Modelling and Analysis of Distributed Simulation Protocols with Distributed Graph Transformation

Juan de Lara
Escuela Politecnica Superior
Universidad Autónoma de Madrid (Spain)
jdelara@uam.es

Gabriele Taentzer
Computer Science Department
Technische Universitat Berlin (Germany)
gabi@cs.tu-berlin.de

Abstract

This paper presents our approach to model distributed discrete event simulation systems in the framework of distributed graph transformation. We use distributed typed attributed graph transformation to describe a conservative simulation protocol. We use local control flows for rule execution in each process, as the use of a global control would imply a completely synchronized evolution of all processes. These are specified by a Statechart *in which transitions are labelled with rule executions. States are encoded as process attributes, in such a way that rules are only applicable if the process is in a particular state. For the analysis, we introduce a* flattening *construction as a functor from distributed to normal graphs. Global consistency conditions can be defined for normal graphs which specify safety properties for the protocol. Once the flattening construction is applied to each rule, the global conditions can then be translated into pre-conditions for the protocol rules, which ensure that the protocol fulfils the global constraints in any possible execution. Finally, the paper also discusses tool support using the AToM^3 environment.*

Keywords: Distributed Graph Transformation, Distributed Simulation, Protocols, Discrete Event Simulation.

1 Introduction

Traditionally, simulation has been classified as continuous, discrete or hybrid. In discrete-event simulation there is a finite number of events in each finite time interval. There are several ways to describe discrete-event systems. Here, we concentrate in the *event-scheduling* view, where events are the basic elements of the model. In this 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* [15], which we extended and formalized in [12] and use in this work.

Some discrete event systems have such a complexity that techniques for speeding up their simulation are essential. One of these techniques consists on partitioning the simulation model, in such a way that different parts are executed in parallel in different processors [5]. Processes are usually not independent, but they need certain events produced by others in order to properly perform the computation. In distributed simulation, protocols synchronize the evolution of each process, governing how they handle their local time and preventing causality errors when processing events coming from other processes.

In our approach, system dynamics are expressed as graph transformation. The algebraic approach to graph transformation has a rich body of theoretical results, developed in the last 30 years (see [2]). Transformations expressed as graph grammars become formal, declarative, high-level and graphical models, subject themselves to analysis. Distributed graph transformation [16] (DGT) was developed with the aim to naturally express computations in systems made of interacting parts. In this way, a distributed graph has two levels: a network and a local level. Network nodes are assigned local graphs, which represent their state.

In this work we show that DGT is a suitable framework for the modelling (i.e. design) and analysis of distributed discrete event simulation systems, by modelling a conservative protocol [5]. To specify each process behaviour, any discrete event simulation language can be used if its operational semantics are described by means of graph transformation. In particular, we use an extension to event graphs for this purpose. The framework of DGT allows describing both the protocol and the specification language semantics in a uniform way, facilitating analysis. Other new results that we show include a new characterization and extension of distributed graph transformation, to include type graphs and attributes at the network level, a flattening functor to go from distributed graphs to normal graphs and the definition of local control flows (with Statecharts) for network nodes. The presented examples have been implemented in

the AToM^3 tool [11] by flattening the distributed graphs and explicitly modelling the hierarchy between network and local graphs.

2 Distributed Event Graphs

In this section we briefly present event graphs [15]. In this formalism, events are depicted as nodes in a graph. These have a specification of the state change in the present (as variable assignments, specified between keys) and events to be scheduled in the future. The latter are depicted as arrows between the occurring event and each scheduled event. Arrows may have a time specification and a condition. If the latter is true the target event is scheduled after the specified amount of time. Figure 1 shows the main elements of an event graph. We have extended event graphs for component-based systems by allowing a port specification in transitions [12]. In this way, the target event is sent through the given port.

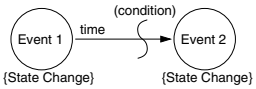

Figure 1. Main Elements of an Event Graph.

A simulator for event graphs makes use of an event queue, where the scheduled events are stored, ordered by simulation time. Initial events are scheduled at time zero. A simulator takes the first event, executes the specification of the state change and advances the time to the time of the event. Then it schedules new events according to the specification, which are again stored in the queue. The process continues until a final time is reached, or a final event is processed.

3 Distributed Simulation

In distributed simulation, the system is divided in a number of *logical processes* (LPs) [5], each one of them executing a part of the simulation. The simulation is carried out by the interaction of the LPs, which send time stamped events to each other, although LPs can also produce internal events. In distributed simulation, there must be a means to synchronize the LPs, as each have a local clock (called *local virtual time*, LVT). There are two kinds of algorithms to handle event synchronization: *conservative* and *optimistic*. In the former algorithms, a causality error due to a LVT higher than the time stamp of an incoming event can never happen. In optimistic protocols, the situation may happen, and then a part of the simulation performed by the LP must be undone (rollback) [6]. In this work, we concentrate in *asynchronous conservative* protocols, also known as Chandy-Misra-Bryant (CMB) [1] [6]. Listing 1 shows a typical pseudocode for a CMB protocol (adapted from [5]).

```
[1]  LocalVirtualTime.time = 0
     // Simulation time for component
[2]  EventQueue.first = Initial Event
     // Event queue is initially empty
[3]  for all i in InputPort: i.clock = 0
     // Set clocks for each port to 0
[4]  ExecutionPointer.STEP()
     // Do one step, inserting internal events in queue
[5]  while LocalVirtualTime.timeHorizon < LocalVirtualTime.finalTime:
     // loop until simulation final time
[6]     for all i in InputPort: await not_empty(i)
        // wait for an event in the input port
[7]     for all i in InputPort: i.clock=max_timeStamp(i)
        // i.clock is the bigger timeStamp of any event in i
[8]     LocalVirtualTime.timeHorizon=min(i.clock for i in InputPort)
        // The process time horizon is the smaller clock of all ports
[9]     min_channel_id = i such that its clock is the smallest
[10]    if (Event.Queue.first.scheduledTime <= LocalVirtualTime.timeHorizon)
[11]       or (InputPort[min_channel_id].first.scheduledTime <=
[12]          LocalVirtualTime.timeHorizon):
[13]       if (Event.Queue.first.scheduledTime <
[14]          InputPort[min_channel_id].first.scheduledTime):
[15]             event = removeFirst(Event.Queue)
[16]       else:
[17]             event = removeFirst(InputPort[min_channel_id])
[18]       LocalVirtualTime.time = event.sheduledTime
[19]       if isExternal(event):
[20]          put(event@LocalVirtualTime.time+lookahead) in
[21]               OutpuPort[event.port]
[22]       else:
[23]          ExecutionPointer.STEP() // execute event
[24]     for all o in OutpuPort:
[25]      if is_empty(o) put(null@LocalVirtualTime.time+lookahead)
[26]                          in OutpuPort[o]
[27]    for all o in OutpuPort:
[28]       send(o.contents)
```

