

Domain-Specific Discrete Event Modelling and Simulation using Graph Transformation

Juan de Lara¹, Esther Guerra¹, Artur Boronat², Reiko Heckel², Paolo Torrini²

¹ Universidad Autónoma de Madrid (Spain), e-mail: {Juan.deLara, Esther.Guerra}@uam.es

² University of Leicester (UK), e-mail: {aboronat, reiko, pt95}@mcs.le.ac.uk

Received: date / Revised version: date

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 to accurately represent the behaviour of a real system and to obtain useful simulation metrics in order to measure throughputs, utilization times and average delays.

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 provide rules with the ability to schedule 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 and provide an implementation on top of the rewriting logic system Maude.

Key words Graph Transformation – Discrete Event Simulation – Domain-Specific Modelling

1 Introduction

Model-Driven Engineering (MDE) proposes the use of models to conduct the different phases of the development process. In this way, models – frequently described using Domain-Specific Modelling Languages (DSMLs) – are used to specify, simulate, test, understand and generate code for the final application. Being of higher level of abstraction than code, they aim to increase the productivity and quality in the development [24].

In order to express and analyse the behaviour of models and model-based systems, Graph Transformation [15,39] (GT) is becoming increasingly popular as

GT rules are intuitive and allow the designer to use the concrete syntax of the manipulated models. Moreover, its formal semantics permits analysing the transformation itself [15]. For example, GT has been extensively used to describe the operational semantics of DSMLs in areas such as reliable messaging in service-oriented architectures [21], web services [30], gaming [44] and manufacturing [9].

When GT is used to specify the semantics of a DSML, 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 simulation purposes and for modelling real-time systems, where the system behaviour depends on explicit timing (e.g. time-outs in network protocols) and performance metrics are essential, a mechanism to model how time progresses during the GT execution is needed.

Computer simulation [47] 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 (not necessarily computer scientists) who are hardly proficient in programming languages, but have deep knowledge on the domain-specific notations used in their scientific domain.

There are several types of computer simulation. In particular, discrete event simulation (DES) [7,20] studies systems where time may be 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 [1,18,20,31,33,34,41,47]. However, these require specialized knowledge that domain experts usually lack, or consist of libraries for programming languages like Java. Therefore simulationists

would strongly benefit if they could define the simulators using the concepts of the domain they are experts in.

In this paper, we propose a language for defining domain-specific behavioural specifications by extending the GT formalism with explicit mechanisms for handling control flow and time. In this way, based on the *event scheduling* approach to simulation [42], we allow rules to program the occurrence of other rules in the future. Our approach makes use of two concepts: explicit rule invocation and cancellation with parameter passing, and time scheduling of rule executions. This improves efficiency in two ways: rule execution is guided by 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. Hence, we keep the best of both worlds: the intuitive, graphical nature of GT rules and its analysis capabilities, and the flexibility and efficiency of the event scheduling approach.

Our approach is specially suited for MDE, where DSMLs are defined and used to describe systems in particular domains. GT rules enable the direct use of the DSML concrete syntax to describe its timed semantics without resorting to encodings in external simulation languages. The visual nature of GT makes the resulting simulator amenable to the visual animation of the domain-specific models. GT rules are also a natural means to express dynamic structural changes, which are generally more difficult to model in traditional simulation approaches, where the structure of the model is usually fixed [41, 47].

This work is an extension of [11]. In particular here we have extended our formalization with match-dependent distribution functions; we propose an approach to specify domain-specific performance metrics based on our formalization; we have developed GT theory to detect errors in parallel schedulings; we report on a tool implementation atop Maude supporting our approach; and we discuss additional case studies and related research.

The paper is organized as follows. Section 2 gives an overview to DES and *event scheduling*. Section 3 introduces the use of (untimed) GT to describe the semantics of DSMLs. Next, Section 4 extends GT with rule invocations and parameter passing, called *flow grammars*. These are extended with time scheduling in Section 5. Section 6 discusses how to model other timed approaches with ours. Section 7 introduces domain-specific simulation metrics. Section 8 presents some case studies, followed by an implementation of the framework using Maude in Section 9. Section 10 covers related research and Section 11 concludes. An appendix details the developed GT theory to detect parallel scheduling errors.

2 Discrete Event Simulation: World Views

Discrete-event systems can be modelled using different styles or *world-views* [7, 20]. Each world-view focuses on a particular aspect of the system: events, activities or processes. An *event* is an instantaneous change in an object 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.

These concepts are illustrated in Fig. 1 through a simple messaging system. The figure shows the arrival of two messages at time t_1 and t_2 respectively, and their dispatch through a channel of capacity one, so that message m2 has to wait while m1 is in the channel. Therefore, a message can perform two activities: waiting or being transported in the channel. The message process is formed by the sequence of these two activities, or just by the second in case the message is immediately sent after its creation. The figure shows in the lower row the state evolution of the system, represented by pairs where the first component is the number of messages in the channel, and the second the number of waiting messages.

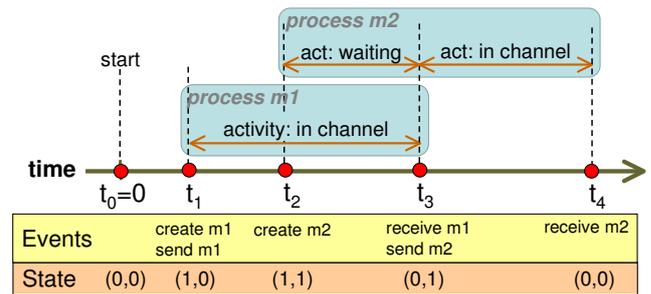


Fig. 1 Discrete-event simulation concepts.

Events, activities and processes give rise to three different approaches to describe discrete time models [7, 31]: *Event Scheduling* focusing on events, *Activity Scanning* focusing on activities, and *Process Interaction* focusing on object processes.

Event Scheduling 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. *Activity Scanning* 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 and check at each moment whether new activities can be started. To increase efficiency, the *three-phase approach* combines *Event Scheduling* and *Activity Scanning* so that the start of new activities is only checked after handling an event. Finally, *Process Interaction* provides constructs to describe the life-cycle (the process) of the active enti-

ties of the system [40]. These entities are called transactions, which move through the different blocks describing their process.

Among the three approaches, Event Scheduling 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 Event Scheduling approach, and in particular on the *Event Graphs* notation [42].

Fig. 2 shows an example event graph. It models the communication network protocol shown in Fig. 1, where a computer sends messages periodically to a receiver computer through a channel with limited capacity cap . The nodes in the event graph represent events. There are two special events: the start and the end of the simulation, identified with a small arrow and a double circle respectively. The state is represented with variables, in our example ch being the load of the channel and w the number of waiting messages. Below the event nodes, a sequence of expressions over variables describes state changes. Arrows between events represent schedulings (i.e. programming of new events in the future). For instance, the arrow from the event *start* to the event *end* indicates 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 some is given) holds at the time the source event is processed [42].

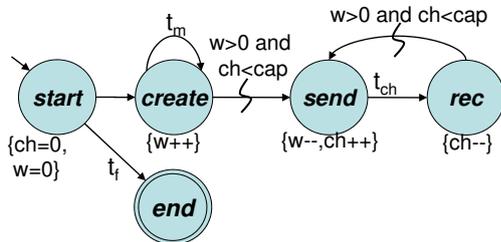


Fig. 2 An event graph model.

Discrete event 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 efficiently the FES [20, 46]. As the approach we will present in this paper uses a FES, it can profit from these algorithms.

Fig. 3 shows some execution steps of the model in Fig. 2, using as parameters $t_m = 5$, $t_f = 100$, $cap = 1$, $t_{ch} = 7$. Each state is enclosed in a rounded box that contains the current time in the upper part, the scheduled events, and the value of the variables ch and w . 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 or deletes events according to the invocation and cancelling edges in the model. The simulation continues until processing the *end* event.

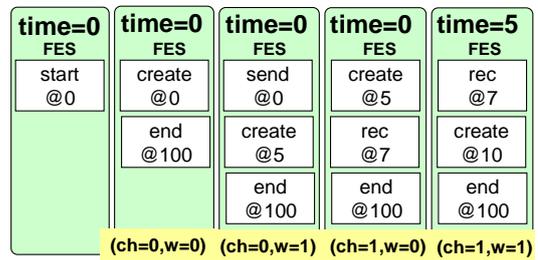


Fig. 3 Some steps in the execution of the model.

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 models with a richer structure in a particular domain. For example, even though the concept of message is relevant in the example, such entity does not appear in the event graph of Fig. 2, but only indirectly through counters (variables ch and w). This is especially inadequate in MDE processes, where specifications of models are conformant to a given DSML. In this paradigm, one needs a way to define simulators using the concepts of the domain in terms of DSMLs. Next we show how GT provides an intuitive formalism for this task but lacks time handling capabilities, which we subsequently add in Section 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 DSMLs. The syntax of DSMLs is usually defined through a meta-model, or type graph, which contains the node and edge types that can be used to define models. For example, Fig. 4 shows a 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 terminals, whose attribute *isFinal* is *true*.

