in their ability to ensure that a model being constructed in a formalism is always correct. In precise terms, this means that the model is always in the set of all possible valid models and never enters a configuration that would constitute an invalid model for the given formalism. Syntax grammars are enforced at the level of each manipulation to the model. This means that it is impossible to interactively modify the model in such a way that even a temporarily incorrect model occurs.

In an interactive setting, it is sometimes highly desirable to be able to modify a model into a temporarily incorrect configuration. This is sometimes called freehand-editing. To enable this, syntax grammars must be weakened to allow some incorrect configurations, but now model checking is compromised! Ideally we would like to have our cake and eat it too, hence the motivation for using a parse grammar. The parse grammar is run by the user to check the correctness of the model and works by either reducing the model to an empty one using just the valid graph grammar rules, or in the opposite direction by growing an empty model to the current model.

Specification of a grammar in GenGED begins with an alphabet defined in the alphabet editor. GenGED then automatically generates basic grammar rules from this alphabet. There are three types of generated rules: insertion, deletion, and attribute modification. They essentially do the obvious thing; insert entities and relationships along with their associated attributes, delete entities and relationships, and change the attribute values. A typical insert rule for an arrow would have a source and target entity on the left hand side of the rule, and a source and target entity connected by an arrow on the right hand side.

Using the automatically generated rules as primitives, rules of greater complexity can be created. As a trivial example, in a class diagram formalism, one might want to a single rule to insert both a new class diagram object and a new class inside of it. Thus a new rule would be created with an empty left and right hand side. The primitive rule for class diagram insertion would be applied to the right hand side, followed by the primitive rule to insert a class inside a class diagram.

From a set of primitive and composite rules, GenGED allows the creation of a syntax or parse grammar for use in a final formalism specification. The final specification simply takes as input the alphabet, syntax, and (optionally) parse grammar to create a tool that can be used to create models in the specified formalism. Note that the final specification can also include simulation and/or animation grammars as well.

2.2. AToM^3

The second meta-modeling tool, AToM^3 or A Tool for Multi-formalism and Meta-Modeling, is also designed to make the creation of visual formalism easy. The multi-formalism part of the name stems from the fact that models can be created using more than one formalism at once or can be transformed from one formalism to another. This tool also implements a graph re-writing system and this is what is typically used to perform the aforementioned formalism transformation. Unfortunately, at the time of this writing the tool lacks the flexible graphical layout constraints available in GenGED, so layout is typically achieved by hard-coding it into the formalism, or using some of the few layouting methods available. The tool is written entirely in Python and has been successfully run on Windows, Linux, and Mac operating systems.

In AToM^3, visual formalisms are generated from a model. Indeed, the Entity Relationship diagram formalism is generated from a model in that very same formalism. Although the Class Diagram formalism is far superior to Entity Relationship due to the inheritance mechanism, it was not mature enough to use until recently and has not made a big impact on the AToM^3 literature yet. Thus the focus here shall be on Entity Relationship diagrams for the generation of new formalisms.

An Entity Relationship diagram consists of just an entity object and a relationship object that links together multiple entities, including a loop on a single entity. For each entity and relationship, one can specify a name, graphical appearance, cardinality, attribute list, constraint list, and action list. The graphical appearance consists of a set of primitive shapes and can include attributes, which can change at run-time, from the attribute list. The cardinality is just like the mechanism employed in UML class diagrams. The attribute list consists of any number of attributes that entities in the generated formalism will posses, such as strings for names. The constraints list allows the formalism maker to specify constraints that catch certain events and run code to check that the model is correct. Finally, the actions list allows the creation of actions that occur at a specific event, such as performing a layout operation.

Thus in AToM^3, the generation of a formalism is accomplished by creating a model in an existing formalism, such as Entity Relationship diagrams, and filling out the necessary details directly in the visual model. Unlike GenGED, no syntax or parse grammars are defined, but instead cardinalities and constraint code are used to ensure the correctness of the model.

2.3. DEVS

A flavor of DEVS known as classic DEVS, shorthand for Discrete EVent System Specification, is now quickly summarized. The difference between DEVS and many other simulation models is the fact that it is derived from mathematical dynamical system theory and thus has a formal framework supporting it. Moreover, despite the discrete nature of the simulation, continuous systems can be successfully approximated with it. DEVS has been applied or is being applied to large real-life problems, including next generation GPS systems, spaced based laser systems, and controllers for blast furnaces used in steel production [1]. An informal description of DEVS, taken from the inventor of the formalism, Ziegler [12], follows.

DEVS models have input and output ports through which all interaction with the external world takes place. By coupling together output ports of one system to input ports of another, outputs are transmitted as inputs and acted upon by the receiving system. Thus, there are two types of DEVS models, *atomic* and *coupled*. An atomic model directly specifies the system's response to events on its input ports, state transitions, and generation of events on its output ports.

A coupled model is a composition of DEVS models that presents the same external interfaces as do atomic models. For example, in Figure 1, CM is a coupled model with four components. A coupled model specifies three types of coupling:

- *external input* – from the input ports of the coupled model to the input ports of the components (e.g., from start of CM to start of counter)
- *internal* – from the output ports of components to input ports of other components (e.g., from explosion of bomb to strike of target)
- *external output* – from the output ports of components to output ports of the coupled model (e.g., from damage of target to damage of CM)

Although arbitrary fan-out and fan-in of coupling is allowed, no self-loops are permitted. DEVS is closed under coupling, which means that a coupled model can itself be a component within a higher level coupled model, leading to hierarchical, modular model construction.