Listing 1: Pseudocode for a Conservative Protocol.

In conservative protocols, LPs have a *time horizon*, which is the maximum simulation time it is safe to reach. Beyond this point causality errors may occur with incoming events. The time horizon should be iteratively increased during simulation by the particular protocol being used. The *lookahead* is the simulation time below which no external event will be generated. It is sent as a timestamp with each event. Thus, as a difference from other protocols, we assign events two time specifications: the *timestamp* and the *scheduled time*. The *timestamp* is the lookahead of the LP when the event was generated. The *scheduled time* stores the simulation time at which the event should be executed.

In order to avoid deadlock, in each simulation loop, if a process does not send events through an output port, it sends a *null event*. These are not taken into account for the simulation, but are used by each LP to calculate its time horizon. This is inefficient and should be avoided whenever possible. Nonetheless, null events prevent the possibility of deadlock for some situations, (but do not work for all possible situations). The protocol does not indicate how to calculate the lookahead, this depends on each particular model. In our case, it is the scheduling time of the first event in the queue.

All these protocol details about time handling should be kept transparent to the language used to describe LP behaviour. One of our goals is to obtain a way to automatically "port" a simulation language for its use in a distributed environment, with any distributed protocol and vice versa. In this way, one could have a multi-formalism system where LPs are specified with different formalisms.

4 Graph Transformation

Graph transformation [2] is a formal means to manipulate graphs based on rules. In analogy to Chomsky grammars on strings, graph rules are made of left and right hand sides (LHS and RHS), both containing graphs. Intuitively, when applying a rule to a graph, a *match* morphism should be found between the LHS of the rule and the graph. If such morphism is found, then the rule can be applied. Then, the matched part in the graph can be substituted by the RHS of the rule. This process is called a derivation step. An example is shown in Figure 2, where a rule modelling a process sending an event through an output port is applied to a graph G yielding graph H. In this example, events in processes are stored in a queue, shown as a black diamond. The second event in the queue is sent by the rule.

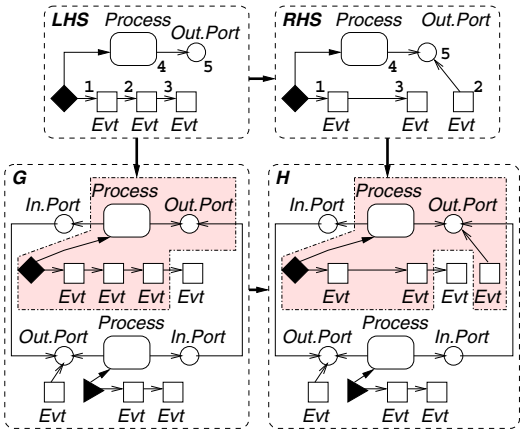

Figure 2. A Rule Application.

A *graph transformation rule* $r : L \rightarrow R$ consists of a pair of graphs L, R such that the union $L \cup R$ is defined. Thus, $L \cup R$ forms a graph again, i.e. the union is compatible with source and target. The LHS L represents the pre-conditions of the rule, while the RHS R describes the post-conditions. $L \cap R$ defines a graph part which has to exist in order to apply the rule, but which is not changed. $L \setminus (L \cap R)$ defines the part which shall be deleted, and $R \setminus (L \cap R)$ defines the part to be created.

A *graph transformation step* is defined by first finding a match m of the LHS L in the current object graph G such that m is structure-preserving (see the colored, dotted region in Figure 2). If a vertex embedded into the context, shall be deleted, dangling edges can occur. These are edges which would not have a source or target vertex after rule application. There are two ways to handle this problem: either the rule is not applied at match m, or it is applied and all dangling edges are also deleted.

The applicability of a rule can be further restricted, if additional application conditions have to be satisfied. A special kind of application conditions are *negative application conditions* (NACs) which are pre-conditions prohibiting certain graph parts.

Performing a graph transformation step with rule r at match m, all the vertices and edges which are matched by $L \setminus (L \cap R)$ are removed from G. The removed part is not a graph in general, but the remaining structure $D := G \setminus m(L \setminus (L \cap R))$ still has to be a legal graph, i.e., no edges should left dangling. This means if dangling edges occur during a rule application, they have to be deleted in addition. Moreover, the so-called *identification* condition prohibits the rule application if an element in L is identified (by means a non-injective match) into a single element in G if one of the elements is deleted and the other is preserved. In the second step of the transformation, graph D is glued with $R \setminus (L \cap R)$ to obtain the derived graph H. Since L and R can overlap in a common graph, its match occurs in the original graph G and is not deleted in the first step, i.e. it also occurs in the intermediate graph D. For gluing newly created vertices and edges into D, graph $L \cap R$ is used. It defines the gluing items at which R is inserted into D.