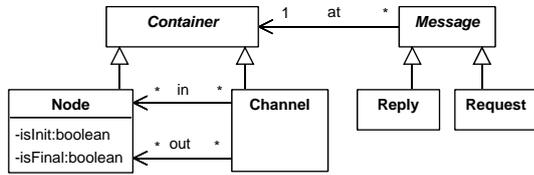


Fig. 4 Meta-model of DSVL for communication networks.

Fig. 5 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.

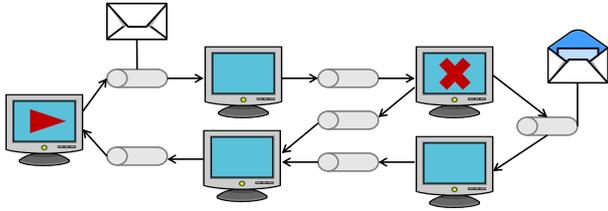


Fig. 5 A communication network model.

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 that traverses the network randomly until it reaches the initiator. Since channels can lose messages, the initiator sends a new request periodically. We are also modelling changes in the net topology, so that nodes can be dynamically connected and disconnected from channels.

We define this protocol and the topology changes by using Double Pushout (DPO) graph transformation rules [15] with the form $p : \langle L \xleftarrow{l} K \xrightarrow{r} R, NAC = \{n_i : L \rightarrow N_i\}_{i \in I} \rangle$. In this approach, L is the left-hand side (LHS) of the rule and contains the elements that need to be present in the host graph for the rule to be applicable. Graph K (kernel or interface) identifies which of the elements in the LHS are not modified by the rule application. Finally, graph R is the right-hand side (RHS) and accounts for the rule post-conditions. In this way, the difference $L \setminus K$ represents the elements deleted by the rule, $R \setminus K$ are the elements created by the rule, and K are the preserved elements. Additionally, NAC is a set of Negative Application Conditions expressing extra graph conditions that prevent the rule application whenever they are found.

As an example, Fig. 6 shows a rule owning two NACs. The rule simulates the dispatch of a message: it deletes the connection between the message and the source node,

and creates a new connection between the message and a channel connected to the node. The rule contains an abstract object m of type message (depicted as a dotted envelope) which can be matched to both requests and replies (subtypes of message in the DSVL meta-model). In this way, this rule becomes a compact specification of two different rules [10,15]. The rule is not applicable 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). In this paper, we sometimes depict rules using just their LHS and RHS, as done in Fig. 8, and use the concrete syntax of the DSVL.

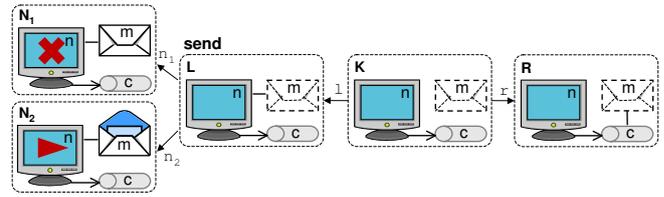


Fig. 6 A DPO rule.

A rule can be applied to a graph if we find a morphism (an occurrence or match) from the rule’s LHS to the graph, and this morphism satisfies the NACs of the rule. A morphism from graph L to G can be thought as an embedding of L in G . Formally, we say that a morphism $m : L \rightarrow G$ is a valid match for rule p in the host graph G , written $m \models_G p$, if m satisfies the glueing conditions [15] and if $\forall n_i : L \rightarrow N_i \in NAC$, then $\nexists n : N_i \rightarrow G$ with $n \circ n_i = m$. The latter condition demands that there is no occurrence of any NAC in the same context where the LHS was found. The glueing conditions include the *dangling edge* and the *identification* conditions. The first condition states that if a rule deletes a node, it should delete all its incident edges as well in order to avoid dangling edges. The second condition concerns non-injective matches that identify several elements in the LHS of a rule with a unique element in the graph. In this case the rule cannot demand deleting some of these elements and preserving others.

If a match m exists and the glueing conditions and NACs are satisfied, we have $m \models_G p$, and hence we can perform a direct derivation. Fig. 7 shows an example. There is only a valid match m from the LHS of the rule to G , the one that identifies the message in the rule with $m1$ in G . The message $m2$ does not belong to a valid match of the rule as it does not satisfy the first NAC. The direct derivation proceeds in two steps: it first deletes from G the elements in $L \setminus K$ yielding graph D (i.e. it disconnects message $m1$ and node $n3$), and then it creates the elements in $R \setminus K$ yielding graph H (i.e. it connects $m1$ to the channel $c2$). Graph D is embedded in both G (which in addition contains the elements deleted by the rule) and H (which in addition contains the elements created by the rule). These embeddings are

given by the morphisms d and h . The two squares in the figure commute ($m \circ l = d \circ k$, $g \circ r = h \circ k$) and are pushouts [15]. The left one is in charge of deleting elements by computing the so-called pushout complement D , whereas the right one is in charge of adding new elements. The execution of a graph transformation system proceeds by applying its rules in non-deterministic order, until none is further applicable.

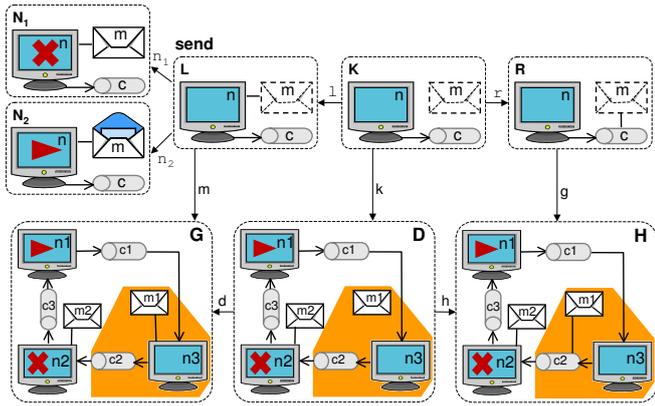


Fig. 7 A direct derivation.

Fig. 8 shows some GT rules of the simulator for our DSVL. Messages are generated from initiators by rule *init*. Nodes can *send* and *receive* messages. As explained before, rule *send* 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. The GT system also contains rules accounting for changes in the topology and modelling channel reliability. In particular, rule *lose* simulates the loss of messages, while rules *createConnection* and *deleteConnection* model the creation and deletion of connections from nodes to channels. The former only applies to nodes without output channels.

For simulation purposes, the standard approach to GT has some advantages. First, the rules are domain-specific, using the concepts and the concrete syntax of the DSVL. Second, the rules can very naturally describe changes in the model topology, more difficult to express with standard simulation techniques. However, GT has two main drawbacks when used for simulation. First, even though the rules capture the untimed semantics of a DSVL, 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, as well as to model transmission delays and the average rate at which channels lose messages. Moreover, some execution paths may not correspond with the behaviour of the system under study, like a path in which the rules

move always the same message and leave “frozen” the other ones.

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 the events are executed and passes part of this context to subsequent events. This would result in more efficient simulations, as matches could be “completed” instead of being sought from scratch. Moreover, different processes may interact. For instance, we may want to prevent the deletion of connections if they are being used to send a message. Hence, next we extend GT with these two features: explicit rule invocation and time.

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 explicit rule invocations, as well as more efficient to provide a data dependency between rules so that the context (a part of the match in which the rule is to be executed) is passed as a parameter.

Hence, our aim is to use an event graph (cf. Fig. 2) as a control structure for rule execution. In our approach, each node in the event graph contains a rule, and the edges of the event graph represent rule dependencies, where for the moment the time scheduling is neglected. Although tools like Fujaba [19], GReAT [2] and VMTS [27] support similar features, here we give a novel formalization in terms of DPO, consider a truly parallel semantics, 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 graph and model transformation tools to handle time.

Thus we start by defining a *flow grammar* as a set of productions P with two sets I and C of invocation and cancelling edges between productions. Each edge defines 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 $I = \{(p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k)\}$ of invocation edges, where $p_i \in P \cup \{\perp\}$, $p_k \in P$, R_i is p_i 's RHS, and L_k is p_k 's LHS;

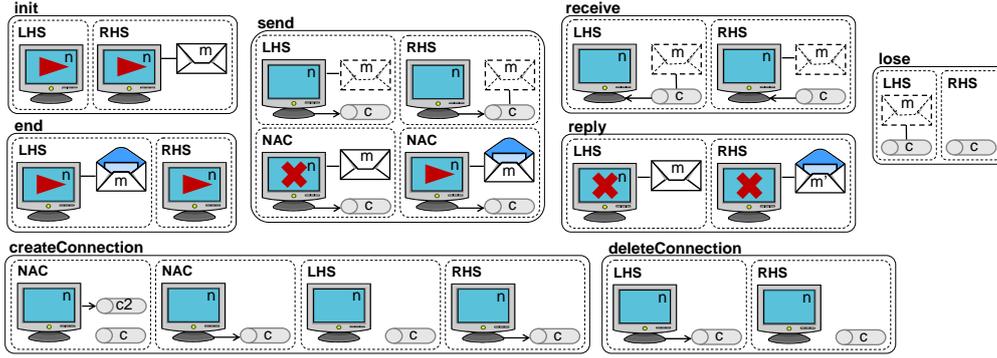


Fig. 8 Some rules of the DSVL simulator.

