Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 145–162

www.elsevier.com/locate/entcs

Distributed Event Graphs: Formalizing Component-based Modelling and Simulation

Juan de Lara[1]

Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
Madrid, Spain

Abstract

In this work an extension to the classical *Event Graphs* formalism for discrete-event simulation is presented. The extensions are oriented towards the specification of component-based models. The abstract syntax has been defined through meta-modelling. Several methodological issues are discussed, concerning the use of two different meta-modelling levels or collapsing the language into a single one, where *"instance-of"* relationships are used between processes and their classes. The operational semantics have been defined through graph transformation. This formal definition enables analysis before code is generated from the model. The syntax and semantics of the visual language have been implemented in the *multi-paradigm* tool AToM^3, together with a code generator that produces stand-alone applications able to run the analysed models in real-time.

Keywords: Meta-Modelling, Graph Transformation, Modelling and Simulation, Component Frameworks, Event Graphs.

1 Introduction

Traditionally, simulation has been classified as continuous, discrete or hybrid. In discrete-event modelling and simulation [13] there is a finite number of events in a finite time interval. There are several ways (called *"world views"*) to describe discrete-event systems. Whereas in the *process-interaction* view one describes the life-cycle (the sequence of activities) of the model entities, in the *event-scheduling* view events are the basic elements of the model. In

[1] Email: Juan.Lara@ii.uam.es

doi:10.1016/j.entcs.2004.08.052

the latter approach, event classes are defined with the effects of the event on the system state and in the future (as new events can be scheduled). One of the *event-scheduling* modelling languages is event graphs [11].

Event graph models are graph-like, where nodes represent events. These specify the actions (changes in the system state) that are executed when the event occurs. Events are related through transitions, which represent the scheduling of the target event when the source event occurs. Transitions may specify an amount of time and a condition for the target event to be scheduled. Although well-known in the simulation community, this formalism is not suitable for object-oriented and component-based simulation, where the system state is partitioned in components, which implement their own behaviour and interact via ports. Component-based modelling solves the problem of scaling, as models become simpler by their partition, one can have many instances of the defined components and these are more adequate for distribution and parallelization. In the present work, an extension to event graphs is proposed in order to consider the communication of processes via events sent through ports. We call the new formalism *distributed event graphs* (DEGs).

In this work we use meta-modelling for the definition of DEGs, whereas the operational semantics are given by means of graph transformation. In DEGs models, the specification level, where classes of processes and behaviours are defined, can be distinguished from the executable instance level, where networks of process instances are built. Two meta-modelling alternatives – separate meta-levels versus single meta-level – are discussed in order to define such levels. The formal definition of syntax and semantics enables analysis of DEGs models using theoretical results of graph transformation [9].

We have used the meta-modelling tool AToM^3 [7] for the implementation of these ideas. AToM^3 was built in collaboration with Hans Vangheluwe from McGill University in Montreal. The tool allows describing the syntax of Visual Languages by means of meta-modelling, and define and execute graph transformation rules. From these high-level descriptions, customized modelling environments are automatically generated. We have created a modelling environment for DEGs and extended it with a code generator that produces stand alone applications. In this way, applications are first visually modelled and analyzed in AToM^3, and then code can be generated from them.

The rest of the paper is organized as follows: section 2 introduces meta-modelling and graph transformation for the definition of Visual Languages; section 3 defines DEGs syntax by means of meta-modelling; section 4 deals with the definition of its operational semantics; section 5 presents an example, implemented in the AToM^3 tool, in which we generate code from the DEG model after its validation through simulation; section 6 discusses related

research and finally, section 7 ends with the conclusions and future work.

2 Meta-Modelling and Graph Transformation for the Definition of Visual Languages

Visual languages have been traditionally described either using meta-models or graph grammars [9]. Meta-modelling allows the definition of the structure of admissible models by defining a model of their (usually abstract) syntax. This model is called a meta-model. When the meta-model is equipped with additional information – for example, regarding visualization (concrete syntax) and additional constraints (for example in the form of logic constraints) – tools can automatically generate modelling environments for the described visual language [7] [10]. Thus, in a meta-modelling approach, one has several meta-levels. In each level, models are instances of some model at a higher meta-level. Moreover, in a *strict meta-modelling* approach [1], each model element is an instance of another element in the corresponding model of the definition language, at a higher meta-level. For example, in the definition of the UML family of diagrams [12] four meta-levels were defined. In the third meta-level (M3), one finds models (that is, the meta-models) of different formalisms (such as DEGs). In the second meta-level there are instances (models) of the different M3 meta-models. In the M4 level we can put the descriptions (meta-metamodels) of the formalisms (that we call meta-formalisms) we used to describe the M3 formalisms. For example, here we can put the descriptions of the core UML, or the meta-object facility (MOF). Finally, at the M1 level, we have execution data.

Graph grammars [9] can also be used to describe a visual language. They are made of rules, each one of them having graphs in their left and right hand sides (LHS and RHS). In order to apply a rule to a graph (called *host graph*) a morphism has to be found between the LHS of the rule and a part of the host graph. If such a morphism is found, the elements in the host graph can be substituted by the elements in the RHS. Rules may also have negative application conditions (NAC), which are patterns that should not be found in the host graph for the rule to be applicable. In the algebraic approach [9], rules are described as pushouts in the **Graph** category. There are two main approaches to describe rules: the Double Pushout (DPO) and the Single Pushout (SPO). In the DPO approach the morphism between the LHS and the host graph must satisfy the *dangling* and the *identification* conditions. The *dangling condition* specifies that if an edge is not deleted its source and target nodes should be preserved. The *identification condition* specifies that if two nodes or edges in the LHS are mapped onto a single node or edge in the

host graph (via a non-injective morphism), then both should be preserved. In the SPO approach, a production is represented as a single (partial) mapping from LHS to RHS. If applied, dangling edges and nodes or edges in conflict due to the identification condition are deleted. Thus, in the SPO approach rules may have *secondary effects*. This is the approach we follow in this article.