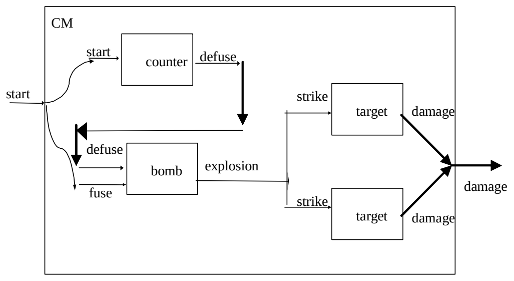

Fig. 1. Bomb defusal DEVS example

3. Prototype DEVS Tool

In this section, the development of a prototype tool that allows us to create basic visual models in the DEVS formalism, using both GenGED and AToM^3 is described. First the necessary components for a visual DEVS formalism are enumerated.

There are four different entities: a coupled DEVS, an atomic DEVS, a state, and a port. Visually they, appear as a rectangle and a name, a slightly different rectangle and a name, a circle and a name, and a port and a name respectively. Relating these entities are four insideness relations. A coupled DEVS can be inside a coupled DEVS, an atomic DEVS can be inside a coupled DEVS, a state can be inside an atomic DEVS, and a port can be on the border of an atomic or coupled DEVS. Finally, there are two types of arrows, channels between ports and transitions between states. The transitions can be external or internal, thus there are a total of three distinct arrows.

Given these basic visual primitives, a minimum amount of work is done, in both meta-modeling tools, to generate a limited-functionality DEVS modeling tool. Bear in mind that although the two tools require essentially the same steps to be performed, that the order is quite different.

3.1. GenGED

The first step in GenGED involves the purely visual specification of each entity and arrow. In the current version of the software (version 1.1, dated 05/11/2004), the Graphical Object Editor and the TypiEditor have been merged into a single editor called the Symbol Editor. Thus when implementing a coupled DEVS entity in the DEVS formalism, we must define two separate symbols, a rectangle primitive and an instantiation of a string for the name attribute. Arrows, such as for transitions and channels, are simply line primitives with an arrow head. Note that layout constraints are not employed at this stage since no composite visual objects are needed for a simple implementation of the DEVS formalism.

The second step is to specify the actual relationships between the visual symbols. For the coupled DEVS, this means we must attach the associated name attribute to it. Furthermore, a layout constraint is specified such that the name attribute is anchored to the top of coupled DEVS object at all times. This is fairly straightforward to do, one simply selects the anchor top constraint, and then selects a target and source object for the constraint.

Completing the second step is the insideness and arrow relations. Insideness is very straightforward, for a coupled DEVS inside a coupled DEVS, one simply selects as source one instance of coupled DEVS and as target another instance. Then one just chooses the inside constraint, select source and target again, and layout is taken care of. Arrows are somewhat more complicated, as it is necessary to specify first what the start of the arrow connects to, and then what the end of the arrow connects to in two separate connection entries. Two different layout constraints are also needed, a typical constraint for this will force the arrow end point to lie on the border of the target object.

A prototype visual DEVS modeling tool can now be generated. This tool allows us to create each graphical symbol defined and connect them together according to the connections defined. Moreover, layout constraints are enforced. Unfortunately, creating a coupled DEVS, for example, requires first creating the rectangle symbol, then the associated name attribute, and finally using an operation to tell GenGED that these two belong together. Creation of arrows is just as user unfriendly, since the arrow must first be created, and then for each endpoint the correct attachment operation must be selected, followed by selection of the arrow and the target object.

Thus a prototype DEVS tool is implemented in GenGED. Unfortunately, the prototype has an interface that is too restrictive to create anything more than a trivial model, no model checking, and no capacity for extra capabilities. The latter means that it is not possible to add arbitrary action code that would, for example, allow us to generate code. However, automatic layout works very nicely and it is already possible to save and load models.

3.2. AToM^3

Since the meta-modeling tool AToM^3 allows meta-models to be explicitly modeled, the first step is to choose the meta-modeling environment in which to model the DEVS formalism. Two possibilities for this are Entity Relationship diagrams and Class diagrams. As mentioned previously, the latter were not mature enough at the time the DEVS formalism was implemented, so the former was chosen, despite its lack of inheritance.

The second step is then to create one entity for each of the entities described at the beginning of this section, and one relationship for each of the insideness relations and arrows. Relationships are specified by simply drawing arrows between the entities or around a single entity. In contrast with GenGED, the relationships are defined in a far more natural and visual fashion. On the other hand no mechanism for setting up layout constraints exists.

As a third step, cardinalities are specified for each entity and relationship. This is just like the UML (Unified Modeling Language) notion of cardinalities. For example, a coupled DEVS can contain 0 to N atomic or coupled DEVS, but can only be contained by another coupled DEVS 0 to 1 times. These cardinalities are enforced at run-time and provide at least some limited model correctness checking. This step is shown in figure 2.

In the fourth step, generative attributes are added to the model. For example a coupled DEVS will receive a string attribute for name. As a more complicated example, a state receives not only a string attribute for a name, but also two text attributes for storing time advance and output function code and a Boolean attribute to indicate if it is the default state.