One of the ways of formalizing rules and rule application is based on category theory [2] and is called Double Pushout (DPO). In this approach, rules are represented as three graph components and two injective span morphisms as follows: $L \xleftarrow{l} K \xrightarrow{r} R$. K is called the interface graph and contains the preserved elements by the rule application. $L - K$ and $R - K$ are the elements deleted and added by the rule application respectively. A rule application is thus modelled by two pushouts in the **Graph** category.

Thus, graph transformation can be used in several ways: to specify a formalism operational semantics, for model transformation and model optimization. As expressed in the introduction, the advantage of using graph transformation for model manipulation is that it is a graphical, natural and formal way to express computations in graphs.

5 Distributed Graph Transformation

While the basics of graph transformation are regular graphs, in DGT [16] *distributed graphs* are transformed by means of *distributed rules*. A distributed graph has two levels of abstraction. The *network graph* has nodes that contain graphs (called *local graphs*). Edges of distributed graphs represent total graph morphisms between the local graphs (see Figure 3). For the present definition, both network and local graphs are attributed and typed with respect to a type graph. In our case, we use network nodes to represent LPs and ports, and local graphs depict each LP and port states.

For representing network graphs, we use the category of attributed typed graphs. As in [3], we first use the notion of

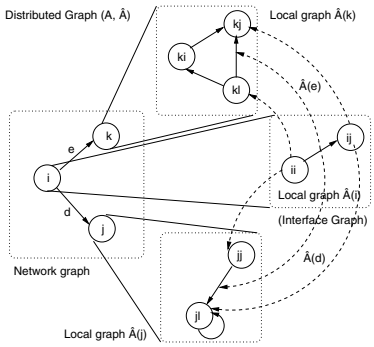

Figure 3. A Distributed Graph.

E-graph [1]. These are extended graphs with two sets of vertices (representing graph and data – attributes – nodes) and three sets of edges (connecting graph nodes, graph nodes and attributes and graph edges and attributes). We call the graph formed with graph nodes and graph edges the "raw graph" of the E-graph. E-graphs, together with E-graph morphisms form the category **EGraphs**.

We can define attribute graphs by providing E-graphs with an algebra over a data signature (that we call $BASIC$), in such a way that the union of a subset of carrier sets of the algebra is the set of data nodes of the E-graph. Attributed graphs, together with attributed graph morphisms form the category **AGraphs**. A *type graph* can be defined as an attribute graph where the algebra is final. Figure 4 shows an type graph for the definition of process network models.

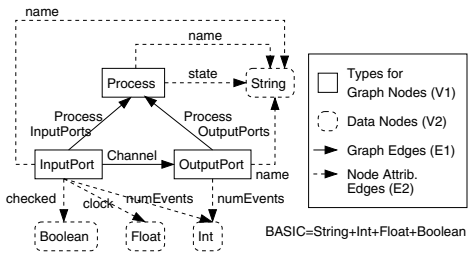

Figure 4. An Attributed Type Graph.

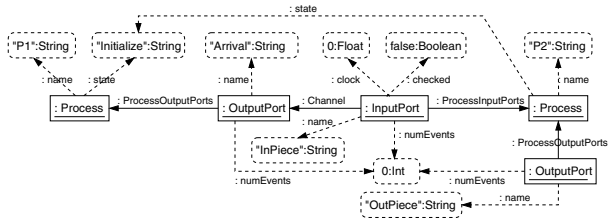

Figure 5. An Attributed Typed Graph.

[1]For space limitations, we keep the definitions informal

Attributed typed graphs can then be defined as a co-slice category $\mathbf{TG} \uparrow \mathbf{AGraph}$ (denoted as $\mathbf{AGraph_{TG}}$), where TG is an attributed type graph. Objects in this category are of the form (AG, t), where AG is an attributed graph and $t\colon AG \to TG$ is an attributed graph morphism called the typing of AG. Figure 5 shows an attributed typed graph (AG, t) with respect to the type graph in Figure 4. Nodes and edges are labelled with their identity (sometimes omitted), followed by their type (in UML notation).

We define the category of distributed graphs $\mathbf{Distr}(\mathbf{AGraph})$ with objects of the form $\hat{A} = (A, \tilde{A})$, where A is an attributed graph (the network graph), belonging to category $\mathbf{AGraph}$ and $\tilde{A}\colon A_{RAW} \to \mathbf{AGraph}$ is a functor from the small category induced by the *raw graph* to category $\mathbf{AGraph}$. We provide distributed attribute graphs with a typing by defining a co-slice category. Thus, the category of typed attributed graphs over a distributed type graph TG is the co-slice category $\mathbf{TG} \uparrow \mathbf{Dist}(\mathbf{AGraph})$ (denoted by $\mathbf{Dist}(\mathbf{AGraph})_{\mathbf{TG}}$). This category has as objects all pairs (t_X, X), where $X \in \mathbf{Dist}(\mathbf{AGraph})$ and $t_X : X \to TG$ is a **Dist(AGraph)**-morphism.

Distributed typed attributed graphs are transformed via distributed typed attributed rules. These consist of a network rule and a set of local rules, one for each node in the network rule left hand side (LHS) [16]. Thus, a distributed typed attributed rule $\hat{p} = (\hat{L} \xleftarrow{\hat{l}} \hat{K} \xrightarrow{\hat{r}} \hat{R})$ consists of two injective $\mathbf{Dist}(\mathbf{AGr})_{\mathbf{TG}}$-morphisms $\hat{l}$ and $\hat{r}$ such that $\forall i \in K_{V_1}$ (i.e. the set of graph nodes of K), the span $\hat{p}_i = (L_{l(i)} \xleftarrow{l_i} K_i \xrightarrow{r_i} R_{r(i)})$ is a typed attributed rule.