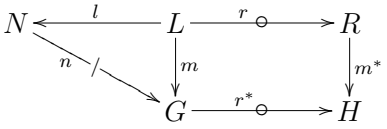

Fig. 1. Application of a Rule to a Graph G in the SPO Approach.

In the graph grammars approach [3] for the definition of Visual Languages, these are defined by either a generating or a parsing grammar. The first kind of grammars are able to generate all the valid models of the language. The second kind of grammars can reduce the valid models of the language into an initial symbol.

In [2] some initial work regarding the combination of both approaches was done. The idea is to use in graph transformation rules the information in the meta-model inheritance hierarchy. In this way, if an abstract node appears in the LHS of a rule, then the rule (called *abstract rule*) is equivalent to all rules resulting from the valid substitutions of the node by the concrete nodes in its inheritance clan (called *concrete rules*). This technique greatly simplifies the rules and is used in the present work in order to specify consistency checking rules for the language (defined by a meta-model).

3 Meta-Modelling Distributed Event Graphs

In order to describe DEGs, we first need to describe process networks. These have two levels. In the first one (specification level), classes of processes are described. In this specification, we include port types (input or output), specify the allowed connectivity between these ports and the kind of events they can produce and receive. Later we will define behaviours for process classes by means of DEGs. In the second level (instance level), we specify processes (instances of some process classes), their pins (instances of ports) and their connectivity. An example is shown in Figure 2, where two process classes and three instances are declared. All ports produce and receive events of type *"arrival"*. For simplicity, we assume that the connectivity for all pins is "0..*", but we could include this information in the ports, at the specification level.

There are two options in order to define this two-level language. The first one is to put each level in separate meta-levels [1]. The second one is

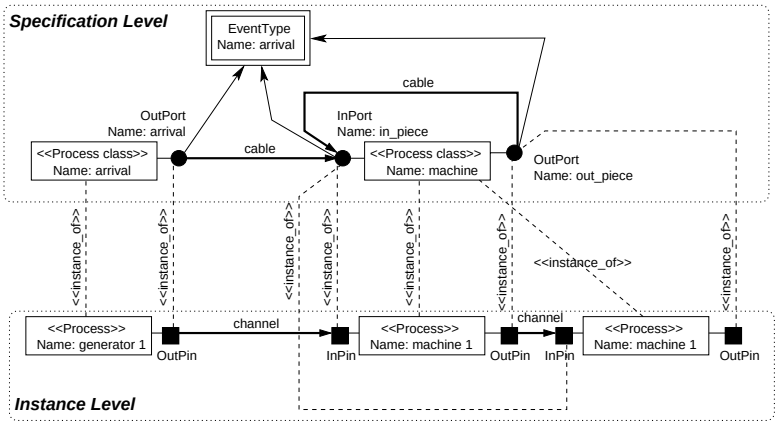

Fig. 2. An Example with the Main Elements of Process Nets.

to put both levels in the same meta-level and explicitly relate elements in both levels by means of *"instance-of"* relationships. Here we use the second option, although in the AToM^3 tool [7], both approaches are possible. The second approach is more flexible, as it allows one to modify the specification level at run-time (possibly using graph transformation rules). Additionally, as behaviour is defined at the specification level using DEGs, it can be executed using graph transformation rules.

In both approaches, one must ensure consistency in models at each level (*intra-level consistency*) and between the specification and the instance levels (*inter-level consistency*). In the latter case for example, we have to ensure that for a certain process at the instance level, all its pins are instances of the appropriate ports at the specification level. Additionally, we have to check that the pin connections at the instance level are permitted at the specification level. Whereas with two separate meta-levels, consistency between both levels is guaranteed by construction, with one meta-level, consistency has to be ensured by using textual (in the form of OCL for example) or graphical constraints (in the form of graph transformation rules) that are evaluated while the user builds the model. For intra-level consistency at the specification level, we have to check that input ports cannot receive connections from output ports that generate events that the input port cannot handle.

Figure 3 shows the meta-model for process networks. Process classes may have a number of behaviours, but only one is active at a certain moment. Processes change the behaviour they execute when they receive a special event (called *"INVOKE"*) with the name of the new behaviour. An event queue stores the generated events during the simulation execution. Events in the queue are ordered by execution time. The current and the final time are kept by a unique entity of type "GlobalTime". A simulator for DEGs consumes