Finally, the visual appearance of the DEVS formalism is specified. Note how this is done in a completely opposite order from GenGED. For each entity, basic primitives are drawn using a Paint-like icon-editor tool. Connection ports are added for arrows to automatically attach to, and generative attributes like names are added as well. No special layout constraints are added nor seemingly needed here; the entity will appear as it is drawn, although the name attribute will change and the scale may be modified. Lastly, a special dialog is used to specify the appearance of relationships as either arrows or as non-visual for the case of insideness relations.

A prototype can now be generated from this Entity Relationship model. The prototype has buttons for creating each entity and arrow relationships are drawn by selecting one entity as a source and another as a target. Although it is possible to draw a DEVS model, the lack of any notion of layout is a serious bottleneck. Worse, it is not possible to send objects from foreground to background or vice versa, so creating a model requires judicious ordering of entity creation. However, models can be saved and loaded, and it would be possible to immediately generate code for a DEVS simulator using either a graph grammar or by simply adding a button that runs arbitrary code generating… code.

Thus a prototype visual DEVS formalism is now implemented in both meta-modeling tools. In both cases, this is done in a matter of hours. The capabilities are not yet very impressive though, so the implementation of a finalized tool is described next.

4. Final DEVS Tool

A full featured extension, or an attempt thereof, of the basic prototype visual DEVS formalism described in the previous section is now described. The extension requirements for the GenGED and AToM^3 implementations are nearly complete opposites. In GenGED, an improved user interface for constructing models is desperately needed, while in AToM^3 it is layout constraints that are needed. In both cases implementations, model checking and code generation need to be added.

4.1. GenGED

A complete specification of a visual DEVS formalism in GenGED requires the definition of grammar rules. Given that an alphabet has already been defined, GenGED automatically generates primitive grammar rules for the insertion and deletion of entities and relationships as well as the modification of attributes. However, even if the alphabet defined in the prototype tool worked perfectly, it does not necessarily follow that the generated grammar rules will be correct. For example, one might get a primitive rule for inserting an atomic DEVS that has a coupled DEVS in the left hand side. Worse only a single primitive rule is generated for the insertion of any given entity, thus no primitive or composite rule will allow us to create an atomic DEVS on its own. This merely means that the specification of a visual formalism must be given with greater care in the alphabet editor.

However, let us assume correct grammar rules are generated, and that composite rules are not needed in the DEVS formalism. GenGED then takes the grammar rules as is, and generates a syntax grammar specification. Yet another editor simply takes as input the alphabet and the syntax grammar and generates a final visual DEVS modeling tool. Note that the parse grammar is also being ignored since the focus is on functionality rather than model correctness at this point.

The final visual DEVS modeling tool now allows us to create entities and their associated attributes in one click. Seemingly the user interface issue of the prototype tool is resolved. Unfortunately, the creation of new relations and all specific deletions have become much harder than in the prototype tool. This is due to the heavy reliance on graph grammars in GenGED. In order to connect a channel arrow for example, it requires 8 user clicks, to select the correct insertion rule, select the source DEVS port in the model, the source DEVS port in the left hand side of the rule, etc.

Moreover, GenGED does not have any built-in support for adding arbitrary code even at this point. This means it is not possible to generate code that a DEVS simulator would understand. Our only option would be to implement a graph grammar DEVS simulator inside GenGED, assuming a meaningful DEVS simulator can even be implemented as a graph grammar.

Thus the attempt to implement a visual DEVS formalism in GenGED ends here for two reasons: 1) The user-interface, while quite interesting and perhaps of research value, is completely innadequate for use in the creation of non-trivial models and 2) It is not possible to generate code from GenGED models in the current implementation.

4.2. AToM^3

In the second AToM^3 implementation phase, sorely needed graphical layout constraints are added. Since AToM^3 has no layout constraint manager custom layout code must be developed and/or built-in layout routines must be manually called. This will require explicitly keeping track of the hierarchy implicit in a DEVS model, which is also not automatically handled by AToM^3 yet.

To keep track of the hierarchy of the DEVS model, simple actions are added to each entity. For example, in relationships, Connect actions are required to inform the source of the relationship of a new child entity, and the target of the relationship of its new parent. Likewise Delete actions, in a relationship, will remove children and reset the parent status of the previously connected entities. This is sufficient to create and maintain a hierarchy, not necessarily just for the DEVS formalism. See figure 3, for what the final Entity Relationship model with these actions looks like. Note that actions like "setParent" and "addChildren" are not really actions but are just methods called by the Connect actions of the relationships.

Now that a hierarchy is constructed, it is tempting to simply call a method that performs hierarchical layout each time part of a model is modified. Indeed the first implementation was done as such, but additional complexities, such as allowing for drag-and-drop hierarchical add and removal, resulted in spaghetti code inside the actions. Thus the reactive behavior of the DEVS formalism was modeled in the DCharts formalism. DCharts are a form of statecharts developped by Thomas Feng [13].

4.2.1. Entity Relationship diagrams and DCharts

Merging what has become two models in two different formalisms, Entity Relationship and DCharts, is not too difficult in AToM^3. The current implementation accomplishes this by having entities in the Entity Relationship model instantiate a compiled statechart upon creation. In other words, the DCharts model separately generates a statechart description which is then compiled into executable code. Actions in the Entity Relationship model can then send messages to the compiled statechart.

In the Entity Relationship diagram shown in figure 3, this corresponds to "initilize" actions and "Connect" actions. These actions pass several messages to the statechart shown in Figures 4 and 5, causing the statechart to switch to describing the behaviour of the given entity or relationship. Note that a statechart could have been created for each entity and relationship, however the current method was chosen as it minimizes redundant code, improving robustness and understandability.

