Graph Transformation for Domain-Specific Discrete Event Time Simulation

Juan de Lara[1], Esther Guerra[2], Artur Boronat[3], Reiko Heckel[3], and Paolo Torrini[3]

[1] Universidad Autónoma de Madrid (Spain), `Juan.deLara@uam.es`
[2] Universidad Carlos III de Madrid (Spain), `eguerra@inf.uc3m.es`
[3] University of Leicester (UK), `{aboronat, reiko, pt95}@mcs.le.ac.uk`

Abstract. Graph transformation is being increasingly used to express the semantics of domain specific visual languages since its graphical nature makes rules intuitive. However, many application domains require an explicit handling of time in order to represent accurately the behaviour of the real system and to obtain useful simulation metrics.
Inspired by the vast knowledge and experience accumulated by the discrete event simulation community, we propose a novel way of adding explicit time to graph transformation rules. In particular, we take the event scheduling discrete simulation world view and incorporate to the rules the ability of scheduling the occurrence of other rules in the future. Hence, our work combines standard, efficient techniques for discrete event simulation (based on the handling of a future event set) and the intuitive, visual nature of graph transformation. Moreover, we show how our formalism can be used to give semantics to other timed approaches.

1 Introduction

Graph Transformation [5] (GT) is becoming increasingly popular as a means to express and analyse the behaviour of systems. For example, it has been extensively used to describe the operational semantics of Domain Specific Visual Languages (DSVLs) in areas such as reliable messaging in SOA [7], web services [12], gaming [16] and manufacturing [3]. The success of GT is partly because rules are intuitive and allow the designer to use the concrete syntax of the DSVLs.

When used to specify the DSVL semantics, the rules define a simulator, and their execution accounts for the state change of the system. This is enough for languages with a discrete, *untimed* semantics, where the time elapsed between two state changes is not important. However, for its use in simulation applications or real-time systems, where system behaviour depends on explicit timing (e.g. time-outs in network protocols) and performance metrics are essential, a mechanism is needed to model how time progresses during the GT execution.

Computer simulation [6] is the activity of performing virtual experiments on the computer (instead of in the real world) by representing real systems by means of computational models. Simulation is intrinsically multi-disciplinary, and is at the core of research areas as diverse as real-time systems, ecology, economy and physics. Hence, users of simulations are frequently domain experts (but not necessarily computer scientists) who are hardly proficient in programming languages but on the domain-specific notations used in their scientific domain.

Discrete-event simulation (DES) [2, 6] studies systems where time is modelled in a continuous way ($\mathbb{R}$), but in which there is only a finite number of events or state changes in a finite time interval. Many languages, systems and tools have been proposed over the years in the DES domain [6]. However, these require specialized knowledge that domain experts usually lack, or consist of libraries for programming languages like Java. Therefore simulationists would strongly benefit from a domain-specific, graphical language to describe simulations.

In this paper, we propose such a language by incorporating an explicit handling of time into the GT formalism. In this way, based on the *event scheduling* approach to simulation [14], we allow rules to program the occurrence of other rules in the future. For this purpose, our approach makes use of two concepts: (i) explicit rule invocation (and cancellation) with parameter passing between invocations and (ii) time scheduling of those matches. This improves efficiency in two ways: rule execution is guided by the parameter passing, and the global time is increased to the time of the next occurring event (instead of doing small increments). Our goal is to provide the simplest possible time handling primitive, on top of which other more advanced constructs can be added. We show that *scheduling* is one such primitive mechanism, and demonstrate its use to model (stochastic) delays, timers, durations and periodic activities.
Paper organization. §2 gives an overview to DES and *event scheduling.* §3 introduces the use of (untimed) GT to describe the semantics of DSVLs. Next, §4 extends GT with rule invocations and parameter passing, called *flow grammars.* These are extended with time scheduling in §5. §6 discusses how to model other timed approaches with ours. §7 covers related research and §8 concludes.

2 Discrete Event Simulation: World Views

Discrete-event systems can be modelled using different styles, or *world-views* [2, 6]. Each world-view focuses on a particular aspect of the system: events, activities or processes. An *event* is an instantaneous change in an object's state. An *object activity* is the state of an object during a time interval, between two events. A *process* is a succession of object states defining its simulation life-cycle. Therefore, there are three different approaches to describe discrete time models [2]: *Event Scheduling* (ES) focussing on events, *Activity Scanning* (AS) focussing on activities, and *Process Interaction* (PI) focussing on object processes.

ES languages offer primitives to describe events, their effect on the current state, and the scheduling of future events. Time is managed efficiently by simply advancing the simulation time to the time of the next event to occur. AS languages focus on describing the conditions enabling the start of activities. They are less efficient because, lacking the concept of event to signal state changes, they have to advance the time using a small discrete increment. To increase efficiency, the *three-phase approach* combines ES and AS so that the start of new activities is only checked after handling an event. Finally, PI provides constructs to describe the life-cycle of the active entities of the system.

Among the three approaches, ES is the most primitive, as events delimit the start and end of activities, and a flow of activities makes up a process. Hence, we concentrate on the ES approach, and in particular on the Event Graphs notation [14], an example of which is shown to the left of Fig. 1. The event graph models a simple communication network, where a node sends messages periodically to a receiver node through a channel with limited capacity.