In addition to the usual *dangling* and *identification* conditions, two additional conditions should be verified: the *connection* and the *network* conditions [16]. The former condition says that if a rule deletes or adds elements in source or target local graphs, the local mapping should be changed as well. A rule satisfies the *network* condition, if whenever a network node is deleted, its local graph is deleted as well.

In addition, we equip distributed rules with certain conditions that prohibit the application of the distributed rule when the condition is met by the host graph [16]. Given a production $\hat{p}$ defined as before, an attributed graphical constraint has the form $cc_{\hat{L}} = (\hat{c} : \hat{L} \to \hat{A}, c_{\hat{A}})$ over $\hat{L}$, where $\hat{c}$ is an injective attributed typed distributed morphism and $c_{\hat{A}} = \{\hat{h}_i : \hat{A} \to \hat{A}_i\}_{i \in K_{V_1}}$ is a K-indexed set of injective attributed typed distributed morphisms $\hat{h}_i$, such that $\forall x \in L_{V_1}$ the typed attributed graph morphisms c_x are constraints over $\tilde{L}(x)$, and $\forall i \in K_{V_1}$ and $y \in A_{V_1}$, the typed attributed graph morphisms h_{i_y} are constraints over $\tilde{A}(y)$. Notice that if $c_{\hat{A}}$ is empty, we have a negative application condition (NAC), thus a match from $\hat{A}$ should not be found in the host graph for the rule to be applicable.

Finally, we can define a distributed typed attributed

graph grammar, with conditional distributed typed attributed rules (with respect to a distributed graph TG) as a tuple $DGG_{TG} = (SG, \{(\hat{p}^i, A^i_{\hat{L}})\})$, where SG is a distributed typed attributed graph over TG (the start graph) and $\{(\hat{p}^i, A^i_{\hat{L}})\}$ is a set of typed distributed attributed rules (with application conditions) with respect to TG. The semantics of the grammar (i.e., its transition system) is the set of all possible graph transformation sequences resulting from the repeated application of the rules in the grammar.

A flattening functor can be defined, which puts together in a single attributed graph the network and local graphs, adding edges (that we call "hierarchy edges") from each node in the local graph to its network node, and from domain and co-domain nodes of morphisms induced by the network edges. Again, we skip the formal definition of the construction for space limitations. The flattening of typed distributed attributed graph production is made by flattening the kernel, left and right hand sides.

6 Distributed Simulation as Distributed Graph Transformation

In this section we model distributed systems as distributed graphs, where network nodes are LPs and ports, and local graphs depict their states. Figure 4 showed a part of the type graph, corresponding to the network level. The type graph for network graphs contains processes, corresponding to LPs, with attributes *name* and *state*. The latter is used to define local control flows for rule execution. In this way, rules are applicable only if the LP is in a certain state. LPs may be connected to input and output ports, which may contain events. Input ports have an attribute indicating the number of events they contain, and a clock with the maximum time stamp of the included events. Input ports are connected to output ports via channels, which represent morphisms from the elements (events) of the input ports to the elements of the output ports.

Figure 6 shows the type graph for local graphs in nodes of type *Process*. As shown before, each LP is provided with a LVT, which contains the current and final simulation time, the time horizon and the lookahead. If an LP does not have input ports, its time horizon is made equal to the final simulation time. Each event that is sent through the ports is timestamped with the LP lookahead. This information will be used by other LPs receiving these events to adjust their time horizon. In addition, each LP has an event queue and a pointer (*SentEvents* node) to each sent event. An edge is kept from the *SentEvents* node to each sent event until the event is erased (see protocol rules). Additionally, events have attributes to store its type, its scheduling time, its time stamp (the LP lookahead when they were generated), the port they have to be sent through (*None* if it is internal) and a flag (*checked*) used by the protocol. For transparency, the time stamp of the event is not set by the simulation language, but by the protocol rules when the event is sent.

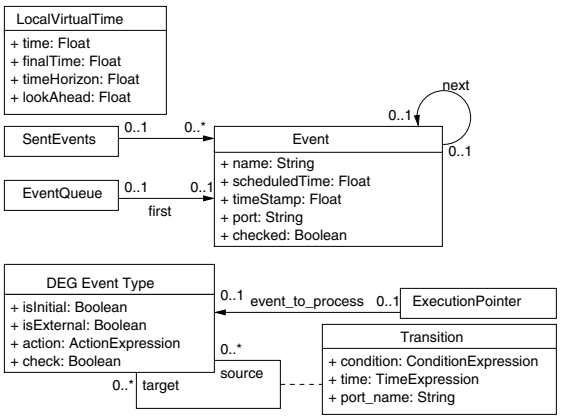

Figure 6. Type Graph for Local Graphs in LPs.

The type graph for local graphs in port nodes contains just node "event". Events in input ports are used to compute the time horizon of the receiver LP, as the maximum time of every event in every input port. Here we use one of the properties of conservative protocols: LVT in each LP (and thus, its lookahead) advances monotonically. When an event is used for this computation, it is marked and can be eliminated from the input port.

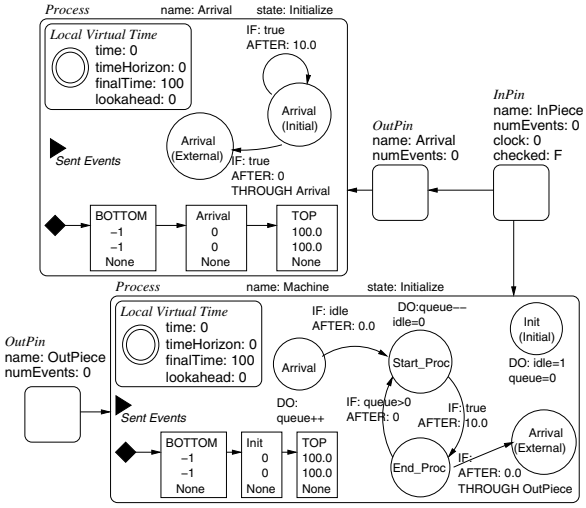

Figure 7. A Distributed Simulation System.