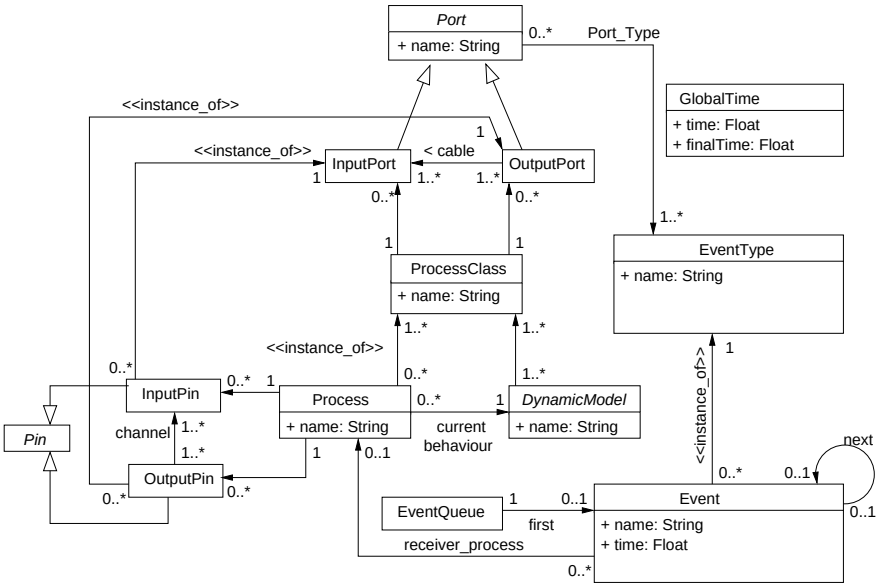

Fig. 3. Meta-Model for Process Nets

events in the queue and creates new events according to the behaviour specification. This is a standard procedure in discrete-event simulation. Events also have a pointer to the process instance that receives the event.

Figure 4 shows the consistency rules mentioned before. The first three rules are inter-level consistency rules and check if pins are correctly instantiated (regarding type and number) and connected. All the rules produce an error if the consistency check fails (that is, if the rule can be applied). The first two rules are *abstract rules*, as we are interested in checking pins and ports, without considering whether they are input or output. For example, the first rule checks whether pins are correct instances of ports. As stated in the previous section, this abstract rule is equivalent to two *concrete rules*, resulting from the valid substitutions of abstract classes *pin* and *port* in their inheritance clan [2]. The second rule checks that a process has at least one pin for each port. Again, for simplicity we allow several pins for each port, although restrictions regarding minimum and maximum values could be set in the specification level. Finally the third rule checks that pins are correctly connected. The fourth rule is an intra-level consistency rule, that checks if in the specification level there is some input port receiving a connection which may produce non-allowed events. The graph transformation rules can be executed by the user at any time during the modelling phase, and they stop their execution as soon as one of the rules can be applied (that is, when a consistency error is found). Note how in AToM^3, this checking could also be done by means of

textual constraints. These are pre- and post- conditions that allow or deny the execution of user events (create, edit, connect, etc.)

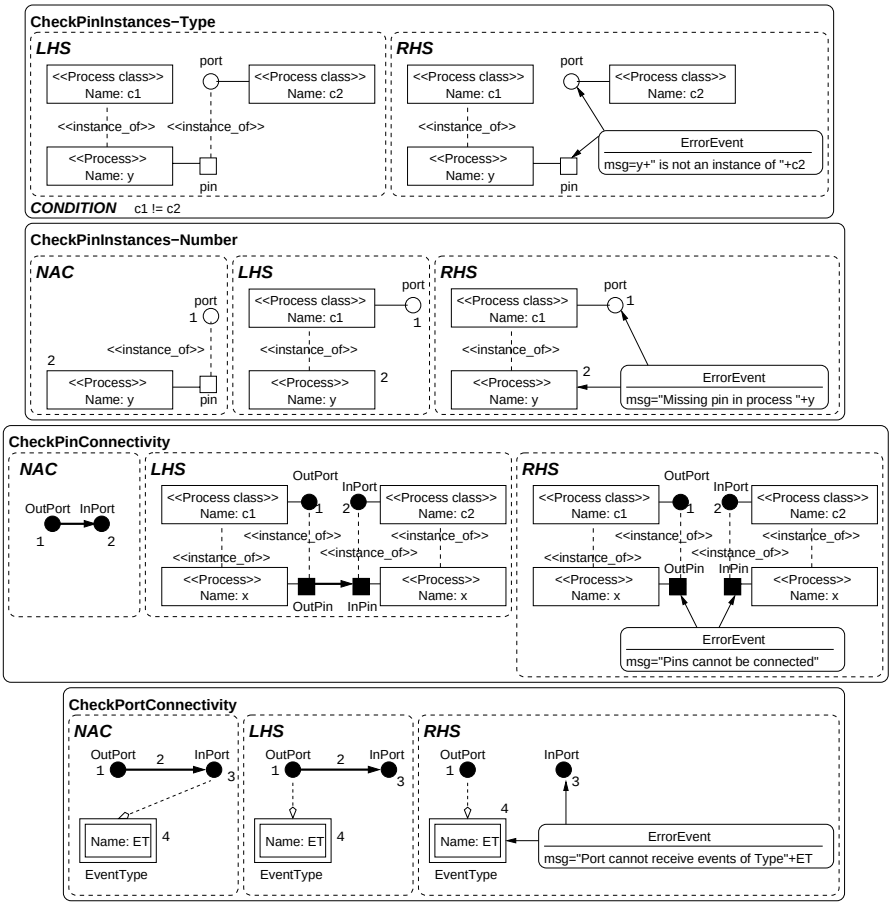

Fig. 4. Some Consistency Rules

In principle, the language for the specification of behaviours is left open (even we could have components specified with different languages), but in Figure 6 we define DEGs for this purpose. The main elements of a regular event graph are shown in Figure 5. Events are represented as nodes in the graph, which depict between brackets the state change (usually variable assignments) that should occur when the event takes place. Events are related through transitions, which can have a time expression and a condition. This means that when the event source of the transition occurs, if the condition is met, the target event is scheduled after the specified time.