Nodes in the event graph represent events, and there are two special ones: the start and the end of the simulation, identified with a tick and a double circle respectively. The state is represented with variables (*ch* being the load of the channel and w the number of messages waiting). Below event nodes, a sequence of variable expressions describes state changes. Arrows between events represent schedulings. For instance, the arrow from the event *start* to *end* means that, once *start* happens, an occurrence of *end* will happen after t_f time units. If no time is indicated (like in the arrow from *start* to *create*) then the target event is scheduled to occur immediately. Arrows can be decorated with a condition that is evaluated after processing the source event, and that must be *true* in order to schedule the target event at the indicated time. For example, the arrow from *create* to *send* means that after creating a message, this will be sent only if there is some message waiting and the channel has enough capacity. Finally, although not shown in the example, event graphs can also contain *event-cancelling* edges, represented as dashed arrows. These edges indicate the deletion of all events of the target type scheduled after the indicated time units, if the condition (if given) holds at the time the source event is processed [14].

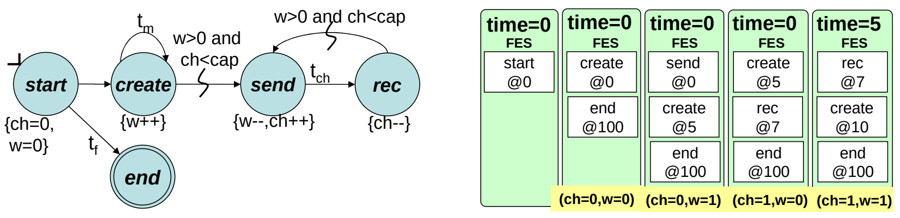


Fig. 1. An event graph model (left). An execution of the model (right).

DES simulators use a future event set (FES) which contains the events scheduled to occur in the future. The simulation proceeds by taking the event with earliest occurrence time and executing its specification as given by the event graph (i.e. modifying the system state and scheduling new events). Many algorithms and data structures exist to handle the FES efficiently [6].

The right of Fig. 1 shows some execution steps of the model, using as parameters $t_m = 5$, $t_f = 100$, $cap = 1$, $t_{ch} = 7$. Each state transition consumes the earliest event in the set, updates the current time to the time of this event, modifies the variables, and schedules new events according to the model. The simulation continues until processing the *end* event.

The execution of event graphs is efficient, but its modelling sometimes lacks intuitivity. This is so because event graphs are not domain-specific and force the use of scattered variables for expressing state changes instead of full-fledged models and DSVLs. Next we show how GT provides such intuitive formalism, but lacks time handling capabilities, which we subsequently add in §5.

3 Rule-Based, Domain-Specific, Untimed Simulation

In this section we give an overview of the use of GT to describe the semantics of DSVLs. The syntax of DSVLs is usually defined through a meta-model, or type graph, which contains the node and edge types that models can use. For example, Fig. 2 shows on the left an example meta-model describing a DSVL in the domain of communication networks and protocols. In this language, a network is made of nodes which exchange messages through channels. Messages can be either requests or replies. There are two special kinds of nodes: initiators, whose attribute *isInit* is *true*, and terminal, whose attribute *isFinal* is *true*.

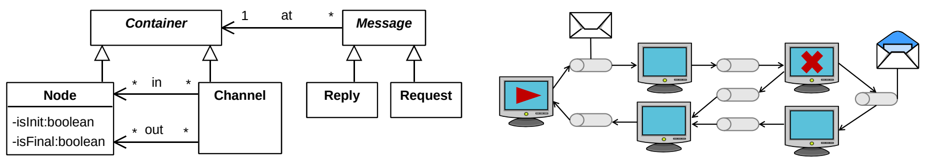


Fig. 2. The DSVL meta-model (left). A model (right).

The right of Fig. 2 shows an example model in concrete syntax, with an initiator node to the left (marked with a "play" icon), and a terminal node marked with a cross to the right. Requests are shown as closed envelopes (like the one to the left) and replies as open envelopes (like the one to the right). Channels are depicted as pipes.

We are using this DSVL to describe the dynamics of a simple protocol where the messages are propagated through the network at random. When a request reaches a terminal node, this node sends back a reply which traverses the network randomly until it reaches the initiator. Since channels can lose messages, the initiator sends a new request periodically. We also model changes in the topology, so that nodes are connected and disconnected from channels.

We define this protocol with DPO GT rules [5], in which rules have the form $p : \langle L \xleftarrow{l} K \xrightarrow{r} R, NAC = \{n_i\colon L \to N_i\}_{i \in I}\rangle$. NAC is a set of Negative Application Conditions. In this paper, we sometimes depict rules using just their LHS and RHS, and use the concrete syntax of the DSVL.

A morphism $m\colon L \to G$ is a valid match for rule p in the host graph G, written $m \models_G p$, if m satisfies the gluing conditions [5], and if $\forall n_i\colon L \to N_i \in NAC$ $\nexists n\colon N_i \to G$ with $n \circ n_i = m$. If $m \models_G p$, we can perform a direct derivation.

Fig. 3 shows some GT rules of the simulator. Messages are generated from initiators by rule *init*. Nodes can *send* and *receive* messages. As in [5], objects in rules can be matched to objects in the host graph with a more concrete type. For example, rule *send* contains an object `m` of type message (depicted as a dotted envelope) which can be matched to both requests and replies. The rule does not apply if the message is a request and the node is terminal (first NAC), or if the message is a reply and the node is initial (second NAC). The first case is handled by rule *reply*, which processes the request and generates a reply, whereas the second case is handled by rule *end*, which removes the reply from the net. Rules *createConnection* and *deleteConnection* model the creation and deletion of connections from nodes to channels. The former only applies to nodes without output channels. Finally, rule *lose* simulates the loss of messages.