Figure 7 shows an example that uses the previous type graphs. It is made of two LPs, the one on the left is a "generator" that produces "arrival" events. The one on the right is a "machine" connected to the generator. Inside each LP there is a behaviour specification (using event graphs), a LVT, an event queue and a sent event node (a black tri-

angle). The machine has an unconnected output port, thus, arrival events sent through it are lost.

7 LP Behaviour Specification

This section models the operational semantics of event graphs, as we use them to specify LP behaviour.

7.1 Specification of Local Control Flow

In distributed graph transformation if rules are executed with a certain *global* control flow, for example layers, then all LPs are implicitly synchronized. As rules represent LP actions, all LPs execute the same kind of action in a synchronized way. A more realistic model of execution is to provide each LP with *local* control flows. This can be done by using a *statechart* to model the states each LP can be in. The transitions in the statechart are labelled with actions representing either the execution or the failure of rules. The LP state can be implemented as an attribute of network nodes (see Figure 4). Figure 8 shows the main elements of a statechart for control flow specification. We assume the type graph in Figure 4 for network nodes, and we assign the statechart to LPs.

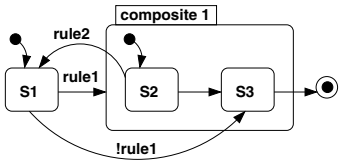

Figure 8. Control Flow Statechart.

We can identify different kinds of transitions. If a transition is labelled with a rule name (*rule1* in the figure), the rule's LHS should check the state the LP is in, and in the RHS the LP state can be changed. For the example, *rule1* should check that the state is *S1* and change it to *composite1*.

A transition may lead to a composite state. This is the case of transition labelled as *rule1*. In this case a rule is automatically generated to change the state from the composite state to its initial state. That is, these rules go down in the statechart hierarchy. In our example a rule should be generated, taking LPs from state *composite1* to *S2*.

A transition may have no label (such as the transition going from *S2* to *S3*). In this case, a rule is automatically generated that changes the LP state from the source to the target state.

Finally, a transition may be labelled with the failure of a rule. In this case, the transition takes place if the rule is not applicable. In the example this is the case with transition labelled as *!rule1*. In this case, it is possible to build a rule named *!rule1* which is applicable in the source state if and only if *rule1* is not. In general, we assume that rules can have application conditions [4] of the form $\{c_i\colon P \to Q\}$, but we are able to build the converse of the rule only if the set c_i is empty. Moreover, this construction can be generalized to calculate the converse of a set of rules. The resulting rule is applicable if none of the rules in the set is applicable. For this purpose, we introduce the *converse construction* on rule sets as follows:

Definition 1 *(Converse of a set of rules) Given a set of local rules with application conditions* $P = \{p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, \{x_i\colon L_i \to P_i, \{c_{ij}\colon P_i \to Q_i\}\})\}$ *for node* n *(that is, we assume node* n *is in the host graph), with each* c_{ij} *empty, we define the converse of* P *as:* $!P = (L \xleftarrow{l} K \xrightarrow{r} R, \bigcup_{p_i \in P}\{x_i'\colon L \to L_i, \{x_i\colon L_i \to P_i\}\})$*, where* $L = K = R$ *and they contain only node* n

Lemma 2 *Given a set of local rules* P *for node* n *(as in the previous definition), if any rule in* P *is applicable, then* $!P$ *is not applicable. Moreover, if none of the rules in* P *is applicable, then* $!P$ *is applicable.*

We use converse rules as labels for transitions. In our example, rule *!rule1* would be applicable if *rule1* is not and change state from *S1* to *S3*.

Additionally, from a given state, several transitions may depart. If several of the rules they are labelled with are applicable, we have a non-deterministic choice. In the example, this is the case of state *S2*, (but not of *S1*, as *rule1* and *!rule1* are mutually exclusive by construction). Altogether, the example defines the following execution trace (given as a regular expression): $(rule1 rule2)^*(rule1 + !rule1)$. Note how, in general, one can obtain a regular expression from a statechart (without parallel components) by first flattening the statecharts and the using the well-known algorithms of automata theory.

Figure 9 shows a statechart depicting the local control flow for a step (an event execution) in a simulator for event graphs (rules are explained in the next subsection). The "Advancing Time" state is the initial one. Here either rule "Advance Time" can be executed, or not. In the latter case the LP goes to state "Step Ends" and the simulation step finishes. We assume that some other rules (the ones implementing the distributed protocol in the next section) bring the LP state to "Advancing Time", and that additional ones change the state from "Step Ends". As a convention, we assume that in order to use a certain simulation language in our framework, after each simulation step, the LP should end in state "Step Ends", even if no event can be consumed in its event queue. One of the reasons for which the rule can be applied is that the time horizon is less than the event time.

Ending the simulation step here gives the LP the opportunity to increase the time horizon (by the protocol rules). At this point, the protocol rules can be executed.

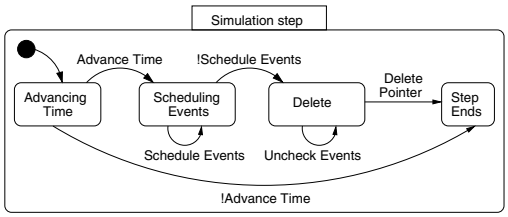

Figure 9. DEG Simulator Control Flow.

Once the first event is consumed and the LP is in state "Scheduling Events", rule "Schedule Events" is executed as long as possible. When it is no longer applicable, then its converse is applicable and changes the state to "Delete". Here either "Uncheck Events" or "Delete Pointer" are executed. In the latter case, the state is changed to "Step Ends".