- a set $C = \{(p_j, R_j \leftarrow M_{jl} \rightarrow L_l, p_l)\}$ of cancelling edges, with $p_j, p_l \in P$;
- 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. 9 shows 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.

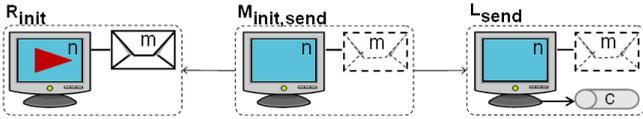


Fig. 9 Parameter passing.

Fig. 10 shows a visual representation of a flow grammar that uses the rules of Fig. 8 plus the rule *channelCheck*, which is a rule containing just a channel in both its LHS and RHS. We use a notation similar to that of event graphs, where each node represents a rule, and the edges show the parameters passed between rules (i.e. the M_{ij}) as this is more informative. For example, the invocation edge depicted in Fig. 9 is represented as a directed edge from *init* to *send* decorated with $M_{init,send}$. The nodes marked with an incoming arrow, with no source, 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 $I(\perp)$.

The event graph in Fig. 10 constrains the behavioural specification given by production rules acting on models such as the one represented in Fig. 5. This fact, together with the parameter passing from one event to

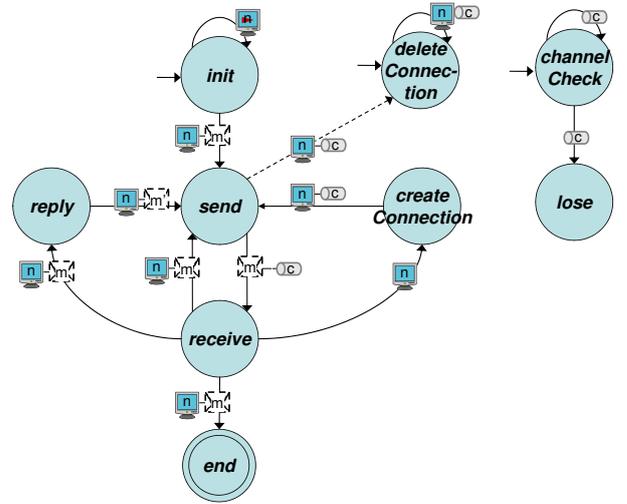


Fig. 10 Flow grammar for the example.

other events, makes our approach share some characteristics with process-interaction approaches. The event graph representation of the example reveals three processes, given by the three initial rules and the components connected through invocation edges. Hence, there are three active entities: messages, connections and channels. The process for messages starts in *init* and terminates in *end*, and the invocation edges in this process show how messages are passed from one event to the invoked ones. The process for channels to the right makes them to lose a message periodically. Processes can interact explicitly by means of invocation and 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 optimised to the most used connections.

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 E of events storing rule invocations (i.e. elements in I) together with the match at which the rules should be applied (i.e. matches of the invoked rule's LHS). Hence, E contains all valid matches of explicitly invoked rules that are ready to be executed.

Def. 2 (Event and state) Given a flow grammar FG , an event is a tuple $e = \langle m: L_j \rightarrow G, s \rangle$, with $s = (p_i, R_i \leftarrow M_{ij} \rightarrow L_j, p_j) \in I$ 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 $E_0 = \{(m: L_i \rightarrow G_0, s = (\perp, \emptyset \leftarrow \emptyset \rightarrow L_i, p_i)) \mid s \in I(\perp) \text{ and } m \models_{G_0} p_i\}$.

Example. Fig. 11 shows the initial system state, taking G_0 in the upper part as initial graph, for the flow grammar of Fig. 10. The initial state contains one event for each match of the initial rules, that is, one event e_0 whose production $p(e)$ is $init$, three events due to matches of $channelCheck$, and three events due to matches of $deleteConnection$ (the matches from the events to the graph are represented by equality of identifiers). These events are the starting point for the autonomous execution of the three processes in the grammar.

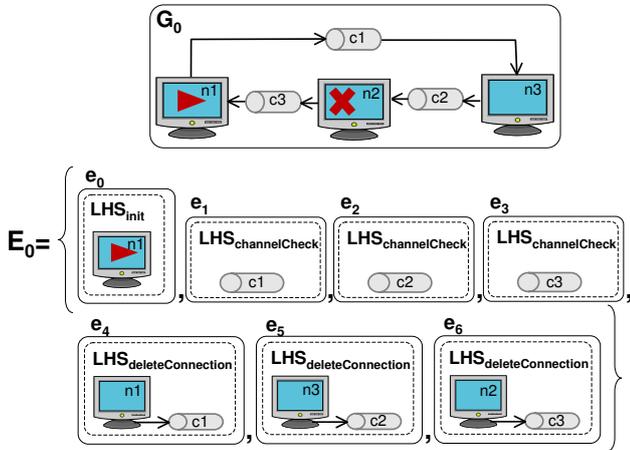


Fig. 11 An example of initial system state for the flow grammar in Fig. 10.

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 following 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 $CANC$),

and the events in E whose match is destroyed by the application of $p(e)$.

Def. 4 (Derivation) Given a flow grammar $FG = \langle P \cup \{\perp\}, end, I, C, G_0 \rangle$, and a state $S = \langle G, E \rangle$, a direct derivation $S = \langle G, E \rangle \xRightarrow{e} S' = \langle H, E' \rangle$ 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 $G \xrightarrow{m_i, p_i} H$, as Fig. 12 shows, where $p(e) = p_i$ and $m(e) = m_i: L_i \rightarrow G$.
- $E' = NEW \cup (OLD \setminus CANC)$, where:
 - $NEW = \{(m_k: L_k \rightarrow H, s) \mid s = (p_i, R_i \leftarrow M_{ik} \rightarrow L_k, p_k) \in I, \nexists (m'_k, s) \neq (m_k, s) \in NEW, (1) \text{ commutes in Fig. 12 and } m_k \models_H p_k\}$,
 - $OLD = \{(h \circ m'_j, s_j = (p_k, R_k \leftarrow M_{kj} \rightarrow L_j, p_j)) \mid e_j = (m_j: L_j \rightarrow G, s_j) \in E, e_j \neq e, \exists m'_j: L_j \rightarrow D \text{ with } d \circ m'_j = m_j \text{ (see Fig. 12) and } h \circ m'_j \models_H p_j\}$,
 - $CANC = \{(m_c: L_c \rightarrow H, s'_c) \in OLD \mid s'_c = (p_i, R_i \leftarrow M_{ic} \rightarrow L_c, p_c) \in C, p(s'_c) = p_c \text{ and (2) commutes in Fig. 13}\}$.

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

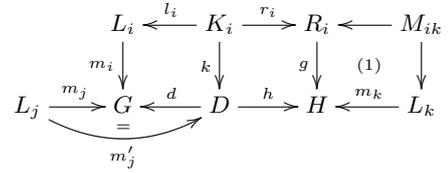


Fig. 12 NEW and OLD events.

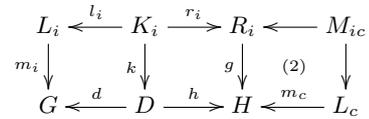


Fig. 13 CANC events.

Remark. The required commutativity of square (1) in Fig. 12 means that the new match $m_k: L_k \rightarrow H$ of the invoked rule has to identify the elements in M_{ik} in the same place as the execution of the previous rule p_i . The required commutativity of square (2) in Fig. 13 means that the matches $m_c: L_c \rightarrow H$ to be cancelled are those that identify the elements in M_{ic} in the same place as the execution of the previous rule p_i .

Remark. 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 . If more than one match exists, one is chosen non-deterministically.

This means that if a rule can be applied in several places (say n), there are n possible different simulation paths at that point. A non-deterministic choice enables the exploration of all such paths. The ability to explore the whole state space is important to achieve completeness (otherwise there could be simulation paths that our system would not produce) and is in line with standard techniques of GT. For the purpose of simulation, repeatability of executions can be achieved by using the same seed to generate the pseudo-random numbers governing the non-deterministic choices.

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 invocation edges, but we show the context of execution passed from the source to the target rule instead.