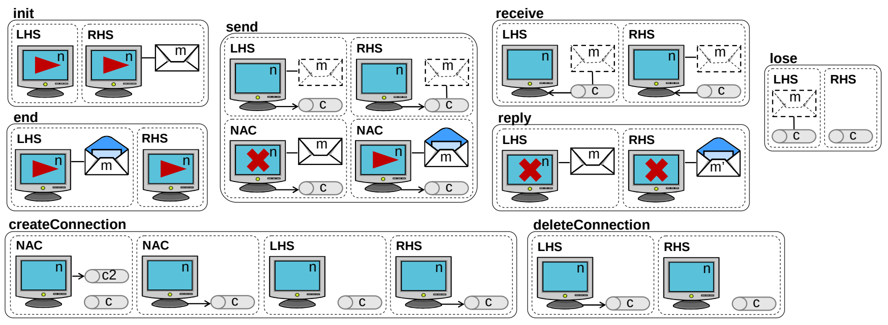


Fig. 3. Some rules of the DSVL simulator.

For the purpose of simulation, the standard approach to GT has two drawbacks. First, even though these rules capture the untimed semantics of the language, they cannot represent time-outs, delays or be used to obtain metrics about the system performance. For example, we would like to set a time-out in the initiator so that it sends requests each 50 time units, and also to model transmission delays and the average rate at which channels lose messages.

Second, rules represent events which signal the start or end of activities of the entities in the system. Thus, the focus on active entities requires an explicit model for event processing (a *process*) which identifies the context in which events are executed in order to pass part of this context to subsequent events. This would result in more efficient simulations. Moreover, different processes may interact, e.g. we would like to prevent the deletion of connections if they are being used to send a message. Hence, next we extend GT with these two features.

4 Flow Graph Grammars

An important need in modelling DES is the ability to describe the order in which events should be executed and their context of execution. For example, a message has to be sent before it is received. Even though these conditions can be encoded in the LHS and NACs of the rules, it is sometimes simpler to resort to rule invocations, as well as more efficient to provide a data dependency between rules so that the context is passed as a parameter. These dependencies are conceptually the edges of the event graph (cf. Fig. 1), where for the moment we are not taking into account the time scheduling.

This feature is already present in tools like Fujaba, GReAT and VMTS[4], but here we give a novel formalization in terms of DPO, and include event cancelling edges, also new in the GT literature. This formalization will be used in the next section to incorporate a time scheduling distribution function to rule invocations. By separating rule invocation from time scheduling we show how to extend existing tools to handle time.

Hence we start by defining *flow grammars* as a set of productions P with two sets I and C of invocation and cancelling edges between productions. Each edge defines in addition a parameter passing from the source to the target rule. For technical reasons, we define an auxiliary empty rule $\perp = \langle \emptyset \leftarrow \emptyset \rightarrow \emptyset \rangle$, which is used to invoke the initial rules of the flow.

Def. 1 (Flow grammar) *A flow grammar* $FG = \langle P \cup \{\perp\}, end, I, C, G_0 \rangle$ *is made of a set* $P \cup \{\perp\}$ *of rules; a set* $end \subseteq P$ *of final rules; a set of invocation edges* $I = \{(p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k)\}$, *where* $p_i, p_k \in P \cup \{\perp\}$, R_i *is* p_i*'s RHS and* L_k *is* p_k*'s LHS; a set* $C = \{(p_j, R_j \leftarrow M_{jl} \rightarrow L_l, p_l)\}$ *of cancelling edges; and an initial graph* G_0.

Given a rule $p_i \in P \cup \{\perp\}$, *we use the notation* $I(p_i) = \{s = (p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k) \mid s \in I\}$ *and* $C(p_i) = \{s = (p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k) \mid s \in C\}$.

Remark. The structure $R_i \leftarrow M_{ij} \rightarrow L_j$ of invocation and cancelling edges is used to pass the context of execution from R_i to L_j. M_{ij} identifies the elements of R_i and L_j that have to be matched in the same elements of the host graph. If M_{ij} is empty, there is no data dependency, but still rule invocation.

Example. Fig. 4 shows to the left the definition of an invocation edge which passes the node and linked message from rule *init*'s RHS to rule *send*'s LHS. The typing of the message in the $M_{init,send}$ component is abstract, as the typing of the message in L_{send} is abstract too. The figure shows in the center a visual representation of a flow grammar built using the rules of Fig. 3 plus the rule *channelCheck* shown to the right. We use a notation similar to that of event graphs, where each node represents a rule, but in the edges we depict the parameters passed between rules (i.e. the M_{ij}), as these are more informative. For example, the invocation edge depicted to the left is represented as a directed edge decorated with $M_{init,send}$ in this visual representation. The rules marked

[4] `http://www.fujaba.de`, `http://www.isis.vanderbilt.edu/tools/GReAT`, `http://avalon.aut.bme.hu/~tihamer/research/vmts/`

with a tick are the initial rules, which receive an invocation from rule $\perp$. We take the convention of not showing the rule $\perp$ and its invocation edges $I(\perp)$. This event graph representation reveals three processes, given by the connected components. Hence there are three active entities: messages (their process starts in *init* and ends in *end*), connections and channels. The latter makes each channel to lose a message periodically. Processes can interact explicitly by means of cancelling edges. For example, if a connection is used by rule *send*, then we cancel its programmed (i.e. invoked) deletions so that the network connectivity is optimized to the most used connections.

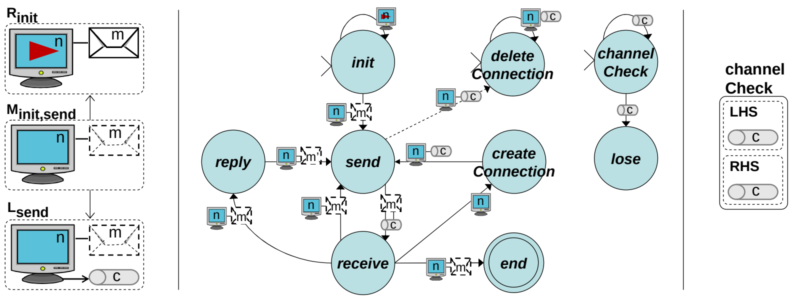


Fig. 4. Parameter passing (left). Flow grammar (center). Additional rule (right).