As in regular Event Graphs, DEGs are made of events (called *DEGEvent-Type* in the meta-model) in which actions can be specified. In AToM^3 ac-

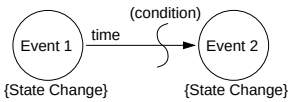

Fig. 5. Main Elements of an Event Graph

tions are specified as Python code and can access variables that can be public (shared between all processes) or private. Transitions are similar to the ones of event graphs, but they may also specify a port. In this case the target event is called external and is sent through the port. Otherwise it is internally generated to the own process. A consistency rule must ensure that the port specified in the transition is either *None* or a valid port. Another rule should verify that in each behaviour there is at most one event of type *initial*. All the initial events of the current behaviour of all the processes are scheduled at time zero. When a process changes its behaviour, its initial event is also scheduled at the current time. It is possible for several outgoing transitions from an event to meet their conditions. In that case, all the target events are scheduled. As next section shows, during simulation an auxiliary entity of type "ExecutionPointer" will be created. This element points to the event that is consumed (by means of relationship "event_to_process") and to all events that are scheduled (by means of relationship "next_event"). A precise specification of the simulator behaviour is given in next section by means of graph transformation rules.

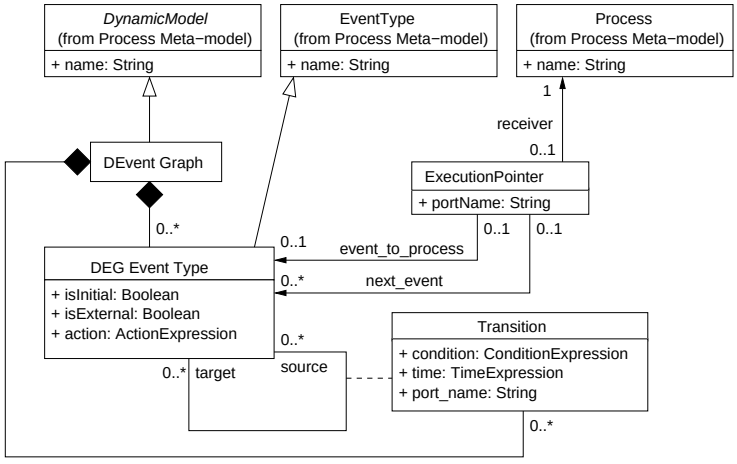

Fig. 6. Meta-Model for Distributed Event Graphs

Figure 7 shows an example model built using the AToM^3 tool, once a concrete syntax is given to the elements in the meta-model. The upper part of the model shows the specification part, where "Arrival" and "Machine" process classes have been defined. Machines have two ports, the input port

(*"In_Piece"*) can receive "Arrival" events either from arrival processes or from machines. The DEG of the arrival process specifies that arrival events are to be generated at fixed time intervals of 10 through port Arrival. The DEG of the machine specifies that each machine has two local variables: *idle* (that signals whether the machine is idle or not) and *queue* which stores the number of pieces waiting to be processed. Both variables are initialised in the initial event (*Init*) of the behaviour. On the arrival of a piece the queue is increased and if the machine is idle, it schedules a *Start_Proc* event to occur at the current time. When the *Start_Proc* occurs, the state is changed to busy and the queue is decreased. After 10 time steps an *End_Proc* is scheduled. When this event occurs, an arrival event is generated immediately through the *Out_Piece* port and if the queue is not zero, a *Start_Proc* is immediately scheduled. Several instances of these process classes are defined below, and are related to them through relationships "instance-of", which are depicted as dotted arrows. A global event queue, shared by all the processes is shown at the bottom of the picture. This queue always has at least two events (*Bottom* and *Top*) which mark the beginning and the end of the simulation and are kept in order to make the specification of the simulator easier. Machine named "machine 2" has no connection in its output pin, so the arrival events generated by the process are lost.