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 [15] with the execution of e are preserved.

The set $CANC$ contains those events in OLD which are cancelled due to the execution of e . Cancellation only affects to events in OLD (pre-existing events), so that if an event e both invokes and cancels the same kind of event, invocation prevails.

Please note how parameter passing between rules (i.e. non-empty graphs M_{ik}) makes simulations more efficient, as rule matches do not have to be sought from scratch but only completed to $M_{ik} \rightarrow L_k$. This is especially useful in simulation applications, where one describes the flow of the active entities in the system (e.g. messages in our example) and the context of application of the events is passed while these entities evolve.

Example. Fig. 14 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. 10).

As this shows, cancellation edges cannot always be modelled easily with NACs, because these can only refer to *state* conditions present in the host graph, but not

to the execution of other rules. In the standard DPO approach, each rule has to take care of its own execution conditions, using the LHS and an appropriate set of NACs. In flow grammars it is also possible to allocate these responsibilities in other rules, so that one rule may explicitly cancel the programmed executions of other rules. This is a natural way to model inter-process interactions.

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) to be able to take performance metrics. This is so as these metrics are used to observe the evolution of some feature (e.g. the load of a channel) along a sequence of states, given by a trace.

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

Remark. If a trace belongs to $SEM(FG)$, it means that its last step was performed by a final event, and hence the trace has a finite length. Nonetheless, the size of $SEM(FG)$ may not be finite if the flow grammar uses continuous probability density functions, where there is an infinite set of possible choices for the occurrence time of events.

Remark. We can define a set of failure traces $FAIL(FG)$ that contains terminating derivations that did not reach the execution of a final event, as well as infinite derivations.

The use of GT as a basis for the formalization of flow grammars enables the use of its theory [15] for the analysis of some properties of the models, as we will show in next subsection.

4.1 Parallelism

A direct derivation adds to the set 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 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 [28]. 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 $par: I \rightarrow \mathbb{N} \cup \{*\}$ (“*” for unbounded) that assigns a parallelism degree to each invocation edge. 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.

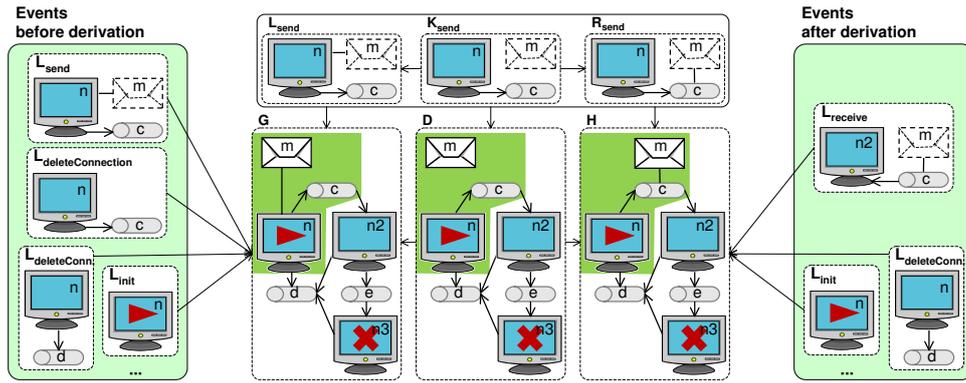


Fig. 14 Example of derivation.

Fig. 15 illustrates the use of annotations to control the parallelism of event invocations. The fragment of the shown flow grammar models broadcasting by introducing two rules, $broadcast_{init}$ and $broadcast_{end}$, the former invoking the latter with “*”. While $broadcast_{init}$ removes the message from the sending node, $broadcast_{end}$ adds a new message in an output channel. The invocation edge with the “*” annotation makes the $broadcast_{end}$ rule to get executed at every existing match commuting with the one in which $broadcast_{init}$ was executed. Hence, the message will be sent to all outgoing channels of node n . Reliable broadcasting (where receivers have to send an *ack* to the sender) could be modelled by providing the nodes with a counter which gets incremented each time a sender receives an *ack* from a receiver. When the counter equals the number of output channels of the sender, it knows that all receivers have processed the request.

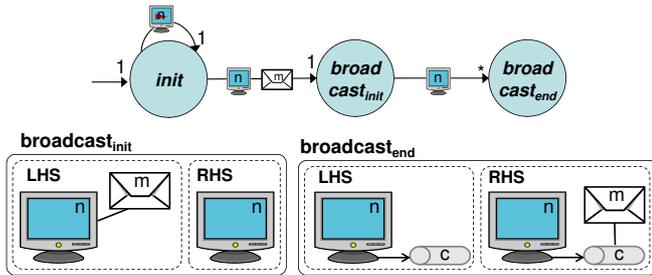


Fig. 15 Controlling the event parallelism.

Defining a flow grammar like the one shown to the left of Fig. 16, with rule $send$ being invoked at all matches, does not produce the desired broadcasting effect. This is so as the “*” semantics considers as many matches as output channels, but then the first match to be executed would disable the others because it would delete the message connection that belongs to the other matches as well, as the message, the connection and the node are passed in the invocation. In this way, we would obtain single server semantics regardless the unbounded annotation. This kind of mistakes can be detected at design time by using the theory of GT to detect all rules that

are invoked with parallel semantics, but which can only get single server semantics. In particular, we have specialized critical pairs analysis [25] for our particular setting. We can detect two kinds of conflicts: *delete-use* and *produce-forbid*.

In the delete-use conflict, the invoked rule deletes some element present in the passed parameters. If that is the case, other matches of the same parallel rule invocation will be inapplicable as the LHS match is destroyed. Fig. 16 illustrates this issue. The right of the figure shows the procedure to detect the delete-use conflict by checking if the elements in M_{ij} are present in the kernel K of the invoked rule. If this is not the case, applying rule $send$ at one match will destroy the other matches in the same invocation. The figure shows that deleting the connection from the message to the node destroys one of the elements passed as parameter, which is needed by other instantiations of the rule in the same invocation, hence indicating a design error. Please note that if a rule preserves the parameters but destroys a different part of the LHS of the invoked rule we could still have conflicts. However, in such a case, not all possible matches would be in conflict and the rule has the chance to be applied more than once.

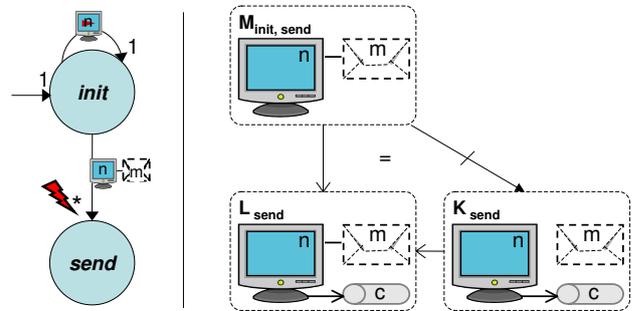


Fig. 16 Delete-use conflict in infinite server semantics.

The second conflict is called produce-forbid, where a rule produces some elements that prevent the same rule from being applied again, as they belong to some of the

rule’s NACs. Due to the technical nature of the detection technique, we illustrate it in the Appendix A.

Annotations can be used in all flow edge types. First, we can label the arc from the \perp rule to each initial rule. By default, our initialisation step in Def. 3 creates one event for every match of each initial rule. With the annotation we control whether all (“*”), at most one (“1”) or at most k (“k”) should be programmed. As we omit the \perp rule from the event graph representations, the annotation is shown visually next to the mark of initial event. We can decorate cancelling edges as well, to denote the cancellation of at most one (“1”), at most k (“k”) or all events (“*”) of a certain type. If no annotation is given, we assume “1” in every possible arc type.

5 Time Scheduling

A flow grammar describes the structure of an event graph, but still lacks the ability to handle time explicitly. Therefore, 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 flow 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. four time units using a degenerate distribution δ_4), as well as discrete and continuous distributions, like the uniform, normal and exponential negative [20].

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 $t_I: I \rightarrow \mathbb{R} \rightarrow [0, 1]$, and a time cancelling function $t_C: C \rightarrow \mathbb{R} \rightarrow [0, 1]$.

Remark. Given $s \in I$, $t_I(s)$ maps s to a probability density function $t_I(s): \mathbb{R} \rightarrow [0, 1]$, 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 invocation 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. Fig. 17 shows 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. This is shown using the interval notation $[5, 7]$. Rule *init* is scheduled periodically each 50 units of time. Among others, the cancelling edge and the invocation edge from *init* to *send* have no timing annotation, so 0 is assumed. Rule *channelCheck* schedules itself at times given by a Normal distribution, and then immediately deletes one message, if there is

any. This periodic behaviour is similar to a timer. Similarly, the *deleteConnection* event gets scheduled periodically using a Poisson distribution with average 4.