In order to define the semantics of a flow grammar, first we need to define the system state. This is made of the host graph, plus a set of events storing rule invocations (i.e. elements in I) together with the match in which the rules should be applied (i.e. matches of the invoked rule's LHS).

Def. 2 (Event and state) *Given a flow grammar FG, an* event *is a tuple $e = \langle m\colon L_j \to G, s\rangle$, with $s = (p_i, R_i \leftarrow M_{ij} \to L_j, p_j) \in I$ and $m \models_G p_j$. We write $m(e) = m$, $s(e) = s$, $p(e) = p_j$ to refer to e's match, edge and invoked rule.*

A state *$S = \langle G, E\rangle$ is a tuple made of a graph G, and a set E of events such that $\forall e \in E, m(e) \models_G p(e)$.*

The execution of a flow grammar starts from the matches of the initial rules (those invoked from $\perp$). These matches are converted into events to populate the event set E_0 of the initial system state.

Def. 3 (Initial state) *Given a flow grammar FG, the initial system state init (FG) is given by $S_0 = \langle G_0, E_0\rangle$, where G_0 is the initial graph of FG, and $E_0 = \{(m\colon L_i \to G_0, s = (\perp, \emptyset \leftarrow \emptyset \to L_i, p_i))|s \in I(\perp)$ and $m \models_{G_0} p_i\}$.*

Example. The initial system state for the flow grammar of Fig. 4, taking as initial graph the one shown to the right of Fig. 2, contains one event e whose production $p(e)$ is *init*, 6 events due to matches of *deleteConnection*, and 6 events due to matches of *channelCheck*. Thus, there is an event for each match of the initial rules. These events are the initial starting points for the autonomous execution of each of the three processes in the grammar.

A direct derivation of a flow grammar from a state $\langle G, E\rangle$ consists of taking one event $e \in E$ (if more than one exist, one is taken at random), performing a standard DPO direct derivation using the match $m(e)$, and then calculating the new set of enabled events E'. This set E' contains the old matches in E that were not destroyed by the application of $p(e)$ (set OLD in the definition), and incorporates at most one event for each rule invoked from $p(e)$ (set NEW). Moreover, E' excludes e from the system state, as well as the events cancelled by the cancelling edges $C(p(e))$ (set $CANC$). Please note that the events in E whose match is destroyed by the application of $p(e)$ are not transferred into E'.

Def. 4 (Derivation) *Given a flow grammar $FG = \langle P \cup \{\bot\}, end, I, C, G_0\rangle$, and a state $S = \langle G, E\rangle$, a direct derivation $S = \langle G, E\rangle \overset{e}{\Longrightarrow} S' = \langle H, E'\rangle$ due to the event $e = \langle m_i, (p_s, R_s \leftarrow M_{si} \rightarrow L_i, p_i)\rangle \in E$ is performed as follows:*

- *H is obtained by a standard DPO direct derivation $G \overset{m_i,p_i}{\Longrightarrow} H$, as the left of Fig. 5 shows, where $p(e) = p_i$ and $m(e) = m_i \colon L_i \rightarrow G$,*
- *$E' = NEW \cup (OLD \setminus CANC)$, where:*
  - *$NEW = \{(m_k \colon L_k \rightarrow H, s) \mid s = (p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k) \in I, \nexists (m'_k, s) \neq (m_k, s) \in NEW$, (1) commutes in Fig. 5 and $m_k \models_H p_k\}$,*
  - *$OLD = \{(h \circ m'_j, s_j = (p_k, R_k \leftarrow M_{kj} \rightarrow L_j, p_j)) \mid e_j = (m_j \colon L_j \rightarrow G, s_j) \in E, e_j \neq e, \exists m'_j \colon L_j \rightarrow D$ with $d \circ m'_j = m_j$ (see left of Fig. 5) and $h \circ m'_j \models_H p_j\}$,*
  - *$CANC = \{(m_c \colon L_c \rightarrow H, s'_c) \in OLD \mid s_c = (p_i, R_i \leftarrow M_{ic} \rightarrow L_c, p_c) \in C$, $p(s'_c) = p_c$ and (2) commutes to the right of Fig. 5 $\}$.*

A derivation $S_0 \Rightarrow^* S_n$ is a sequence of zero or more direct derivations.

$$\begin{tikzcd}
 & L_i \arrow[d, "m_i"'] & K_i \arrow[l, "l_i"'] \arrow[r, "r_i"] \arrow[d, "k"'] & R_i \arrow[d, "g"'] \arrow[r, phantom, "(1)"] & M_{ik} \arrow[l] \arrow[d] \\
L_j \arrow[r, "m_j"] \arrow[rr, bend right, "m'_j"', "="] & G & D \arrow[l, "d"'] \arrow[r, "h"] & H & L_k \arrow[l, "m_k"']
\end{tikzcd}
\qquad
\begin{tikzcd}
L_i \arrow[d, "m_i"'] & K_i \arrow[l, "l_i"'] \arrow[r, "r_i"] \arrow[d, "k"'] & R_i \arrow[d, "g"'] \arrow[r, phantom, "(2)"] & M_{ic} \arrow[l] \arrow[d] \\
G & D \arrow[l, "d"'] \arrow[r, "h"] & H & L_c \arrow[l, "m_c"']
\end{tikzcd}$$

Fig. 5. NEW and OLD events (left). CANC events (right).