To understand how the Entity Relationship diagram and DCharts models complement each other an example is given. Starting with an empty model, the user creates a coupled DEVS. From the Entity Relationship diagram, this triggers a Create event, and thus the "initilize" action. This in turn instantiates the statechart and sends the messages: "createComposite", "isCoupled", and "Drag-n-drop". For the last message, a guard checks if the coupled DEVS has a parent, since it does not, a check is made for whether or not it is inside another coupled DEVS in which case the user is asked if they wish to perform the hierarchical addition of one coupled DEVS inside another. Finally, the layout method coded into the action "doLayout" in the Entity Relationship diagram for coupled DEVS is triggered. The "doLayout" method is simply several hard-coded layout routines that eliminate overlapping at a given hierarchical level (using a built-in force transfer implementation), position and re-size composite components to fit their children, position ports along the border of the composite components, and use built-in AToM^3 code to redraw the arrows automatically.

Continuing this example, suppose an atomic DEVS is now added directly over the existing coupled DEVS. Again a statechart is instantiated and the following messages sent to the statechart: "createComposite", "isAtomic", and "Drag-n-drop". Since the new atomic has no parent, the hierarchical adder asks us to add it or not, and we allow it to add it. Now the coupled DEVS is re-positioned and re-sized to ensure the atomic DEVS fits inside. Now suppose we select the coupled DEVS. This results in a "Select" message being generated, label 1 in figure 5. This highlights the coupled DEVS and generates a "recursiveSelect" message, label 2 in also in figure 5, being generated. Thus the atomic DEVS is selected and highlighted as well. This means that if we now use drag or delete operations, both the coupled and atomic DEVS will be acted upon.

Taking this example a little further, suppose we now select just the atomic DEVS and then drag it outside of the coupled DEVS. As soon as we drop the atomic DEVS, the "Drag-n-drop" message arrives, and since the atomic DEVS has a parent, it is label 3 of figure 5 that is triggered. This causes a check to be made for whether or not the atomic DEVS is still inside the coupled DEVS. Since it is not, the user is prompted as to whether or not they wish to hierarchically remove the atomic DEVS. If not, the coupled DEVS is re-positioned and re-sized appropriately to contain it. Otherwise, the atomic DEVS is disconnected, and a "Drag-n-drop" message is generated for the atomic DEVS, allowing it to be immediately added to another coupled DEVS. In both cases, the parent of the atomic DEVS, the coupled DEVS, receives the message "recursiveDrop" which is propagated to the root parent. This message causes layout to be performed, so that each hierarchical level properly contains its children.

Although the example described is quite trivial, it should give the reader some intuition as to how Entity Relationship diagrams and DCharts have been merged to create a visual DEVS formalism with graphical layout handling.

4.2.2. Model checking

Some form of model checking is already performed thanks to the run-time enforcement of the cardinalities of each entity and relationship, shown in figure 2. However cardinalities are insufficient. For example, it is currently possible to have more than one root node, such as by creating two coupled DEVS with no relation to each other. To deal with this a "checkValidity" action is added to ensure the uniqueness of the root node, shown in figure 3. An action rather than a constraint is used to signal the error to the user, such as by red-highlighting, so that the user has the freedom to interactively edit models. Were a constraint used, incorrect temporary models with two roots would not be possible.

Other model checks are also implemented, including one as a constraint, such that default states are unique in an atomic DEVS, that internal and external transitions only link states of the same atomic DEVS, and that the places where channels between ports of a given type are possible is restricted.

All this additional model checking is essentially hard-coded, and is checked at the appropriate trigger event. It is not as elegant as a syntax or parse grammar, but it is actually faster and easier to create, especially when compared with the creation of an efficient parse grammar.

4.2.3. Code generation

To complete the visual DEVS formalism, a means of exporting the information contained in the model to a simulator is needed. This can be done either with graph grammars or in an entirely hard-coded fashion. Fortunately, a hard-coded implementation by Ernesto Posse was already available and could be adapted to this formalism [3]. This allows models in the visual DEVS formalism to be converted to executable simulation code using [4], which is ready to run as is.

5. Discussion and Future Work

Thus a visual DEVS formalism, has been implemented in the meta-modeling tools GenGED and AToM^3 to varying degrees of completeness. Despite the strong layout and model checking support in GenGED, issues involving the user interface and the lack of code generation capability result in a very basic and incomplete implementation of DEVS. In AToM^3, the issues involved are rather the opposite, but the lack of a dedicated layout constraints and of a grammar based model checker are overcome using its flexibility with regards to the addition of arbirtrary action and constraint code. Of course this means that considerable time is required, about a week in fact, but that is still far cry from the amount of time that would have been required to build the tool from scratch.

The interested reader can find the AToM^3 models used to build the DEVS formalism, along with AToM^3 itself at: http://msdl.cs.mcgill.ca/people/denis/. The link to DEVS is on the left in the navigation toolbar.

The current implementation is by no means finished however. The layout mechanism is currently designed around the interactive user session and cannot build a layout from a randomized model automatically. For example this occurs when you transform a model in another formalism to DEVS. Also, by requiring the use of states and arrows inside an atomic DEVS, the infinite possibilities that are possible with a coded output function are severely restricted, thus suggesting the need for a special atomic DEVS that lets the user code such things explicitly. Another useful addition, in many situations, would be a visual notation for N number of components connected together. Lastly, the ability to completely hide some parts of the hierarchical structure, while continuing to display and interact with the rest, would greatly improve the usability of the visual formalism, especially when used on large problems.