As stated before, these control flows are specified for distributed graphs. For the implementation of the control flow in the system, an attribute is created in the network node representing the node state. However, for analyzing the system, we *flatten* the distributed graph with the functor presented before. In the resulting normal graph, the control flow does not change. With flattening, network nodes and local graphs are merged in a unique graph, where nodes and edges keep their attributes (including the attribute used to store the state). Therefore, the control flow works in the same way for the flattened graph.

7.2 Event Graphs Semantics

In this section we describe the operational semantics of event graphs by means of rules. In the rule in Figure 10, the first event in the queue is selected (an event labelled *"BOTTOM"* is always kept in the first position to make rules easier), together with its specification in the event graph. The rule is applicable if the LP is in state *"Advancing Time"* (see Figure 9). If the rule is applied, the event is consumed, the simulation time is increased, and a pointer is created signalling the event specification that should be executed. In addition, the event action is performed.

Additional rules (not shown for space limitations) schedule new events, following the transitions of the event that is being executed (pointed by the *Execution Pointer*). The event is placed in the local event queue, even if it is an external event (which should be sent through some port). This is done by simplicity and transparency, as additional rules for the distributed protocol will place the external events in the appropriate output port. Other rules set flag *check* to false and delete the execution pointer. After each simulation step, the LP ends up in state "Step Ends". Finally,

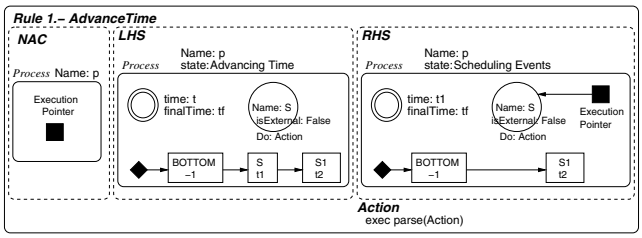

Figure 10. A Rule for the DEGs Simulator.

some other rules must take care of the initialization process. This process is made once, before the simulation execution starts, and schedules the initial event(s) of the event graph in each LP local queue.

As stated before, for the use of this rules in a distributed environment, some NACs will be added later, induced by global safety properties of the given distributed simulation protocol.

8 Modelling a Protocol

In this section, we model a conservative protocol using distributed graph transformation. LP behaviour can be described using the statechart in Figure 11. The actions to occur in states "Simulation Step" and "Initialize Simulation" are implemented by the semantics of the simulation language used in each LP. As stated before, we add a rule to change the process from state "Step Ends" inside "Simulation Step" to state "Consuming Ext Events". The statechart states are indeed embedded in the model as attribute *state* of LPs (see Figure 4).

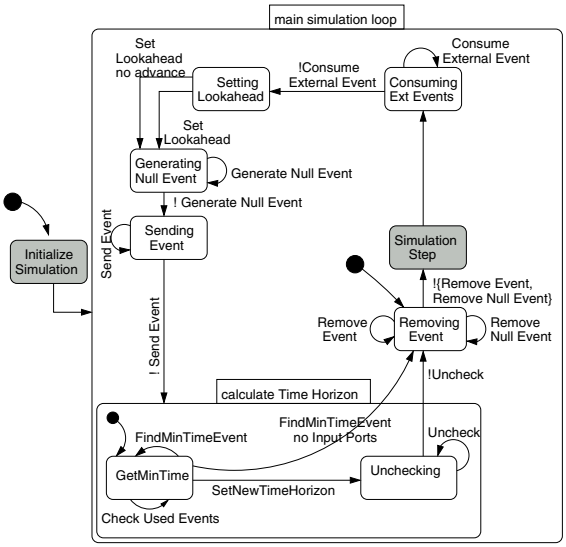

Figure 11. LP General Behaviour.

Again, we just show some rules for space limitations. Figure 12 shows the applicable rule in state "Consuming Ext Events". The rule "Consume External Event" takes one external event from the local queue and places it in the corresponding output port. The event is also connected to the *"Sent Events"* node. The external events are placed into the local queue by the simulator rules. The LP changes its state to "Setting Lookahead" if the converse of this rule is applicable.

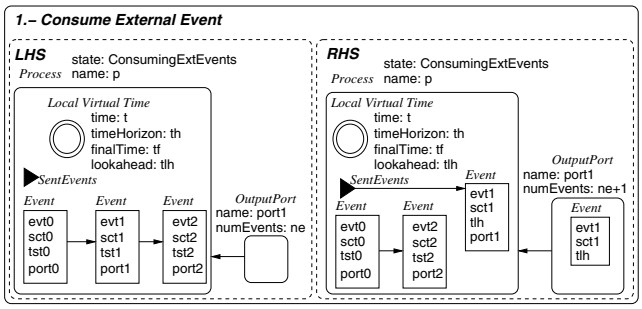

Figure 12. "Consume External Event" Rule.

Rule "Send Event" in Figure 13 sends the events that are already stored in output ports. This rule is applicable if the sending LP is in state *Sending Event* (the receiver can be in any state). When the rule is executed, the event is stored in the input port of the receiver LP as well as in its local queue (properly ordered). The converse of the previous rule changes the LP state to "calculate Time Horizon".

Rules in Figure 14 are used to set the LP time horizon. The first rule (*"FindMinTimeEvent"*) is executed in state "getMinTime" and looks for the event with the maximum time stamp in each input port. This time is assigned to attribute "clock" of the port.

Rule "Check Used Events" marks as "used" all events in each input port. In addition, the port clock is updated if new events arrive after the last rule execution and before this rule was executed. Rule "Set New Time Horizon" sets the LP time horizon as the minimum of clocks in each input port. The first NAC checks that all events inside each port have been considered. The second NAC checks that all ports have been considered. Finally, the third NAC checks that the selected port has the minimum time. If applied, the rule also changes the LP state to "Unchecking".

Two additional rules ("FindMinTimeEvent No Input Ports" and "Uncheck", not shown in the paper) are applicable when the LP has no input ports, and to set the checked attribute of each input port to false, respectively.