Remarks. The condition $\nexists (m'_k, s) \neq (m_k, s) \in NEW$ ensures that at most one match of each invoked rule is added to the set of new enabled matches NEW. Hence, if more than one match exists, one is chosen non deterministically.

The set NEW contains at most one event for each rule invoked from e. Such events are demanded to contain a valid match of the LHS of the invoked rule. If no match is found for a certain invoked rule, then no event is generated for it. In this way, the LHS and NACs of the invoked rules are conditions for programming the rules, although in contrast to traditional event graphs, we do not visually show these conditions in the edges (cf. Fig. 4). The set OLD contains the existing events in the system state whose matches are preserved by the rule execution. In fact, all matches that are right-parallel independent with the execution of e are preserved. Finally, the set $CANC$ contains those events in OLD which are cancelled due to the execution of e. It can be noted that cancellation only affects to events in OLD (pre-existing events), so that if an event e both invokes and cancels the same kind of events, invocation prevails.

Example. Fig. 6 shows an example of derivation. The initial system state is given by the graph G and the events to the left (the actual matches in the events are given by equality of identifiers in L_i and G). Applying the match for *send* in the upper left gives as a result graph H. The set of events is updated as shown to the right of the figure: (i) the applied event is removed, (ii) a new event *receive* is added due to the invocation edge coming out from *send* in the flow grammar, and (iii) the old event *deleteConnection* for the match given by objects `n` and `c` is removed due to the cancelling edge. Note how the cancelling edge only removes one of the *deleteConnection* events in the system state, namely the one that contains the node and channel involved in the execution of *send*, which are passed as parameters (cf. Fig. 4). As this shows, cancellation edges cannot always be modelled easily with NACs.

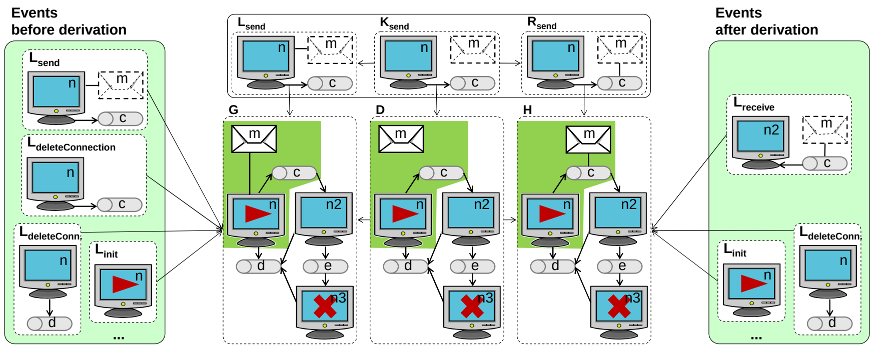


Fig. 6. Example of derivation.

Parallelism. A direct derivation adds to *NEW at most one* match from each invoked rule. However, for certain applications (e.g. to model broadcasting in networks), it is interesting to introduce *all enabled matches* instead. In that case we would just have to remove the condition $\nexists(m'_k, s) \neq (m_k, s) \in NEW$ in Def. 4. This feature is related to the degree of parallelism of the system, called *server semantics* in timed Petri nets [10]. The *single server semantics* assumes that the system can process one invocation at a time, which corresponds to the original Def. 4. The *infinite server* semantics takes into account all enabled matches. The k-server semantics limits the parallelism to at most k matches. These semantics can be included in our model by adding a function $par\colon I \to \mathbb{N} \cup \{*\}$ ("*" for unbounded). We visually annotate the invocation edges in the event graph by placing the value of this function near the arrow end. If no annotation is used in the arrow end, then we assume it is the default value 1.

Next we define the semantics of a flow grammar as the set of all derivations whose last direct derivation was performed by a final rule. We use a set of traces (instead of a set of reachable graphs) in order to take performance metrics.

Def. 5 (Flow grammar semantics) *Given a flow grammar FG, its semantics is defined as* $SEM(FG) = \{init(FG) \Rightarrow^* S_n \overset{e}{\Rightarrow} S_{n+1} | p(e) \in end\}$.

5 Time Scheduling

A flow grammar describes the structure of an event graph, but still lacks the ability to handle time explicitly. That is, we need to introduce an implicit notion of simulation time, and to decorate the edges of the event graph with explicit time values. To this purpose, we extend our grammars with scheduling functions, associating edges with relative time values, or more generally with probability density functions $p(t)$. These distributions give the relative likelihood $p(t)$ of the target rule to be scheduled at relative time t. In this way, we can model either specific times (e.g., 4 using a degenerate distribution δ_4), as well as discrete and continuous distributions, like the uniform, normal and exponential negative.

Def. 6 (Scheduling grammar) *A scheduling grammar* $SG = \langle FG, t_I, t_C \rangle$ *is made of a flow grammar FG, a time scheduling function* $t_I\colon I \to \mathbb{R} \to [0,1]$, *and a time cancelling function* $t_C\colon C \to \mathbb{R} \to [0,1]$.

Remark. Given $s \in I$, $t_I(s)$ maps s to a probability density function $t_I(s)\colon \mathbb{R} \to [0,1]$, which assigns each time value $x \in \mathbb{R}$ a probability $t_I(s)(x)$. Hence at a particular derivation step, we make use of a random variable X_i with density $t_I(s)$, which in the case of scheduling edges can be interpreted as the waiting time before the corresponding rule is applied, and therefore added to the simulation time gives the absolute time the rule application is scheduled for.