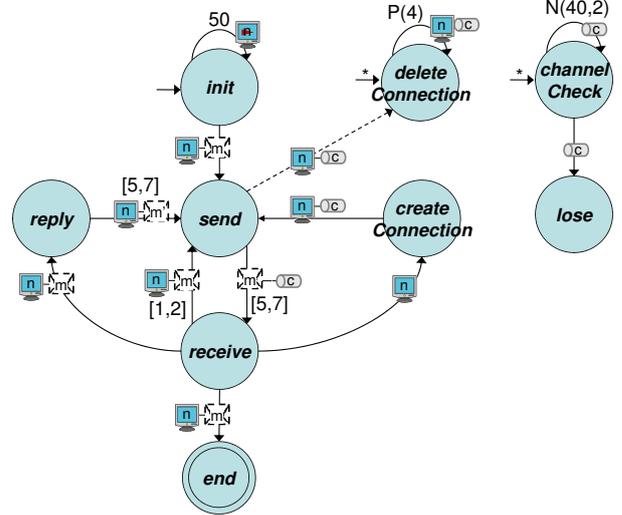


Fig. 17 Scheduling of events.

Match-dependent distributions. For practical purposes, it is important to be able to describe match-dependent probability density functions. For example, the time it takes a message to be received may depend on the current load of the channel. Hence, the density function may depend on certain attribute values given by the match of the rule that is target of the scheduling edge. Formally, this dependency can be expressed by the following scheduling function: $t_I: I \rightarrow (Mor_{AGraph_{TG}} \times \mathbb{R}) \rightarrow [0, 1]$, where $Mor_{AGraph_{TG}}$ is the set of all morphisms in the category of attributed, typed graphs (over a suitable type graph TG). Actually, such function will be meaningful only from morphisms whose domain is the LHS of the invoked rule.

Fig. 18 shows an example of match-dependent distribution. Assuming an attribute *load* in *channels*, we can set a linear dependence between the load of the channel and the transmission delay. In practical implementations, it could be possible to use e.g. the Object Constraint Language (OCL) [32] to obtain suitable features of the model (like the sum of the sizes of the messages in a certain channel) to be included in the distribution functions.

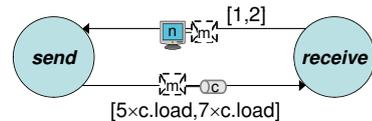


Fig. 18 Example of match-dependent distribution function.

Now we define the semantics of a timed grammar. For this purpose, we extend the 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 $e = \langle m: L \rightarrow G, s, t \rangle$, where $\langle m: L \rightarrow G, s \rangle$ 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 [46].

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 (actually, as many matches as the cardinality annotations marks of the initial states). 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 (immediately if no annotation is given). 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 there are several we take one non-deterministically), 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 $t_C(c)$. For brevity, we avoid duplicating Def. 4 and only indicate how the time for events and states is 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 $S = \langle G, FES, t \rangle \xrightarrow{e} S' = \langle H, FES', t' \rangle$ due to the event $e = \langle m_i, (p_s, R_s \leftarrow M_{si} \rightarrow L_i, p_i), t' \rangle \in FES$ 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’’).
- $CANC = \{ \langle m_c: L_c \rightarrow H, s'_c, t'_c \rangle \in OLD \mid s_c = (p_i, R_i \leftarrow M_{ic} \rightarrow L_c, p_c) \in C, p(s'_c) = p_c, (2) \text{ commutes in Fig. 13, } t'_c \geq t' + X_c, \text{ with } X_c \text{ being a random variable with density } t_C(s_c) \}$.

Remark. Two conditions are needed for cancelling an event: its match should commute as square (2) in Fig. 13 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. Fig. 19 shows a timed derivation like the one in Fig. 14, but considering time and using the scheduling grammar shown in Fig. 17. Before applying the timed derivation, the simulation time is 40 and the following events are scheduled in the FES: *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. 14), (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 (in particular, the one which includes objects *n* and *c* as they are passed in the cancelling edge).

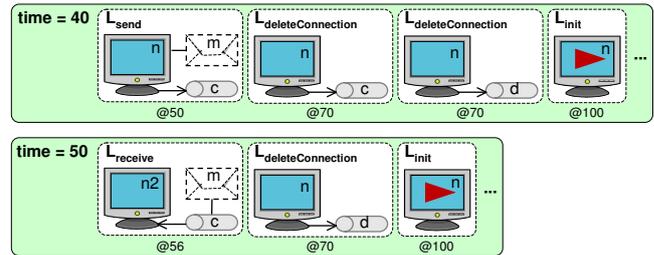


Fig. 19 Example: updating events in a timed derivation.

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 in Section 7.

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 [6, 12, 23, 36, 45].

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. However, in our approach, matches for a certain rule are only sought if the rule is explicitly scheduled.

Inspired by the *three-phase approach* [20], 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 $P \setminus act$, 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, as Fig. 20 shows, it 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*, for all their possible matches (as the “*” indicates), at relative time 0, with empty M_{ij} (so that rules are only invoked but no match is passed).

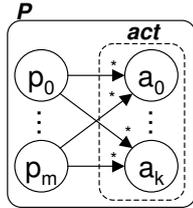


Fig. 20 Activities in the three-phase approach.

Delays. Delays are used in [12,45] to extend GT with time. Once a valid match for a rule is found, the execution of the rule at the match is delayed by a time σ (an interval in [12] and other distributions in [45]). 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 rules, p_{init} and p_{end} , with the former scheduling the latter after σ . Rule 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 Fig. 21, where each initial rule passes its LHS as the context of execution for the end rule.

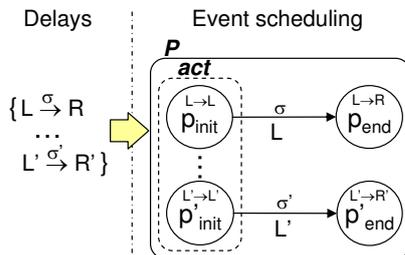


Fig. 21 Rules with delays.

In the semantics of [12], 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 [12], 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 [23], 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 behaviour by using cancelling edges. In particular, we split a stochastic rule in two parts 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 Fig. 22). This is so as, at each derivation, we have to “forget the past” stored in the FES.

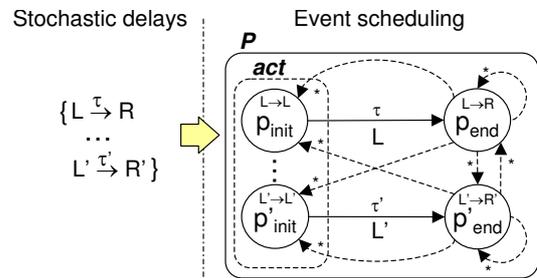


Fig. 22 Stochastic delays.

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. This situation is depicted in Fig. 23, which shows an activity with duration t that passes a context of execution M from the initial to the final rule.

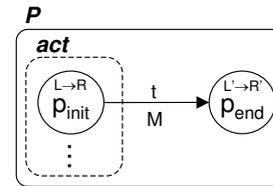


Fig. 23 Activity with duration, initial and final actions.

Activities can be interruptible or not. The behaviour in the first case 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 eventually 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 $m: L \rightarrow G$ cannot be scheduled to occur at absolute time t , if $\exists e \in FES$ with $t(e) \geq t$, where e is the end of some non-interruptible

activity, and m and $m(e)$ are in conflict (i.e. executing the rule at m breaks $m(e)$).

Timers. Several approaches associate timers to elements in the model [6,36]. Timers receive 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. This idea is depicted to the left of Fig. 24, where the LHS of the action rule is not necessarily equal to the identity rule of the timer. The left part of the figure shows a one-shot timer. The timer event p_{timer} passes the execution context L to the action rule act . The right of the same figure shows a periodic timer, which can be interpreted as a periodic activity with unobservable initiation, as we describe next.

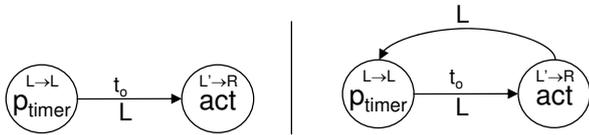


Fig. 24 One-shot timer (left) and periodic timer (right).

Periodic activities. These are activities that are repeated periodically. In this case, the final event of the activity schedules the initial event of the activity, maybe passing certain elements M' in the match, as Fig. 25 shows.

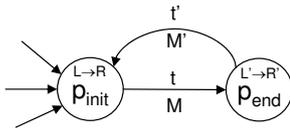


Fig. 25 Periodic activity with initial observable behaviour.

In conclusion, we have seen that our approach is general and primitive enough to model other timed GT approaches in the literature. Moreover, being events and event schedulings the basic building blocks of DES, it is possible to construct and use higher-level primitives – such as activities – when modelling a simulation system.