6. References

[1] Bernard P. Zeigler, "DEVS Today: Recent Advances in Discrete Event Based Information Technology", *Powerpoint*, MASCOTS' 03, Orlando, FL, October 2003, http://www.lsis.org/vie_du_labo/uploads/Recent_advances_in_discrete_ev_36.ppt

[2] John Kitzinger and Prasanna Sridhar, "DEVS TUTORIAL", *Powerpoint*, University of New Mexico, July 2002, http://vlab.unm.edu/documents/Tutorial1.ppt

[3] Hans Vangheluwe, Jean-Sébastien Bolduc, Ernesto Posse, and Spencer Borland, "pythonDEVS", *Webpage*, 2002, http://moncs.cs.mcgill.ca/MSDL/research/projects/DEVS/

[4] Ernesto Posse and Jean-Sébastien Bolduc, "Generation of DEVS modelling and simulation environments", In A. Bruzzone and Mhamed Itmi, editors, Summer Computer Simulation Conference. Student Workshop, pages S139 - S146. Society for Computer Simulation International (SCS), July 2003. Montréal, Canada. http://www.cs.mcgill.ca/~hv/publications/03.SCSC.DEVScodegen.pdf

[5] Hans Vangheluwe and Juan de Lara, "Domain-Specific Modelling for analysis and design of traffic networks", Winter Simulation Conference, pages 249 - 258. IEEE Computer Society Press, December 2004. Washington, DC. http://www.cs.mcgill.ca/~hv/publications/04.Wintersim.Traffic.pdf

[6] C. Ermel and R. Bardohl, "Sencario Views for Visual Behavior Models in GenGED", Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02), Satellite Event of First Int. Conference on Graph Transformation (ICGT'02), Barcelona, Spain, Oct. 2002, pages 71-83, http://www.tfs.cs.tu-berlin.de/~rosi/publications/EB02_gtVMT.ps.gz

[7] R. Bardohl, "A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars", Proc. IEEE Symposium on Visual Languages (VL'98), Sept.1998, Halifax, Canada, pages 48-55, http://www.tfs.cs.tu-berlin.de/~rosi/publications/Bar98_VL98.ps.gz

[8] Bardohl,R., Ermel,C., and Weinhold,I., "GenGED - A visual definition tool for visual modeling environments", Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), pages 407-414, Sept./Oct., 2003, Charlottesville/Virgina, USA.
Also in Lecture Notes in Computer Science (LNCS) 3062, Springer, 2004, pages 413-419, http://www.tfs.cs.tu-berlin.de/~rosi/publications/BEW03_AGTIVE03.ps.gz

[9] C. Ermel and R. Bardohl, "Sencario Views for Visual Behavior Models in GenGED", Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'02), Satellite Event of First Int. Conference on Graph Transformation (ICGT'02), Barcelona, Spain, Oct. 2002, pages 71-83, http://www.tfs.cs.tu-berlin.de/~rosi/publications/EB02_gtVMT.ps.gz

[10] R. Bardohl, "A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars", Proc. IEEE Symposium on Visual Languages (VL'98), Sept.1998, Halifax, Canada, pages 48-55, http://www.tfs.cs.tu-berlin.de/~rosi/publications/Bar98_VL98.ps.gz

[11] Bardohl,R., Ermel,C., and Weinhold,I., "GenGED - A visual definition tool for visual modeling environments", Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), pages 407-414, Sept./Oct., 2003, Charlottesville/Virgina, USA.
Also in Lecture Notes in Computer Science (LNCS) 3062, Springer, 2004, pages 413-419, http://www.tfs.cs.tu-berlin.de/~rosi/publications/BEW03_AGTIVE03.ps.gz

[12] Zeigler, B. "DEVS Theory of Quantized Systems," *Electronic document*, 1998, http://www.acims.arizona.edu/PUBLICATIONS/CDRLs/UnivArizonaCDRL1.pdf

[13] Thomas Huining Feng. Dcharts, a formalism for modeling and simulation based design of reactive software systems. M.Sc. dissertation, School of Computer Science, McGill University, February 2004. http://msdl.cs.mcgill.ca/MSDL/people/tfeng/thesis/thesis.pdf