Example. The left of Fig. 7 shows part of the example flow grammar annotated with time. For instance, when *send* happens, an event *receive* is scheduled with uniform probability between 5 and 7 units of time later. Rule *init* is scheduled to happen periodically each 50 units of time. The cancelling edge and the invocation edge from *init* to *send* have no timing annotation, so 0 is assumed. *channelCheck* schedules itself at times given by a normal distribution, and then deletes one message, if there is any. This periodic behaviour is similar to a timer.

Now we define the semantics of a timed grammar. For this purpose, we extend states and events presented in Def. 2 with a concrete absolute occurrence time. The time of events should be greater or equal than the current simulation time. The occurrence time of an event is produced when it gets scheduled.

Def. 7 (Timed event and timed state) *Given a scheduling grammar SG, a timed* event *is a tuple $e = \langle m\colon L \to G, p, t\rangle$, where $\langle m\colon L \to G, p\rangle$ is an event according to Def. 2 and $t \in \mathbb{R}$. We write $t(e) = t$, to refer to e's scheduled time.*

A timed state *$S = \langle G, FES, t\rangle$ is made of a state $\langle G, FES\rangle$ and the current simulation time $t \geq 0$, where $\forall e \in FES, t(e) \geq t \wedge m(e) \models_G p(e)$.*

Remark. We use *FES* (future event set) instead of E to remark the similarity of this concept with that of discrete-event systems.

The initial state of a scheduling grammar is a state $S_0 = \langle G_0, FES_0, 0\rangle$, where G_0 is the grammar initial graph, zero is the simulation start time, and FES_0 contains one event for each valid match of the initial rules. These are scheduled to occur at an absolute time given by a set of variables X_i that follow the density function assigned to the scheduling edges from $\perp$. For brevity we do not include the formal definition, straightforward from Def. 3.

A timed derivation step is performed according to Def. 4, but we select the event with lowest time (if several, one is taken at random), and we update the current simulation time to the time of this selected event. In addition, when we schedule a new event, we choose an absolute time equal to the actual time plus a random variable with the probability distribution of the scheduled edge $e \in I$. Finally, given a cancelling edge $c \in C$, we cancel all events that have a greater occurrence time than the current time plus a random variable that follows the probability distribution $t_C(c)$. In the interest of brevity, we avoid duplicating Def. 4, only indicating how the times for events and states are calculated.

Def. 8 (Timed derivation) *Given a scheduling grammar $SG = \langle FG, t_I, t_C\rangle$, and a timed state $S = \langle G, FES, t\rangle$, a direct timed derivation or state change $S = \langle G, FES, t\rangle \overset{e}{\Longrightarrow} S' = \langle H, FES', t'\rangle$ due to the event $e = \langle m_i, (p_s, R_s \leftarrow M_{si} \to L_i, p_i), t'\rangle \in FES$ can be performed iff $\nexists e' \in FES$ with $t(e') < t(e)$. The resulting state S' is calculated as in the untimed case (see Def. 4), while the time of events and the set $CANC$ are calculated as follows:*

- *$\forall e_i \in NEW, t(e_i) = t' + X_i$, s.t. X_i is a random variable with density $t_I(s(e_i))$.*
- *$\forall e_i \in OLD$, its occurrence time $t(e_i)$ remains unchanged (so that e_i "ages").*
- *$CANC = \{(m_c\colon L_c \to H, s'_c, t'_c) \in OLD \mid s_c = (p_i, R_i \leftarrow M_{ic} \to L_c, p_c) \in C,\ p(s'_c) = p_c$, (2) commutes to the right of Fig. 5, $t'_c \geq t' + X_c$, with X_c being a random variable with density $t_C(s_c)\}$.*

Remark. Two conditions are needed for cancelling an event: its match should commute as square (2) in Fig. 5 indicates, and the absolute time of the cancelled event should be greater or equal than the current time plus the relative time the cancelling edge indicates (through a probability distribution). Usually, the relative time t_C of cancelling edges is zero.

Example. The right of Fig. 7 shows a timed derivation like the one in Fig. 6 but considering time. Before applying the timed derivation, the simulation time is 40 and there are scheduled the following events: *send* at time 50, *deleteConnection* at two different matches at time 70, and *init* at time 100. Applying the first scheduled rule, which is *send*, updates the system state as follows: (i) the host graph is modified by the derivation of the DPO rule (not shown, it is performed as depicted in Fig. 6), (ii) the simulation time advances to 50 (as this was the scheduled time for the event), (iii) a new event *receive* is scheduled at time $50 + 6 = 56$, and (iv) one of the *deleteConnection* events is cancelled.

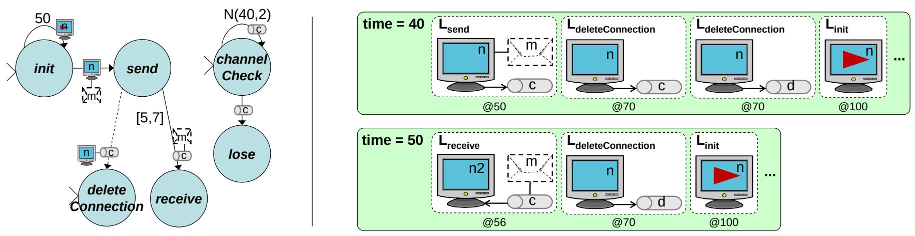
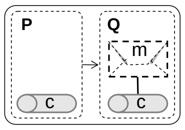


Fig. 7. Scheduling of events (left). Update of events in timed derivation example (right).

The language of a scheduling grammar is similar to that of a flow grammar, but each state is decorated with its absolute time. This is useful to take metrics, as demonstrated next.