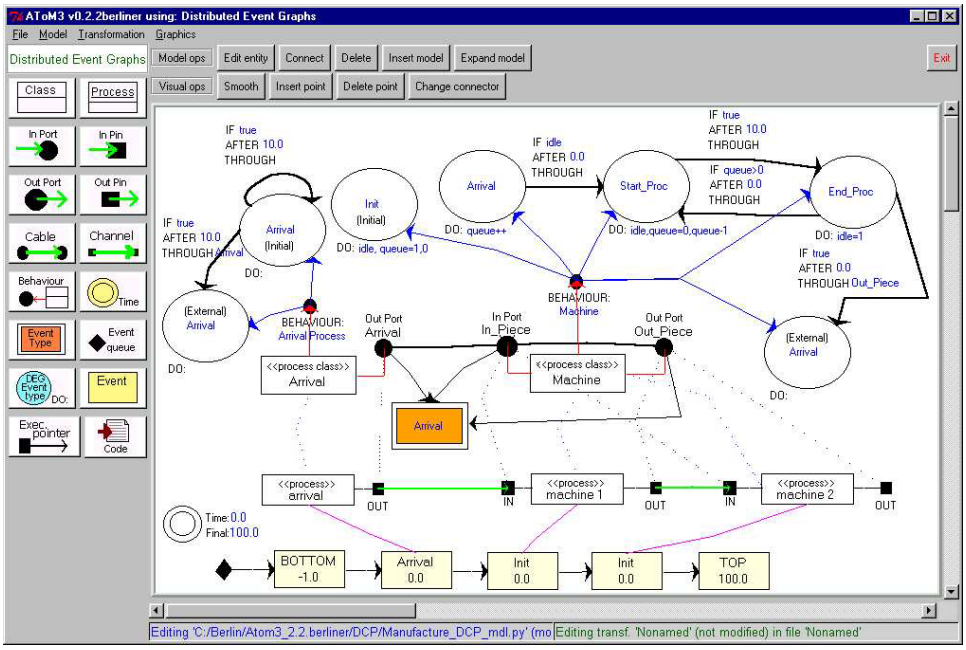

Fig. 7. An Example Model

4 Simulating Distributed Event Graphs

In this section, we model a simulator for DEGs using graph transformation rules. There are two steps in the simulation, which are modelled in separate graph transformation. The first one is the initialization, where the initial events of the current behaviour of each process are scheduled at time 0. The model shown in Figure 7 is the result of the application of this initialization grammar.

The second transformation is the main simulation loop and some of its rules are shown in Figure 8. The first rule consumes the first event in the queue (the one after the "bottom" event) and advances the current time. The rule also creates an "execution pointer" that marks the process which receives the event and the event specification in the DEG describing the current behaviour. When the rule is applied, the action specified (using Python) in the DEG event is performed. The rule is not applicable if there is already an execution pointer. As stated in the previous section, the action can reference shared (global) or local variables. The name of the former variables are preceded by *%glob%* and a single variable is created for all processes in the model. A hand-coded parser (called by function *parse*) executes the state actions. A similar rule to this one was created to remove events from the queue for which there is no *DEGEventType* in the DEG specification.

The second rule searches all the outgoing transitions departing from the event that was last executed. If the transition condition is true, then a new event is scheduled and placed in the event queue. The transition specifies the pin from which the event should be generated. The newly scheduled event points to the process receiving it. In case there are several processes connected to the output pin of the process producing the event, the rule selects one randomly. This is a design decision when defining the language. Other choice could have been to generate one event for each connected process. A similar rule to this one was defined in order to discard events sent through unconnected pins.

Rule 3 is similar to the previous one, but is executed when no port is specified in the transition. In this case, the event is directed to the process that generated it. This is a notation convenience, as one could have a process with one of its output pins connected to one of its input pins and use rule 2 for internal event generation.

Rule 4 handles the event for changing the behaviour of a process. The rule schedules (at the current time) the initial event of the behaviour. A similar rule was defined for the case in which the behaviour does not have an initial event. Finally rule 5 deletes the execution pointer. The rule makes use of

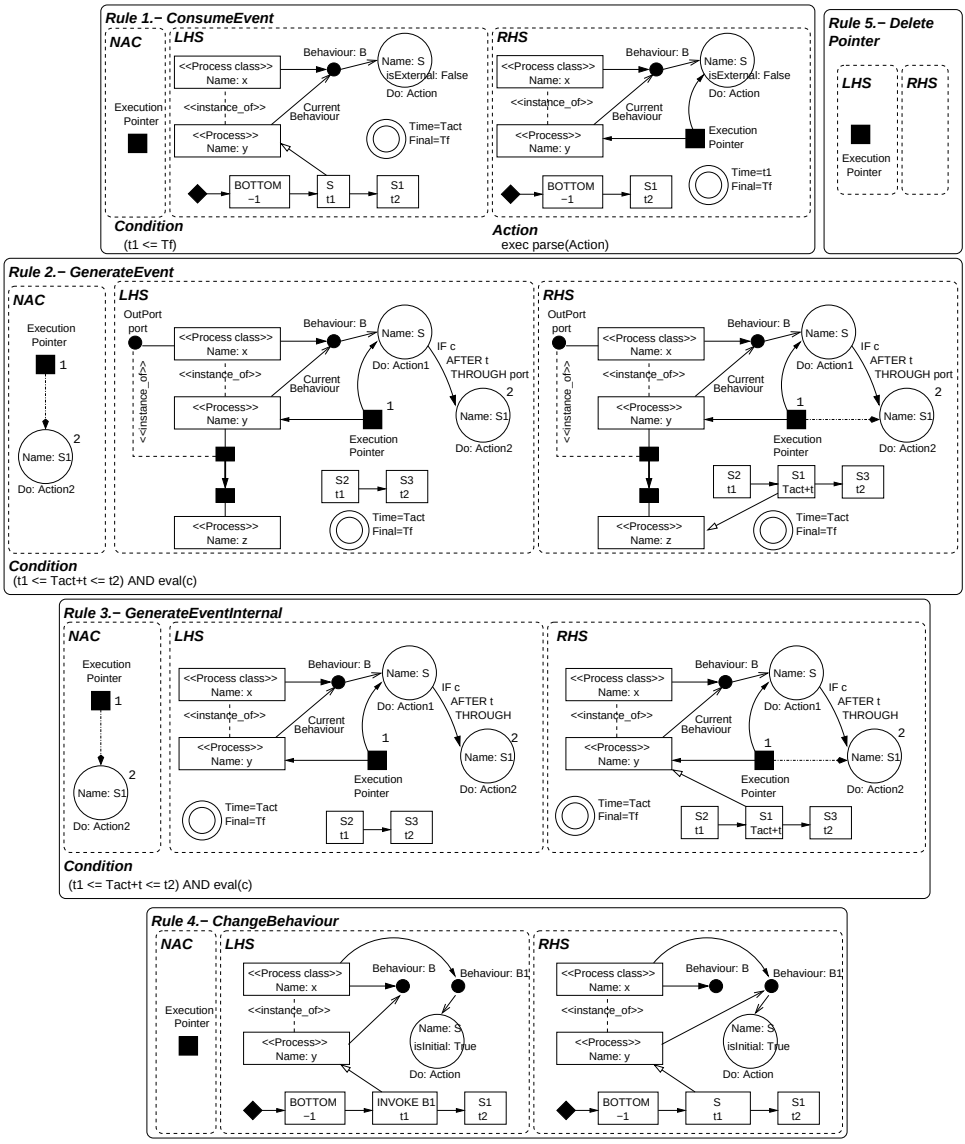

Fig. 8. Rules for the Simulation of DEGs

the property of SPO rewriting (regarding dangling edges) that deletes all the incoming and outgoing edges of the pointer.