9 Consistency Conditions

In this section, we specify global invariants for the system (safety properties). These properties can be translated

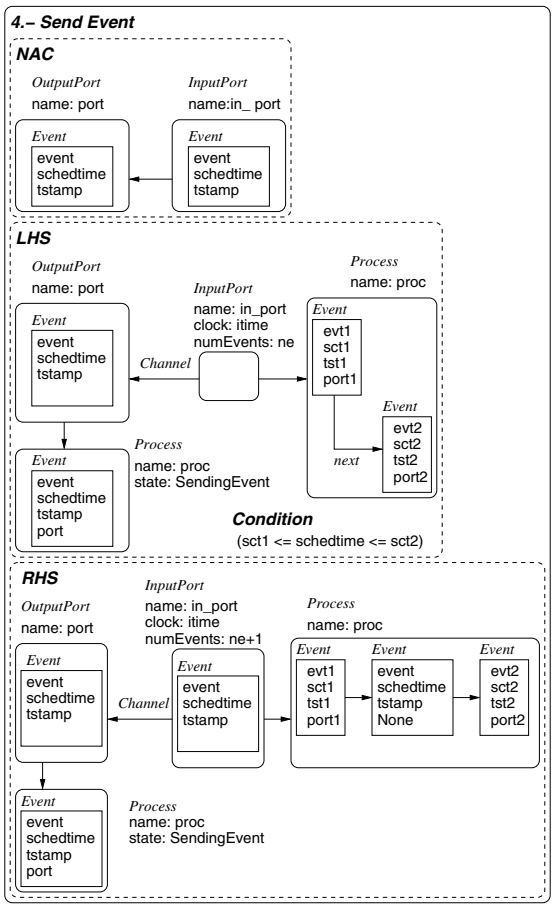

Figure 13. "Send Event" Protocol Rule.

into pre-conditions for the flattened rules. We use this technique for three purposes:

- To detect flaws in already existing rules. In this case, the induced pre-conditions are stronger (more restrictive) than existing pre-conditions in the rule. For the example, we may set constraints detecting causality errors.
- To ensure that a simulator specified with graph transformation rules can work with a distributed protocol.
- To ensure that the protocol rules preserve a safe state in case of the simulation language doing anything incorrect. In our case, one incorrect behaviour is for example sending events in non-monotonic ascending order. In this case, the pre-conditions induced by the safety conditions are a kind of "exception handler".

For space limitations, we just give an example of the second. A complete discussion of the other two kinds of constraints is given in [13].

A global condition labelled as "Time Horizon Overpassing Error", is shown to the left of Figure 15. It detects the

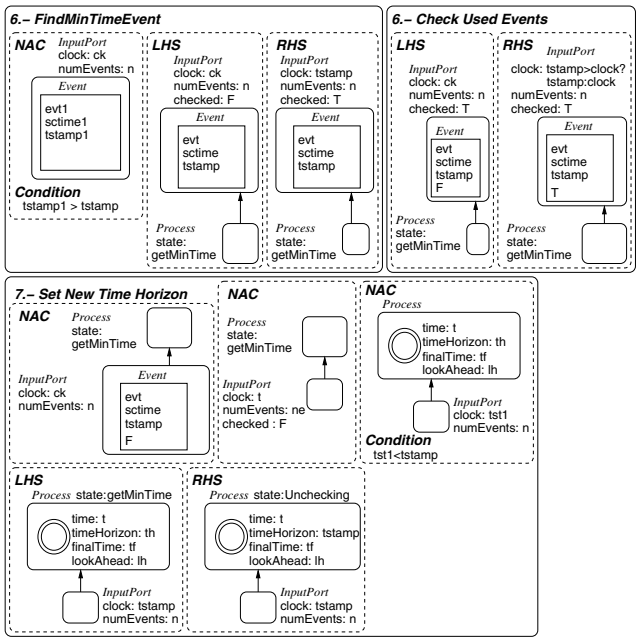

Figure 14. Rules to Update the Time Horizon.

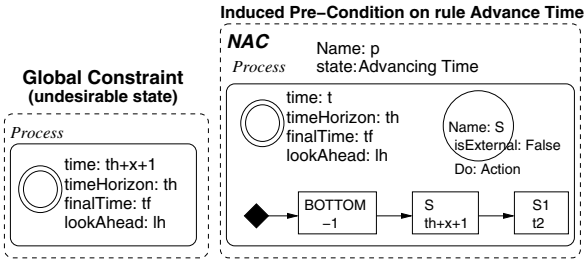

Figure 15. Checking Time Horizon (left) and Induced Pre-Condition on rule "Advance Time"(right).

(non-desirable) situation in which the LVT of a LP becomes larger than its time horizon. The right of Figure 15 shows the NAC induced by this condition on rule "Advance Time" (shown in Figure 10), one of the rules implementing the operational semantics of event graphs, responsible to advance time. The NAC prohibits the application of the rule if the scheduled time of the first event is larger than the time horizon. In general, this consistency condition is very useful in cases where the rules implementing the operational semantics of a certain formalism were not designed for distributed environments, and then did not take into account the time horizon. Thus, it allows the automatic migration of rules into distributed environments. In our case, this is the only safety constraint that we have to consider for the rules implementing the semantics of the visual language. For other distributed simulation protocols additional constraints should be taken into account.

10 Tool Support

We have implemented the described examples in AToM^3 [11], after flattening the graphs. AToM^3 allows the definitionf of visual languages by means of meta-models. For their manipulation, graph transformation rules can be used.