5.1 Metrics

One of the objectives of simulation is to obtain metrics on the system behaviour. We can take metrics in three ways. The first one is just observing the occurrence time of events. In particular, the time of the final event in our example tells us the time taken for the initiator node to get a response. The second is by counting the occurrences of events of different types. In our case we can, e.g., count the number of lost messages. The third way involves defining domain-specific metrics. For this purpose we define graph constraints [5] and use them to check the states in which the constraints start to be satisfied or are no longer satisfied, so that we obtain the different time intervals in which the constraint holds. For example, the figure to the right shows the definition of a constraint that is satisfied whenever a channel has at least one message. Then, for all channels (matches of P), we check the states in which we find a message (matches of Q). This allows measuring the utilization time for each channel.



Other more advanced metrics can be taken by *counting* matches. For instance, in our example, we not only check that a match for Q exists, but we count how many of them are in order to measure the utilization level of the channels. Even though theoretically the metrics are defined on derivations of the language, for practical purposes these metrics are taken while the simulation is running, to avoid storing all intermediate states.

6 Modelling Higher-level Timed Primitives

Now we show that our formalism is low-level and general enough to give semantics to other timing schemes and primitives [1, 4, 9, 13, 17].

Three phase approach. One of the features of standard GT is that, when the host graph changes, new matches for the rules of the grammar can be created and then "discovered" by the pattern matching algorithm. In our approach, matches for a certain rule are only sought if the rule is explicitly scheduled.

Inspired by the *three phase approach* [6], we can combine scheduling and activity scanning by extending the definition of scheduling grammar with an additional set $act \subseteq P$. The rules in act represent the start of activities, so that whenever we execute a rule in $P \backslash act$, in addition to scheduling events, we seek *all* matches from rules in act and schedule them for immediate execution. This does not increase the expressive power of our original formalism, but is a shortcut notation that can be modelled by just adding explicit schedulings from all rules in $P \setminus act$ to each rule in act (we schedule all matches, as the "*" indicates), at relative time 0, with empty M_{ij}, as shown to the left of Fig. 8.

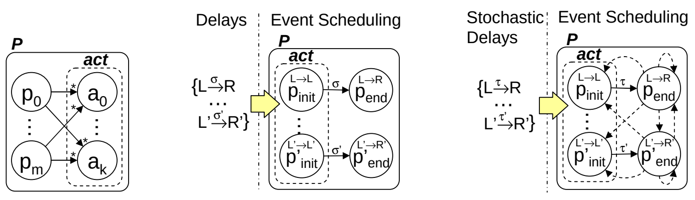


Fig. 8. Activities (left). Rules with delays (center). Stochastic delays (right).

Delays. Delays are used in [4, 17] to extend GT with time. Once a valid match for a rule is found, the execution of the rule at such match is delayed by a time σ (an interval in [4] and other distributions in [17]). We write these rules as $p = \langle L \xrightarrow{\sigma} R \rangle$.

Our events can be used to give semantics to delays. Delayed rules can be seen as activities that do not modify the system state when they start but only when they finish after a delay of σ. Hence, we split a delayed rule p in two, p_{init} and p_{end}, with the former scheduling the latter after σ. p_{init} is the identity rule $L \rightarrow L$, p_{end} is the original rule, and the dependency passes L from p_{init} to p_{end}. The scheme is shown in the center of Fig. 8.

In the semantics of [4], new matches are sought whenever a delayed rule is executed. Its infinite server semantics corresponds to our three-phase approach, where the set act of initial conditions for starting the activities (which in this case are the delayed rules) is given by the events p_{init}. To model the single server semantics of [4], we need to ensure at most one activity of the same type executing on the same set of objects, hence each p_{end} would have a cancelling self-loop with the context of execution as parameter.

Stochastic delays. In [9], GT rules are extended with stochastic delays given by a negative exponential distribution. A rule with stochastic delay $p = \langle L \xrightarrow{\tau} R \rangle$ has similar semantics to a delayed rule, but the difference concerns the *memory* policy when it is executed. After executing a rule, the remaining time of scheduled events has to be *restarted* and *resampled* again. We can model this by using cancelling edges. In particular, we split a stochastic rule in two as before, and in addition, we add cancelling edges from the event p_{end} to each rule p^k in the original stochastic grammar (see the right of Fig. 8). This is so as, at each derivation, we have to "forget the past" stored in the FES.
Activities, duration and conflicts. As seen before, activities are represented by an initial event, a final event, and a duration. However, as a difference from delays, activities may have an observable behaviour when started, and hence p_{init} does not need to be the identity rule. Activities can be interruptible or not. In the first case, the behaviour corresponds with the semantics of our formalism. The behaviour of non-interruptible activities is more complex to model, because an initiated activity *has to be completed*. This means that one cannot schedule the start of new activities if such activities would destroy the match of the final event of running activities. This behaviour can be modelled using FES policies. In this way, a new event at a match $m\colon L \to G$ cannot be scheduled to occur at absolute time t, if $\exists e \in FES$, where e is the end of some activity, with $t(e) \geq t$, and where m and $m(e)$ are in conflict (executing the rule at m breaks $m(e)$).
Timers. Several approaches associate timers to model elements [1, 13]. Timers get an initial value t_o that is decremented as time progresses. When they expire, an action represented by a rule *act* is executed. As the rule *channelCheck* in our example shows, we can model timers by an identity rule identifying the element the timer should be added to, which schedules the rule *act* after t_o time units.
Periodic activities. These are activities that are repeated periodically. In our case, the final event of an activity schedules the initial event of the activity, passing certain elements in the match, like the *init* rule does.

7 Related Work

There are three ways of adding time to GT rules: (i) embedding the time in the host graph (time as data); (ii) incorporating it into the GT formalism (time as control); and (iii) embedding GT into some other simulation formalism.