7 Metrics

One of the objectives of simulation is to obtain metrics on the system behaviour. We can take metrics in three different ways. The first one is by just observing the occurrence time of events in a timed derivation. For instance, the time of the final event in our example is the time taken for the initiator node to get a response.

The second way is by counting the occurrence of events of different types. In our case we can, e.g., measure the number of lost messages by counting how many times event $lose$ occurs in a simulation execution.

The two previous types of metrics have limited expressive power, as one is restricted to measure properties just by counting the execution of events. In our example, with these kinds of metrics, it would be difficult to measure the utilization of a channel. This is so as with rules $send$ and $receive$ we know when a message is added or deleted from a channel, but we need a mechanism to measure, e.g., its average content over the simulation execution, or the total time in which the channel has some message. In this way, we define a third kind of metric to perform domain-specific measurements. For this purpose we can define graph constraints [15] and check the states in which a constraint starts to be satisfied or is no longer satisfied, therefore obtaining the time intervals in which a constraint holds. For simplicity we consider positive atomic graph constraints of the form $a: P \rightarrow Q$, but our approach supports negative as well as more complex constraints as well. Nonetheless, simple positive atomic constraints are highly suitable for our purpose, as they allow identifying specific elements in the system through P , and define properties of interest for them through Q .

Example. The upper part of Fig. 26 shows a constraint $P \xrightarrow{a} Q$ that is satisfied by all channels that have at least one message. Intuitively, this kind of constraints can be interpreted as an implication: if an occurrence of P is found in the graph, then an occurrence of Q should be found as well. Theoretically, an atomic constraint is satisfied on a graph G at match $m: P \rightarrow G$ if there is a match $q: Q \rightarrow G$ commuting with a [15]. A constraint is satisfied globally if it is satisfied for all possible matches $m: P \rightarrow G$.

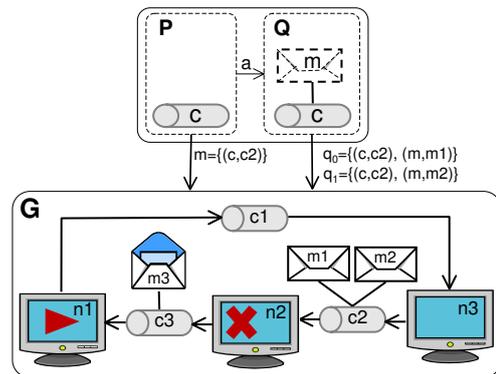


Fig. 26 A constraint and its satisfaction over a host graph.

Next definition presents the notions of constraint enabledness and satisfaction.

Def. 9 (Enabling and satisfaction of constraint) Given an atomic constraint $a: P \rightarrow Q$ and a state $S = \langle G, FES, t \rangle$, the constraint a is enabled at S if there is a match $m: P \rightarrow$

G . We use the notation $ENAB(a, S)$ for the set of all matches $m: P \rightarrow G$.

The constraint a is satisfied at S at match $m: P \rightarrow G \in ENAB(S, a)$, written $m \models_S a$, iff $\exists q: Q \rightarrow G$ s.t. $q \circ a = m$. We use the notation $SAT(a, S)$ for the set $\{m \mid m \models_S a\} \subseteq ENAB(a, S)$.

A constraint is globally satisfied at S iff $\forall m: P \rightarrow G \in ENAB(a, S), m \in SAT(a, S)$.

Example. The constraint in Fig. 26 is enabled at three matches because there are three channels in the graph (three occurrences of P in G). Hence, $ENAB(a, G) = \{m_1 = \{(c, c1)\}, m_2 = \{(c, c2)\}, m_3 = \{(c, c3)\}\}$. The constraint is satisfied at two matches, the ones identifying c with $c2$ and with $c3$, because these channels have a message (i.e. we find at least an occurrence of Q). The constraint is not satisfied in the match that identifies c with $c1$ because $c1$ is empty. Hence, $SAT(a, S) = \{m_2, m_3\}$, and therefore, the constraint is not satisfied globally because $SAT(a, S) \neq ENAB(a, S)$.

We are interested in measuring the span of time during which a constraint is satisfied at a particular match. For this purpose we define the concept of preservation of a constraint at a match in a direct derivation. Intuitively, a constraint is preserved at match m if, after applying a timed direct derivation, the constraint still holds at match m . This notion is similar to the preservation of matches in events (set OLD in Def. 4). We present two notions, the first one refers to the preservation of a match, while the second one refers to the preservation of a constraint at a given match.

Def. 10 (Match and constraint preservation) Given a timed direct derivation $S = \langle G, FES, t \rangle \xrightarrow{\epsilon} S' = \langle G', FES', t' \rangle$, and a match $m: P \rightarrow G$, we say that m is preserved by the derivation iff $\exists k: P \rightarrow D, m' = h \circ k$ s.t. (1) commutes in Fig. 27. We write it $m \xrightarrow{\epsilon} m'$.

Given a timed direct derivation as before, an atomic constraint $a: P \rightarrow Q$, and a match $m \in SAT(a, S)$ that satisfies the constraint; we say that the constraint a is preserved at match m iff $\exists m': P \rightarrow G' \in SAT(a, S')$ with $m \xrightarrow{\epsilon} m'$. We write it $m \xrightarrow{\epsilon} m'$.

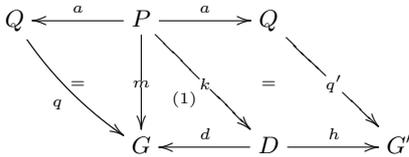


Fig. 27 Match and constraint preservation over direct derivation $G \xleftarrow{d} D \xrightarrow{h} G'$.

Fig. 27 shows the preservation of a constraint at match m by a direct derivation. The preserved match is built as $m' = h \circ k$. The constraint a is satisfied at G' because there exists a match $q': Q \rightarrow G'$ commuting with m' .

Example. Fig. 28 shows the preservation of constraint a at match m . The match $m: P \rightarrow G$ is preserved because the channel persists in the derivation. The constraint is also preserved because the rule application does not delete the channel, but moves one of the messages in the channel to the connected node, leaving the other message in the channel, therefore the constraint is satisfied.

We now introduce the set of successor matches of a given match $m: P \rightarrow G_0$, which is the set of preserved matches along a given derivation. In order to build this set we only require the preservation of the match from the premise P of a constraint $a: P \rightarrow Q$, but not the preservation of the constraint.

Def. 11 (Successor matches) Given a match $m_0: P \rightarrow G_0$ and a derivation $d = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \dots \xrightarrow{e_n} S_{n+1}$, the set of successor matches of m_0 in d is given by $suc_d(m_0, S_0) = \{m_1, m_2, \dots, m_{n+1} \mid m_0 \xrightarrow{e_0} m_1, m_1 \xrightarrow{e_1} m_2, \dots, m_n \xrightarrow{e_n} m_{n+1}\}$.

Example. In the derivation d of Fig. 28, $S = \langle G, FES, t \rangle \Rightarrow S' = \langle G', FES', t' \rangle$, the set of successor matches for $m: P \rightarrow G$ with $m = \{(c, c2)\}$ is $suc_d(m, S) = \{m' = h \circ k\}$.

Next, we define the duration of a constraint at a match as the maximum time span in which the constraint is preserved.

Def. 12 (Duration) Given $m_i \in SAT(a, S_i)$ and a timed derivation $d = S_{i-1} \xrightarrow{e_{i-1}} S_i \xrightarrow{e_i} S_{i+1} \xrightarrow{e_{i+1}} \dots \xrightarrow{e_{n-1}} S_n \xrightarrow{e_n} S_{n+1}$, the maximal satisfaction interval (MSI) of m_i in d is the interval $[S_i, S_n]$ if $\exists m_{i+1} \dots m_n \in suc_d(m_i, S_i)$ s.t. $m_i \xrightarrow{e_i} m_{i+1} \xrightarrow{e_{i+1}} \dots \xrightarrow{e_{n-1}} m_n$, and $\nexists m_{n+1}, m_{i-1}$ s.t. $m_n \xrightarrow{e_n} m_{n+1}$ and $m_{i-1} \xrightarrow{e_{i-1}} m_i$. The duration of m_i in $[S_i, S_n]$ is given by $t(S_n) - t(S_i)$.

Given a timed derivation $d = S_0 \xrightarrow{e_0} S_1 \dots \xrightarrow{e_{n-1}} S_n$, and a match $m_0 \in ENAB(a, S_0)$, the set of maximal satisfaction intervals of m_0 in d is given by $MSI_d(m_0, a) = \{[S_i, S_{i+k}] \mid [S_i, S_{i+k}] \text{ is a MSI for some } m' \in suc_d(m_0, S_0)\}$. The duration of m_0 in $MSI_d(m_0, a)$ is given by:

$$\sum_{[S_i, S_{i+k}] \in MSI_d(m_0, a)} t(S_{i+k}) - t(S_i)$$

Remark. The duration of a match in a derivation is defined as maximal intervals. For example, if a match is satisfied from time 4 to 9 and from 10 to 12, both $[4, 9]$ and $[10, 12]$ are maximal time spans, but not $[5, 9]$. Hence, given a simulation path, we are interested in obtaining those maximal spans, in order to measure the moments when the constraint is satisfied.