Figure 9 shows some steps in the execution of the model in Figure 7. In the first step, the arrival event was consumed and the execution pointer was created. In the second step an arrival event was generated, and finally, in the third step the execution pointer was deleted. The simulation continues by

processing the first *init* event in the queue.

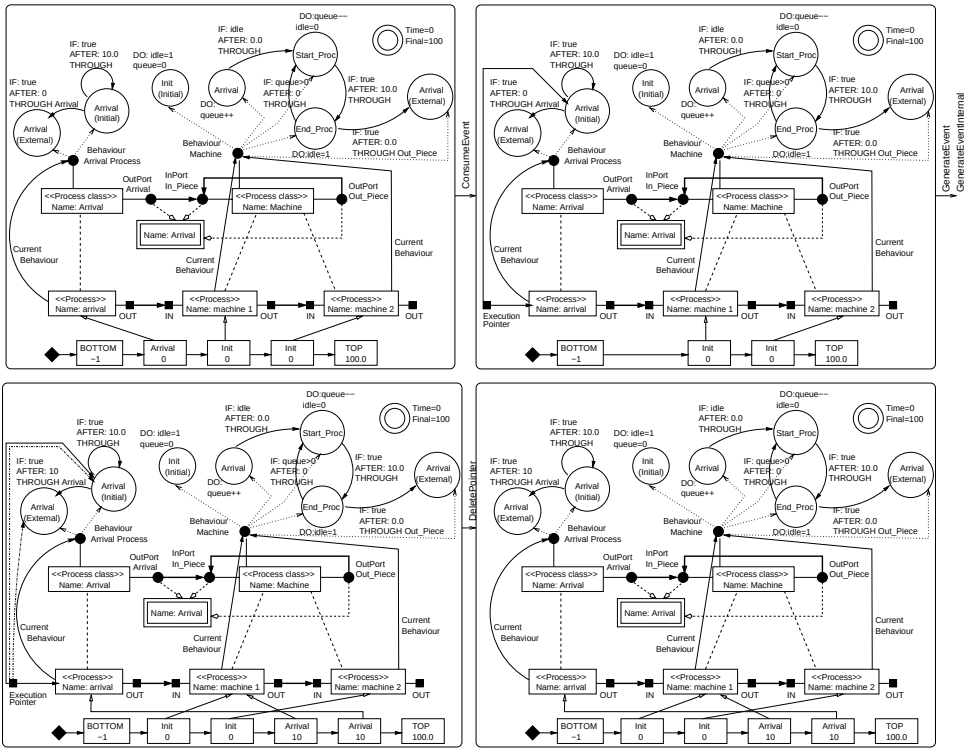

Fig. 9. Some Steps in the Execution of a Model

5 An Example in AToM^3: Code Generation

This section shows a further example of the usefulness of the newly defined formalism and its implementation in the AToM^3 tool [7]. As stated before, AToM^3 allows the definition of visual languages by means of meta-modelling and their manipulation by means of graph transformation rules. The graph rewriting engine can be configured to work in the DPO or SPO approaches and is able to apply parallel rules [8] (in the sense of amalgamation with sharing). Figure 10 shows AToM^3 in the process of editing rule "ConsumeEvent". Mappings from LHS to RHS are given by labelling the graph elements with numbers. In the RHS, we can include Python code to specify attribute values, or copy them from preimage elements in LHS.

In AToM^3, we can take advantage of the interpreted nature of Python, the implementation language of the tool. In the generated DEGs modelling environment (like the one in Figure 7), we can include Python code in the

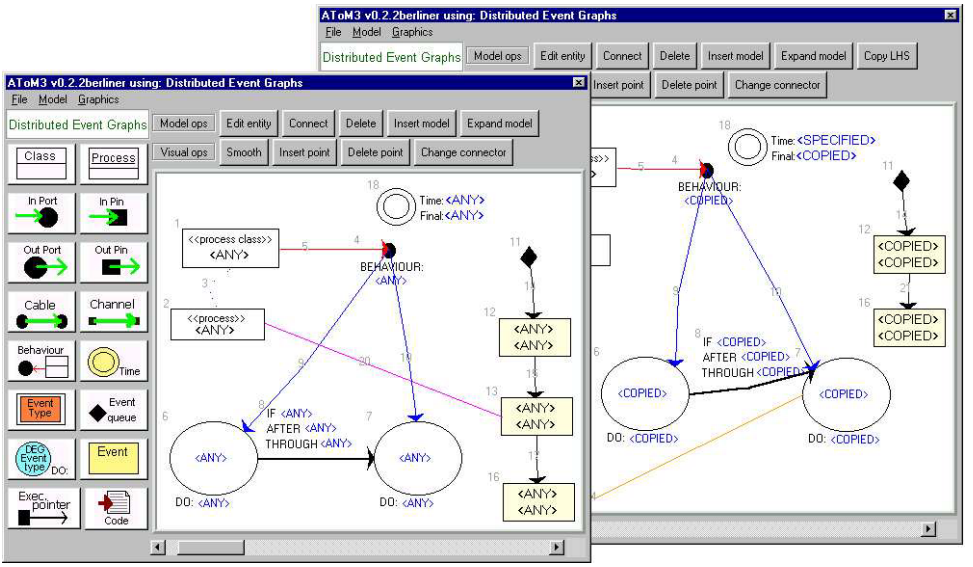

Fig. 10. Editing a Graph Grammar Rule in AToM^3.