In the first approach, [8] proposes using time stamps to mark the elements of the host graph. GT rules are standard untimed rules, but two conditions are demanded concerning the manipulation of local clocks: monotonicity (time should progress) and uniformity (time should progress at equal rates locally). In [15], the authors develop a timed approach with the purpose of animating the execution of GT rules. Conceptually, their rules are classified as internal or external events (the latter may be triggered by users), but the timing information is represented in the model, as additional attributes for the different elements. In [3], the author encodes the list of scheduled events in the host graph, and the events that have to be executed are modelled as edges pointing to the different graph elements. In our view, these approaches pollute the model (and the simulation formalism) with timing elements for control purposes.

In the second approach, [4] adapts concepts from timed Petri nets, so that rules are assigned a range, and rule executions are delayed with uniform probability in such range. The work of [9] takes concepts from stochastic Petri nets, so that rules are assigned a delay given by a negative exponential distribution. An important difference is that, while time is assigned to rules in [4, 9, 17], we assign it to schedulings. Hence, while they interpret rules as activities with unobservable initiation, we interpret rules as events, making our approach able to model all of them in a unified way. In [17], events are related to equivalence classes of matches modulo renaming, and time can follow a general distribution. Our approach, based on parameter passing and scheduling, is more efficient as we do not need to compute the equivalence classes at each derivation step.

Other approaches based on rewriting logic follow a similar purpose. In [1] elements in models can be assigned timed constructs like clocks or timers. The work of [13] provides a variety of high-level timed primitives, like periodic activities. Rules can manipulate the FES, mixing both control and data. In our case, a neat separation between control and data is achieved through the use of scheduling and cancelling relations between events.

With respect to the third approach, in [16], GT rules are embedded into the DEVS simulation formalism. Rule concurrency issues are difficult to handle and have to be solved in an ad-hoc way, whereas we use cancelling edges and the theory of GT to eliminate scheduled matches that are no longer valid.

Finally, our work also relates to the models of computations proposed by the embedded systems and systems-on-chip communities [11]. However, whereas we follow the discrete-time model of computation, our approach is not based on modules (processes) and communication channels between these. Instead, our behavioural specifications are decoupled from the actual model where they are executed, allowing its dynamic change.

8 Conclusions and Future Work

Inspired by the *Event Scheduling* world view of discrete-event simulation, we have presented a new way to incorporate time into GT. We model events as rule matches, which may explicitly schedule and cancel the occurrence of other events in the future, and may pass information (partial matches) between such event occurrences for efficiency purposes. We have presented the approach in two steps. *Flow grammars* organize rule flows into processes with parameter passing, formalizing a mechanism that is present in several tools such as Fujaba, VMTS or GReAT. *Scheduling grammars* are built on top of flow grammars adding time in a modular way so that other (untimed) approaches can be extended in a similar way. We have shown that the approach is general enough to model other timing approaches to GT. Finally, the visual nature of GT makes the approach suitable in application domains where simulation is used.

In the future, we will implement tool support for the approach. We also plan to work on analysis methods, both taken from Event Graphs theory [14] and from GT theory, in particular the analysis of rule independence.

Acknowledgements. Work partially sponsored by the Spanish Ministry of Science and Innovation, under project "METEORIC" (TIN2008-02081) and mobility grants JC2009-00015 and PR2009-0019, as well as by the R&D programme of the Community of Madrid, project "e-Madrid" (S2009/TIC-1650). We are grateful to the anonymous reviewers, which helped in improving the paper.

References

1. A. Boronat and P. C. Ölveczky. Formal real-time model transformations in MOMENT2. In *FASE*, volume 6013 of *LNCS*, pages 29–43. Springer, 2010.
2. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems, 2nd Ed.* Springer, 2008.
3. J. de Lara. Meta-modelling and graph transformation for the simulation of systems. *Bulletin of the EATCS*, 81:180–194, 2003.
4. J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3–4):297–326, 2010.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006.
6. G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer, 2001.
7. L. Gönczy, M. Kovács, and D. Varró. Modeling and verification of reliable messaging by graph transformation systems. *ENTCS*, 175(4):37–50, 2007.
8. S. Gyapay, D. Varró, and R. Heckel. Graph transformation with time. *Fundam. Inform.*, 58(1):1–22, 2003.
9. R. Heckel, G. Lajios, and S. Menge. Stochastic graph transformation systems. *Fundam. Inform.*, 74(1):63–84, 2006.
10. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley & Sons, 1995.
11. D. A. Mathaikutty, H. D. Patel, S. K. Shukla, and A. Jantsch. SML-Sys: a functional framework with multiple models of computation for modeling heterogeneous system. *Des. Autom. Embed. Syst.*, 12:1–30, 2008.
12. M. Naeem, R. Heckel, F. Orejas, and F. Hermann. Incremental service composition based on partial matching of visual contracts. In *FASE 2010*, volume 6013 of *LNCS*, pages 123–138. Springer, 2010.
13. J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *VL/HCC'09*, pages 51–55. IEEE, 2009.
14. L. Schruben. Simulation modeling with event graphs. *Commun. ACM*, 26(11):957–963, 1983.
15. T. Strobl and M. Minas. Specifying and generating editing environments for interactive animated visual models. In *GT-VMT'10*, 2010.
16. E. Syriani and H. Vangheluwe. Programmed graph rewriting with DEVS. In *AGTIVE'07*, volume 5088 of *LNCS*, pages 136–151. Springer, 2008.
17. P. Torrini, R. Heckel, and I. Rath. Stochastic graph transformation with regions. In *GT-VMT'10*, 2010.