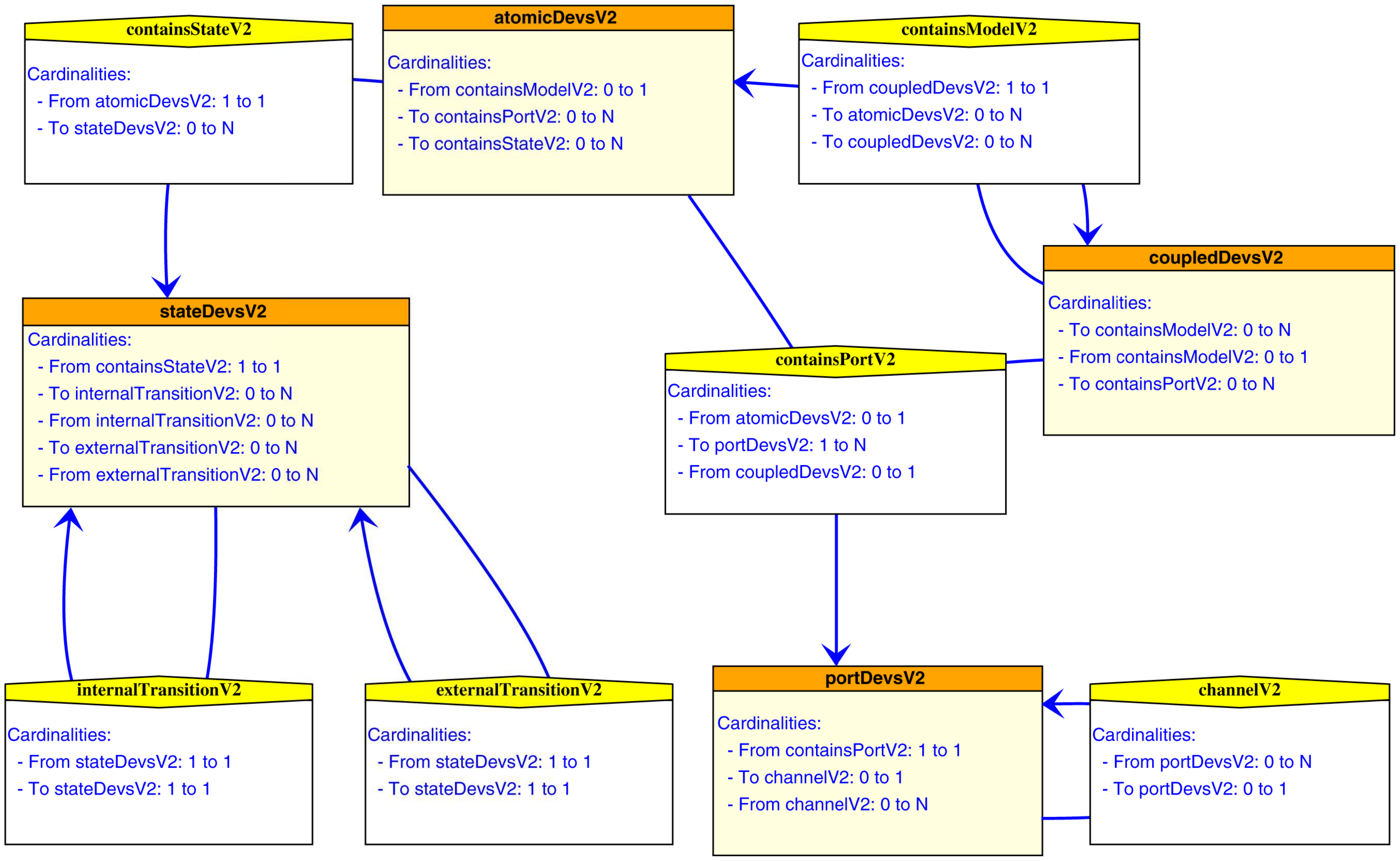
containsStateV2
Cardinalities:
- From atomicDevsV2: 1 to 1
- To stateDevsV2: 0 to N
atomicDevsV2
Cardinalities:
- From containsModelV2: 0 to 1
- To containsPortV2: 0 to N
- To containsStateV2: 0 to N
containsModelV2
Cardinalities:
- From coupledDevsV2: 1 to 1
- To atomicDevsV2: 0 to N
- To coupledDevsV2: 0 to N
coupledDevsV2
Cardinalities:
- To containsModelV2: 0 to N
- From containsModelV2: 0 to 1
- To containsPortV2: 0 to N
stateDevsV2
Cardinalities:
- From containsStateV2: 1 to 1
- To internalTransitionV2: 0 to N
- From internalTransitionV2: 0 to N
- To externalTransitionV2: 0 to N
- From externalTransitionV2: 0 to N
containsPortV2
Cardinalities:
- From atomicDevsV2: 0 to 1
- To portDevsV2: 1 to N
- From coupledDevsV2: 0 to 1
internalTransitionV2
Cardinalities:
- From stateDevsV2: 1 to 1
- To stateDevsV2: 1 to 1
externalTransitionV2
Cardinalities:
- From stateDevsV2: 1 to 1
- To stateDevsV2: 1 to 1
portDevsV2
Cardinalities:
- From containsPortV2: 1 to 1
- To channelV2: 0 to 1
- From channelV2: 0 to N
channelV2
Cardinalities:
- From portDevsV2: 0 to N
- To portDevsV2: 0 to 1

Fig. 2., Entity Relationhip diagram specifying the DEVS formalism, cardinalities shown