DEGEventType actions that are executed when the event occurs. In this way, the DEGs formalism can be used to visually model applications in the style of (textual) event-driven progamming environments such as Visual Basic. This permits, for example, building (by hand) a user interface for each component in the initial event and to modify it in other events. In this way, the user interface of the model is driven by the simulator, inside AToM^3.

Alternatively, we have built a code generator that produces Python code from the DEG models. In this way, models can be run outside AToM^3 and integrated with further code, to form a full application. We use some hand-coded base classes for processes and for the DEGs simulator. Note how the formal (and visual) definition of the simulator as graph transformation rules served as an executable specification for the simulator written in Python. The generated classes inherit from these base classes. AToM^3 creates a Python class for each *process class*. This class has structures to store the different ports (and the connected processes), the behaviours and the events that the component can handle. For each (non-external) event class, a method is created in the generated class. The method is invoked when an event of the corresponding type is consumed by the component. Another Python class is generated for the model. This class creates and connects the process instances and runs the simulation. There are two ways to run the simulation. In the first one, the simulation is run "as fast as possible", in such a way that timing in events do not represent real time. In the second one, the simulation is run in "real time", in such a way that the timing of events is used to drive the execution. That

is, if after executing an event, the next one is in 10 seconds, the execution is suspended for 10 seconds. It is easy to modify the produced code in order to make the user actions in the interface of the generated application produce events that are included in the event queue, directed to the appropriate component (by invoking the appropriate event method in each process).

In this way, one can follow the next sequence of steps in order to generate an application and verify its correctness. The first step is to model the application components in AToM^3 as shown for example in Figure 7. In this step one still does not include information about the user interface in the states actions ("DO" attribute). Once the model is finished, in the second step, we can simulate the model, which corresponds to the main application logic. In the future, further analysis techniques will be implemented (see conclusions section). In the third step, one can include Python code in the "DO" event attributes in order to perform additional actions, such as building the user interface. In the fourth step, it is possible to simulate the model (with the user interface) inside AToM^3. Finally, the application can be generated and further code can be added, for example to link user interface events with the model events. The automation of this task is up to future work.

Figure 11 shows an example (only the specification level) in which we have defined two components, a *Cell* which has an attribute named *colour* and four behaviours: idle (does nothing), shift (on receiving an event, changes the colour and forwards the event with a delay), drain (changes colour but does not forward the event) and delay (forwards the event but does not change colour). A cell component is associated with a controller component (which later will be linked to a button in the user interface) that is able to change the cell behaviour. Additionally, the model has an instance level, where we have connected five cells (the last is connected to the first) and their respective controllers.

The resulting application is shown in Figure 12. We have associated a variable colour *canvas* with the cell components. The canvas also shows the name of the component current behaviour. We have bound the mouse click event with the method that generates the *change* event. We have associated a button (labelled as "Change!") with each controller component and bound the mouse click with the method generating the *CLICK* event. In total, the amount of code added by hand (in the state actions in the model, and after code generation) was negligible.

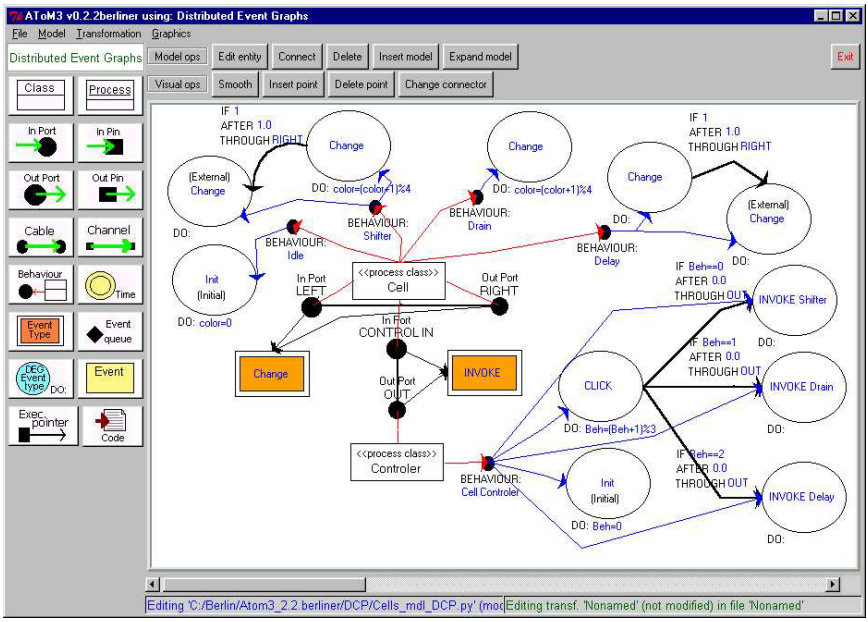

Fig. 11. Modelling the "Shifter Cells" application in AToM^3.