Example. The constraint in Figs. 26 and 28 can be used to measure the utilization time of each channel. This is so as the constraint is satisfied whenever the channel has at least one message. Hence, we can calculate the time each channel is busy with respect to the total simulation time. In the derivation in Fig. 28, we have $MSI_d(m = \{(c, c2)\}, a) = \{[S, S']\}$ and its duration is $t(S') - t(S) = 6$.

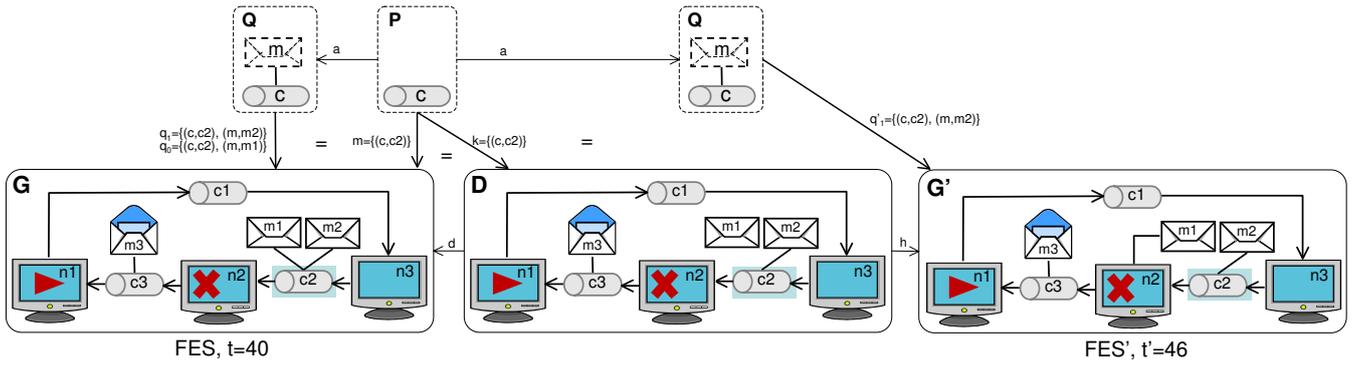


Fig. 28 Preservation of a constraint at the match given by channel $c2$.

Other metric types may require *counting* matches. For instance, in our example, we may not only check that a match for Q exists, but we may need to know how many of them exist in order to measure the utilization level of the channels. This can be seen as a refinement of mere satisfaction, where we count 1 if the match $m \in SAT(a, S)$ and 0 otherwise.

Def. 13 (Satisfaction level) Given an atomic constraint $a: P \rightarrow Q$ and a state $S = \langle G, FES, t \rangle$:

- The satisfaction level of a at S at match $m: P \rightarrow G$ is defined as $\text{satlevel}(S, a, m) = |\{q: Q \rightarrow G | q \circ a = m\}|$, where $|M|$ denotes the cardinality of the set M .
- The satisfaction of a at match $m: P \rightarrow G$ is defined as $\text{sat}(S, a, m) = 1$ if $m \in SAT(a, S)$, and $\text{sat}(S, a, m) = 0$ otherwise.
- The global satisfaction level of constraint a is defined as $\text{satlevel}(S, a) = |SAT(a, S)|$.

Example. In Fig. 26, $\text{satlevel}(S, a, m) = 2$ as there are two messages in channel $c2$, $\text{sat}(S, a, m) = 1$ because a is satisfied at match m , and $\text{satlevel}(S, a) = 2$ because there are two channels with at least one message.

Finally, it is useful to perform a calculation on the attributes gathered by a constraint. For instance, if messages had a certain size, we may be interested in the load of a channel. Moreover, these calculations can be aggregated in several ways to obtain different metrics (e.g. average, minimum, maximum or standard deviation) for each match from Q . Our strategy is to define a function which later can be aggregated over the states in a derivation. For instance, once we know the load of each channel, we can calculate the average network traffic.

Def. 14 (Constraint expression and aggregation) Given an atomic constraint $a: P \rightarrow Q$, a constraint expression is a function $\text{exp}: Mor_{AGraph_{TG}}(Q, -) \rightarrow \mathbb{R}$.

Given an atomic constraint $a: P \rightarrow Q$, a constraint expression exp , a state $S = \langle G, FES, t \rangle$, and a match $m: P \rightarrow G$, a constraint aggregation function is any function of the form $\text{agg}_{\text{exp}}(S, a, m) \rightarrow \mathbb{R}$.

In the previous definition, we have used the notation $Mor_{AGraph_{TG}}(Q, -)$ to denote the set of morphisms

with domain Q . In this kind of metrics, we evaluate such expressions on all matches $q: Q \rightarrow G$ satisfying the constraint a and apply the aggregation function agg_{exp} . In practical implementations, one could use OCL to define the expression function exp and the aggregation function agg_{exp} . There are many useful aggregation functions that can be used in practice, for example:

- **Summatory.** The accumulation of the result of evaluating exp on all commuting matches from Q :

$$\text{sum}_{\text{exp}}(S, a, m) = \sum_{q: Q \rightarrow G | q \circ a = m} \text{exp}(q)$$

- **Minimum.** The minimum value of exp on all commuting matches from Q :

$$\text{min}_{\text{exp}}(S, a, m) = \min\{\text{exp}(q: Q \rightarrow G) | q \circ a = m\}$$

- **Maximum.** The maximum value of exp on all commuting matches from Q :

$$\text{max}_{\text{exp}}(S, a, m) = \max\{\text{exp}(q: Q \rightarrow G) | q \circ a = m\}$$

- **Average.** The average of exp on all commuting matches from Q :

$$\text{avg}_{\text{exp}}(S, a, m) = \frac{\sum_{q: Q \rightarrow G | q \circ a = m} \text{exp}(q)}{\text{satlevel}(S, a, m)}$$

as well as dispersion metrics like the standard deviation.

Example. Fig. 29 shows a domain-specific metric calculated through a constraint expression. Assuming an attribute *size* on messages, we can take the sum of the size of all messages (of any kind) located on a channel to calculate its load using the aggregation function $\text{sum}_{\text{exp}}(S, a, m)$. In a similar way, we can measure the minimum, maximum, average or standard deviation of the size of all messages in a given channel.

Next, we use the functions defined over a specific state, constraint and match (agg_{exp} , satlevel and sat) as the basic building blocks to construct aggregation functions over a derivation d . In this way, we can obtain for example the average or dispersion of the values over certain intervals.

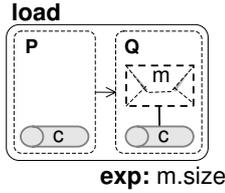


Fig. 29 A metric calculated with a constraint expression.

Def. 15 (Aggregation over timed derivation) Given a scheduling grammar SG , a derivation $d = S_0 \xrightarrow{e_0} S_1 \dots \xrightarrow{e_n} S_n \in SEM(SG)$, a match $m_0: P \rightarrow G_0$, and a function of the form $\mathbf{fun}(S, a, m) \rightarrow \mathbb{R}$ with S a timed state, $a: P \rightarrow Q$ a constraint and $m: P \rightarrow G$ a match, an aggregation function over d has the form: $\mathbf{agg}(d, a, m_0, \mathbf{fun}) \rightarrow \mathbb{R}$.

There are many aggregation functions of interest to calculate different statistics. For example:

- **Weighted Average.** The average of \mathbf{fun} over each interval of the derivation:

$$\mathbf{wavg}(d, a, m_0, \mathbf{fun}) = \sum_{i=0}^{n-1} \frac{\mathbf{fun}(S_i, a, m_i)}{t(S_{i+1}) - t(S_i)}$$

- **Average.** The average of \mathbf{fun} over the time span of the derivation:

$$\mathbf{avg}(d, a, m_0, \mathbf{fun}) = \frac{\sum_{i=0}^{n-1} \mathbf{fun}(S_i, a, m_i)}{t(S_n) - t(S_0)}$$

- **Weighted Sum.** The accumulation of each interval length, weighted by the value of \mathbf{fun} :

$$\mathbf{wsum}(d, a, m_0, \mathbf{fun}) = \sum_{i=0}^{n-1} \mathbf{fun}(S_i, a, m_i) (t(S_{i+1}) - t(S_i))$$

- **Minimum.** The minimum value of \mathbf{fun} during the derivation:

$$\mathbf{min}(d, a, m_0, \mathbf{fun}) = \min\{\mathbf{fun}(S_i, a, m_i) | 0 \leq i \leq n\}$$

- **Maximum.** The maximum value of \mathbf{fun} during the derivation:

$$\mathbf{max}(d, a, m_0, \mathbf{fun}) = \max\{\mathbf{fun}(S_i, a, m_i) | 0 \leq i \leq n\}$$

In all previous functions $m_i \in \text{succ}_d(m_0, S_0)$. It is also possible to define aggregation functions on global metrics, like $\text{satlevel}(a, S)$, which are not measured over a concrete match.