Figure 16 shows an example modelled with AToM^3. The example shows three simple components. A *User* process generates job events, which are sent to a *Buffer* process. After a delay of 2 time units, the *Buffer* sends the event to a *Processor* component. After a delay of 10 time units, the job is done and sent to the user again. As AToM^3 does not support distributed graphs, we used flattened graphs. Note how, comparing with Figure 3, the flattened graphs have *hierarchy edges* from each node inside a local graph to the network graph. In the example, from the local state of processses and ports to each process and port. This results in more complex graphs, although the main idea is the same.

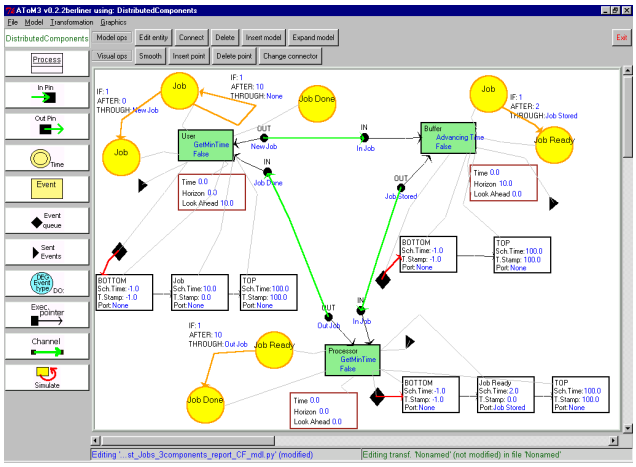

Figure 16. An Example with AToM^3.

11 Related Work

With respect to the specification technique for describing protocols, a similar approach to describe distributed systems (also based on graph transformation) can be found in [10]. Other approaches are based on domain specific languages such as TED [14]. A popular approach for modelling protocols is the use of Coloured Petri Nets [9]. For example, in [7], a part of the TCP protocol was modelled and analyzed. They encoded TCP segments (messages) as

coloured tokens and were able to use Petri nets results to calculate the reachability graph (for certain configurations of processes) and detect possible deadlocks. It could also be possible to calculate the reachability graph of a graph transformation system and to perform reachability analysis (or model checking) on it. Other techniques based on Petri nets use numerical simulation to obtain performance metrics. This kind of simulation is outside the scope of our work, which uses symbolic techniques.

12 Conclusions and Future Work

In this paper, we have explored DGT for the specification and analysis of simulation protocols. We have extended previous definitions of DGT by adding type graphs and using attributes also at the network level. The use of DGT simplifies the models, as one does not have to explicitly include "hierarchy" edges between containers (LPs) and its contained graphs. In the modelling phase, we have also taken advantage of the possibility to specify global safety properties for the system. These are translated into preconditions for the rules implementing the operational semantics. This helps in discovering possible flaws in the designed rules, to port the operational semantics of simulation languages to a distributed environment and to set "exception handlers" for unexpected errors. In addition, we specify by means of statecharts the control flow of actions (rule applications) that each LP can perform. In this way, no global control flow is present, as this leads to a global synchronization of all process actions. We used AToM^3 for the implementation of the visual languages and the protocol rules.

In addition to consistency conditions, for the analysis of the protocol we can also compute critical pairs [8]. This allows us to identify rules that are in conflict: one rule application may disable another one. The technique allows us to investigate the independency of protocol rules (and in particular of those involving interactions of LPs).

For the future, other protocols should be modelled and analyzed as well. Once having a well-designed model, the generation of code for distributed applications from this model is certainly attractive.

Acknowledgements: This work has been sponsored by the SEGRAVIS network and the Spanish Ministry of Science and Technology (TIC2002-01948).

References

[1] Chandy, K., Misra, J. 1979. *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*. IEEE Transactions on Software Engineering, Vol 5. No 5., pp.: 440-452.

[2] Ehrig, H., Kreowski, H.-J., Montanari, U., Rozenberg, G. eds. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1, 2 and 3* World Scientific.

[3] Ehrig, H., Prange, U., Taentzer, G. 2004. *Fundamental Theory for Typed Attributed Graph Transformation* LNCS 3256, pp.: 161-177.

[4] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H. 2004. *Constraints and Application Conditions: From Graphs to High-Level Structures*. LNCS 3256, pp.: 287-303.

[5] Ferscha, A. 1995. *Parallel and Distributed Simulation of Discrete Event Systems.* In Parallel and Distributed Computing Handbook, McGraw Hill, pp.: 1003-1041.

[6] Fujimoto, R. 2000. *Parallel and Distributed Simulation Systems*, John Wiley and Sons, Inc.

[7] Han, B., Billington, J. 2002. *Validating TCP Connection Management.* Workshop of Software Engineering and Formal Methods, Adelaide, Australia. pp: 47 - 55.

[8] Heckel, R., Küster, J. M., Taentzer, G. 2002. *Confluence of Typed Attributed Graph Transformation Systems*. LNCS 2505, pp.: 161-176. Springer.

[9] Jensen, K. 1997. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol.1, Basic Concepts*. Springer-Verlag, Berlin.

[10] Koch, M. 2002. *A graph-based approach to the compositional specification of distributed systems* GETGRATS Closing Workshop, ENTCS 51.

[11] de Lara, J., Vangheluwe, H. 2002 *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* LNCS 2306, pp.:174-188. Springer.

[12] de Lara, J. 2004. *Distributed Event Graphs: Formalizing Component-based Modelling and Simulation.* In Proc. VLFM'2004.

[13] de Lara, J., Taentzer, G. *Modelling and Analysis of Distributed Discrete Event Simulation in the Framework of Distributed Graph Transformation*, to appear as Technical Report of TU Berlin.

[14] Perumalla, K., Fujimoto, R., Ogielski, A. 1998. *TED – A Language for Modeling Telecommunication Networks.* ACM SIGMETRICS Vol.25(4). pp: 4 - 11.

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

[16] Taentzer, G., Fischer, I., Koch, M., Volle, V. 1999. *Distributed Graph Transformation with Application to Visual Design of Distributed Systems*. In [2], Vol.3.