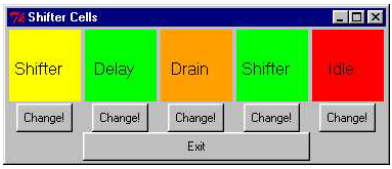

Fig. 12. The generated "Shifter Cells" application.

6 Related work

With respect to the formalism, there have been some approaches for extending event graphs. In [4] two extensions are reported: cancelling edges, and parameter passing. In [5], event graphs are used to describe behaviour of *single* components, but there is no mechanism to express event passing between components via specific ports. To the author knowledge no extension has been proposed to adapt event graphs to component-based simulation.

The defined framework is somewhat similar to DEVS (Discrete Event System Specification) [13]. In DEVS, atomic models are specified by defining transition functions for internal and external events, as well as output functions and a time advance function that sets the amount of time to be spent at each state (if no external event occurs). Atomic DEVS can be coupled via ports to form composite DEVS. In this case there are functions to translate event names from output to input ports. This allows an easier reuse of

components. In our framework, this translation can be done by including two-port components (*"translators"*) that on the arrival of one event produce the appropriate event in the output port. We also allow components to have different behaviours and do not restrict the kind of simulation language used to specify each component (which could be different for each component), if its semantics are based on graph transformation. The use of theoretical results of graph transformation allows the investigation of properties of *multi-formalism* models.

This work has also certain similarities with the concept of components in UML 2.0 [12]. In this new version of UML, components have ports, each one of them can declare required and provided interfaces, which specify the kind of messages the component can send and receive.

With respect to the techniques for the definition of the formalism, graph transformation has been widely used for the definition of operational semantics of formalisms. We can find two main approaches. In the first one, both structure and behaviour are specified with some visual language and graph grammars are used to "interpret" such behaviour. The present work is an example of this approach. In the second approach, structure is described with a visual language as before, but behaviour is directly implemented by means of graph rewriting rules. That is, there is nothing in the model that tells us something about the behaviour: all the information is in the rules. Examples of this approach can be found for example in [6], where process nets with ports are represented with a visual language and their behaviour using context-free grammars.

7 Conclusions

In this work we have extended classical event graphs for their use in component-based models. The definition of the language has been done formally by means of meta-modelling and graph transformation. The language has a specification level – where process classes, ports and behaviour are defined – and an instance level, where the different classes are instantiated. Rules are defined in order to check the intra- and inter-level consistency.

The combination of a formal definition of a language (by means of meta-modelling and graph transformation) and code generation allows the analysis of the model before the application is generated. In our case, we have only implemented a simulator with graph transformation, but further model properties could be investigated using the theoretical results of graph transformation. These include the analysis of parallelism, deadlock, non-determinism, functional behaviour, etc.

In the future, we plan to extend the present work to model distributed discrete-event simulation. For this purpose private event queues are necessary, as well as models (in the form of rules) of protocols. Further planned extensions include for example, the possibility to test the port from which an event reached a component, sending an event through several ports and the definition of hierarchies of events. The latter possibility allows including *abstract* events in the specification in order to make it more compact. Finally, other extensions of the framework to make it more suitable for agent-based simulation are also under consideration. For this application domain, it is needed a way to change the model structure at run-time. This includes creating and deleting new components, and changing their connections.

Acknowledgement

I'd like to thank the three anonymous referees for their comments, and the sponsors of this work: the Spanish Ministry of Science and Technology (TIC2002-01948) and the Santander Central Hispano Bank.

References

[1] Atkinson, C., Kühne, T. 2002. *Rearchitecting the UML infrastructure.* ACM Transactions on Modeling and Computer Simulation, Vol 12(4), pp.: 290-321.

[2] Bardohl, R., Ehrig, H., de Lara J., and Taentzer, G. 2004. *Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation.* In proceedings of ETAPS/FASE'04, LNCS 2984, pp.: 214-228.

[3] Bardohl, R. 2002. *A Visual Environment for Visual Languages.* Science of Computer Programming 44,pp.: 181-203. See also the GENGED home page: http://tfs.cs.tu-berlin.de/~genged/.

[4] Buss, A. 2001. *Basic Event Graph Modeling* Simulation News Europe, Issue 31, Technical Note. pp.: 1-6.

[5] Buss, A. 2000. *Component-Based Simulation Modeling* Proc. of the 2000 Winter Simulation Conference, pp.: 964-971.

[6] Degano, P., Montanari, U. 1987. *A Model for Distributed Systems Based on Graph Rewriting* Journal of the ACM, 34(2), April, pp.: 411-449.

[7] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In Proc. FASE'02, Springer LNCS 2306, pp. 174 - 188. See also the AToM^3 home page, http://atom3.cs.mcgill.ca

[8] de Lara, J., Ermel, C., Taentzer, G., Ehrig, K. 2004. *Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets.* In Graph Transformation-Visual Modeling Techniques (GT-VMT04), Barcelona.

[9] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation.* (1). World Scientific.

[10] Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. *Composing Domain-Specific Design Environments*. IEEE Computer, Nov. 2001, pp.: 44-51. See also the GME home page: http://www.isis.vanderbilt.edu/Projects/gme/default.html.

[11] Schruben, L. W. 1983. *Simulation modeling with event graphs*. Communications of the ACM, 26:957-963.

[12] UML2.0 infrastructure and superstructure specification at the OMG's home page: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

[13] Zeigler, B. P., Praehofer, H., Kim, T. G. 2000. *Theory of Modeling and Simulation 2nd Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.