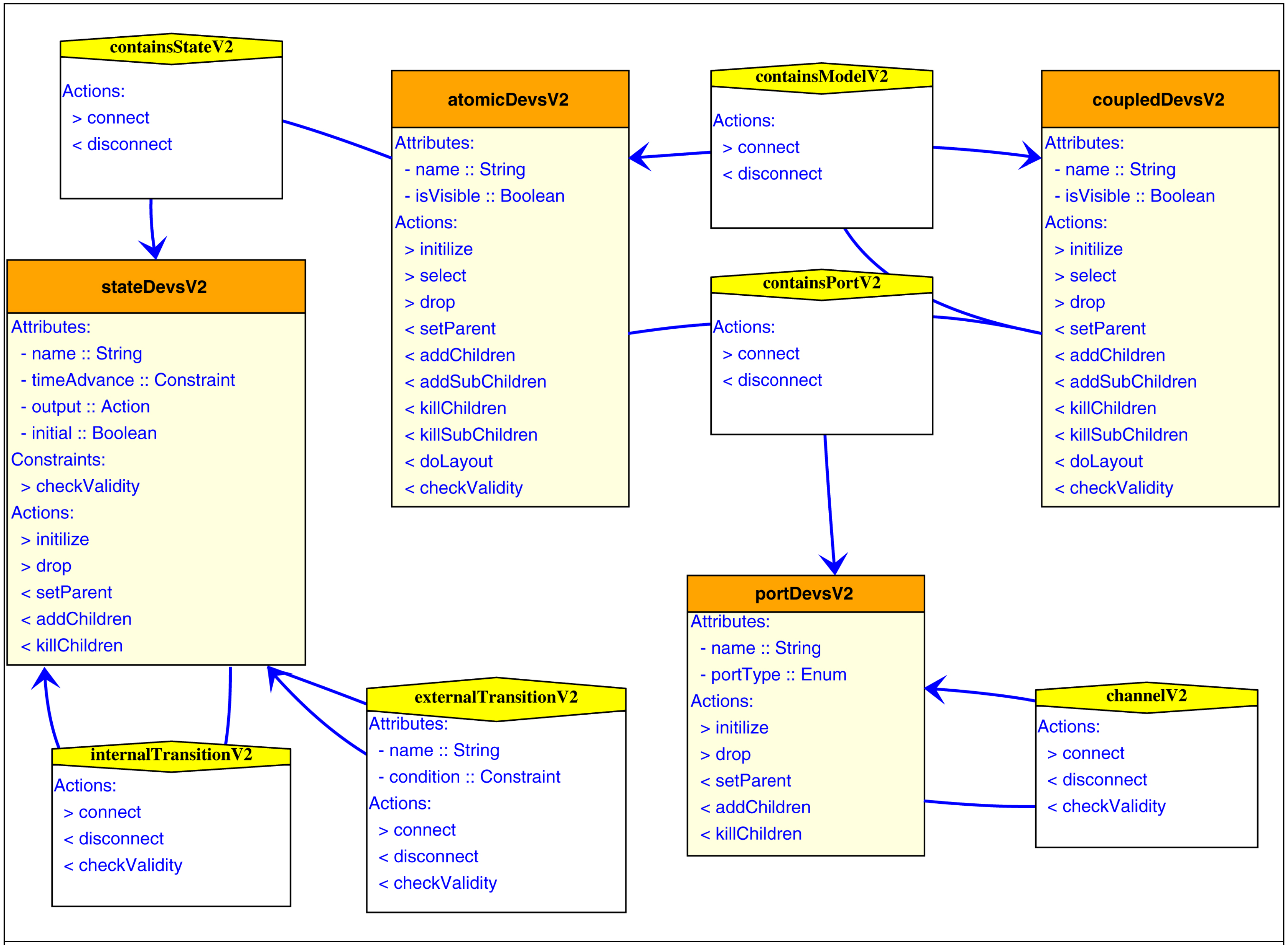

Fig.3., Entity Relationhip diagram specifying the DEVS formalism; attributes, constraints, and actions shown

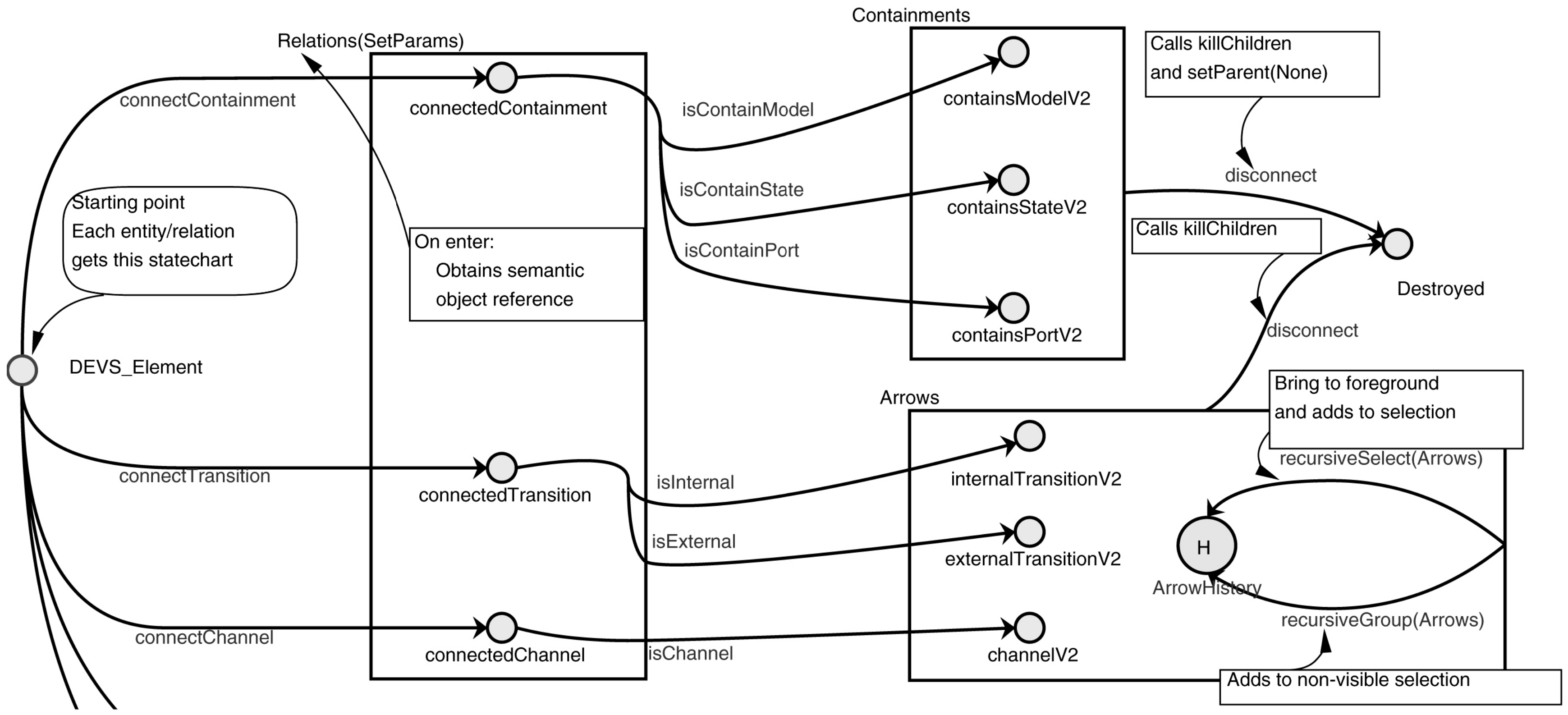

Fig. 4., DCharts model of DEVS reactive behaviour; top part for insidness relations and arrows shown

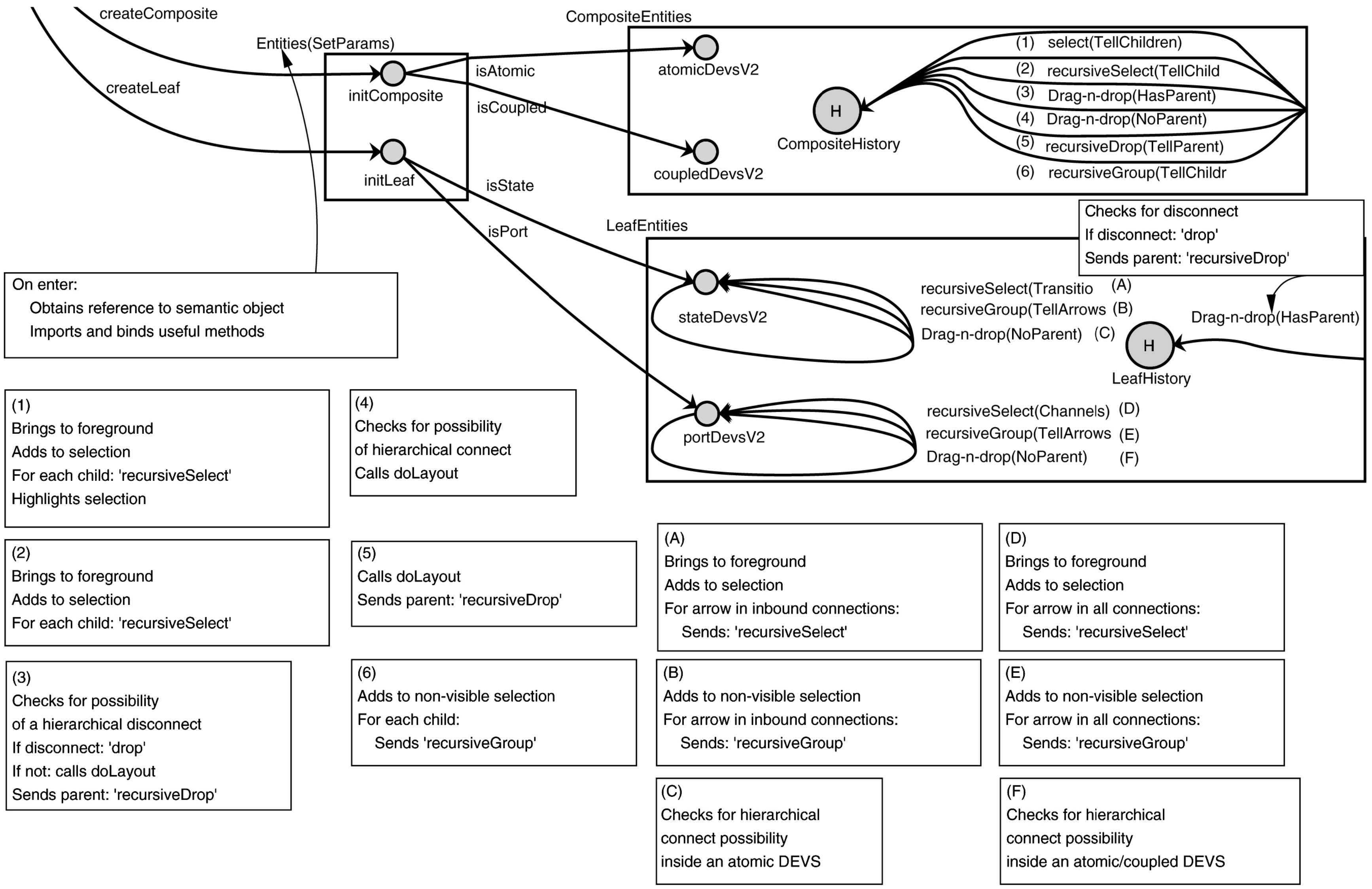

Fig. 5., DCharts model of DEVS reactive behaviour; bottom part for entities shown