Examples. We can measure the total time during which a channel has some message in a given timed derivation d by using the constraint in Fig. 28 and the aggregation function $\mathbf{wsum}(d, a, m, \text{sat})$ over d . This is so as, if the constraint is satisfied, sat returns 1 and hence the satisfaction interval is accumulated. For the derivation of the figure, $\mathbf{wsum}(d, a, m, \text{sat}) = 1 \cdot (46 - 40) = 6$. This metric

can also be calculated using the length of the maximal satisfaction intervals $MSI_d(m, a)$, as previously shown.

The weighted average number of messages that a channel has during a derivation is given by $\mathbf{wavg}(d, a, m_0, \text{satlevel}) = 2/(46 - 40) = 1/3$, which in this case yields the same value as $\mathbf{avg}(d, a, m_0, \text{satlevel})$. The maximum number of messages in the channels during the derivation is given by $\mathbf{max}(d, a, m_0, \text{satlevel}) = 2$. Finally, using the constraint in Fig. 29 and its expression exp , we can use the aggregation function $\mathbf{wavg}(d, a, m_0, \text{sum}_{\text{exp}})$ to obtain the average over d of the sum of sizes of each message in the channel.

As a final remark, even though the metrics are theoretically defined on derivations of the language, for practical purposes they can be taken while the simulation is running, to avoid storing all intermediate states. Hence, we could extend the simulation state with a further set containing the existing matches of the defined graph constraints. Whenever a timed derivation is performed, we calculate the preserved constraints, delete those that are not preserved, and store new matches of created occurrences of constraints. The handling of these matches is similar to the handling of events in the FES, as described in Def. 4.

8 Case Studies

In this section we provide two further case studies that illustrate different features of our proposal and demonstrate its versatility. The first one builds on the running example, showing the extensibility and scalability of our approach. The second one relies on the use of higher-level primitives, activities in particular, and illustrates the use of domain-specific metrics.

8.1 Reconfigurable Networks

This case study specializes the running example by considering dynamic reconfigurable networks. For this purpose, we incorporate different processes to model the creation and deletion of nodes and channels. The aim of the case study is to show the extensibility of scheduling grammars, and how different processes can interact through invoking and cancelling edges and share events.

Fig. 30 shows the additional events that we add to the scheduling grammar of the running example of Fig. 17, while the left part in dark shows the interaction of the new events with the previous ones. The new events make up four cyclic processes to delete nodes (*choose node*) and channels (*choose channel*), and to create nodes (*new node*) and channels (*new channel*).

While the creation events simply create nodes and channels at random (the latter between two randomly chosen unconnected nodes, see Fig. 31), the deletion is

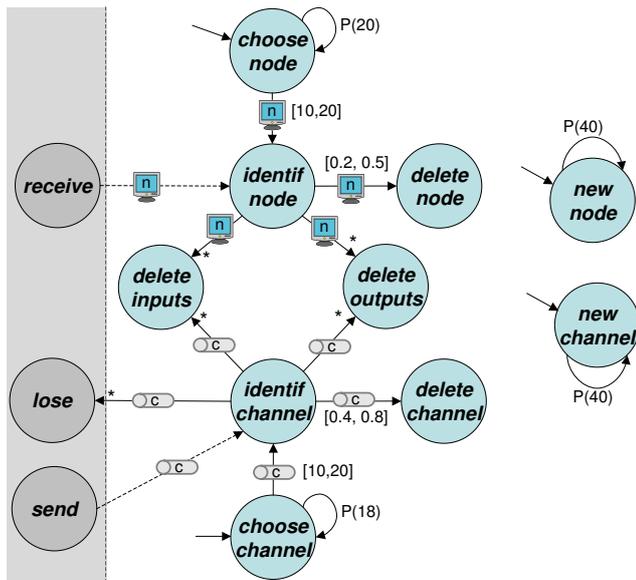


Fig. 30 Scheduling grammar for reconfigurable networks.

more complicated. This is so as nodes in the DPO approach can only be deleted if all their incoming and outgoing links are explicitly deleted as well, or if they have no links at all. In this way, the process of deleting a node is actually made of several activities. First, rule *choose node* shown in Fig. 31 selects the node to be deleted among those that do not have messages. Then, the identity rule *identif node* (with a node in its LHS and RHS) is scheduled after a delay, receiving the identified node as parameter. Immediately after applying this rule, we schedule in parallel the deletion of all incoming and outgoing connections to adjacent channels of the node. Rule *delete outputs*, for example, identifies an output channel of the node and deletes the connection (see Fig. 31). As this rule is scheduled for all possible matches commuting with the node, all output connections will be deleted. Finally, the actual deletion of the node is scheduled after a small delay. We can make this deletion process interact with those of Fig. 17 via cancelling edges. For example, nodes cannot be deleted if they are receiving a message, therefore there is a cancelling edge from event *receive* to event *identif node*. This illustrates that incrementing a simulation model with a new process usually does not involve changing the rules with extra conditions, but only modifying the control flow.

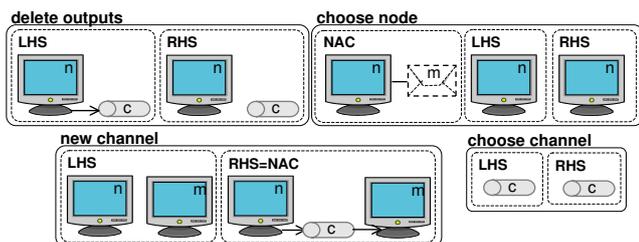


Fig. 31 Reconfiguration rules.

Similarly, we delete channels after deleting all their messages and their input and output connections. Interestingly, we can reuse rules *delete inputs* and *delete outputs*, but this time passing as parameter the channel to be deleted. The process for deleting channels also reuses event *lose* (from Fig. 17) to delete each message of a selected channel. The cancelling edge from *send* forbids deleting the channel if some node is going to use it to send a message. As a remark, please note that in the grammar we do not cancel directly the *choose node* or *choose channel* events, as this would terminate the processes. This is so as there is at most one instance of both processes in parallel (the cardinality of the events is “1”, not “*”). Thus, we cancel events *identif node* and *identif channel* instead. It is not necessary to cancel events *delete channel* or *lose* as these are only scheduled once *identif channel* gets executed.

Altogether, this example demonstrates that the parallel semantics of scheduling grammars allows their extension by simply incorporating new processes. The new processes can interact in several ways: they can reuse events (normally, terminal events that do not schedule other events), and they can invoke or cancel events of other processes. This presents advantages with respect to using standard plain GT rules, where one needs to encode in each rule all possible conditions for its (non-)executability. Instead, cancelling and scheduling edges permit assigning such responsibilities to other processes, hence facilitating extensibility in a more controlled way.

8.2 Factories

The second example comprises a DSML for describing plant factories and its simulator. With this example we want to illustrate the aggregation of events into activities expressing the behaviour for the entities in a system. The meta-model of the DSML is shown in Fig. 32. Thus, a factory is made of different types of machines which either feed different part types into the production plant at a given rate (generators) or consume and produce transformed parts from/to conveyors (transformers). Transformer machines are able to process a number of parts up to their maximum capacity, and need to be operated by humans in order to start processing new parts.

Fig. 33 shows a factory model where a generator of cylinders and another of bars put parts in the same conveyor with rates 6 and 8 respectively. This conveyor holds one cylinder and one bar. An assembler machine, which is operated by a worker, has a maximum capacity of 4 and is currently processing 2 parts. The model also contains a packaging machine that is not being attended by any operator.

Once we have described the DSML syntax, we can use our approach to model its timed semantics. Fig. 34 shows in the upper part two rules modelling the behaviour of generators for cylinders and bars, which put

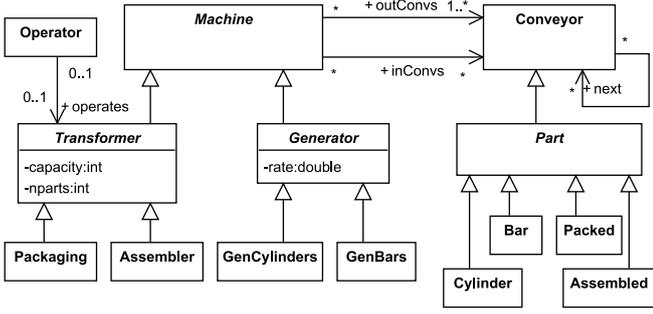


Fig. 32 The meta-model for the factory DSML.

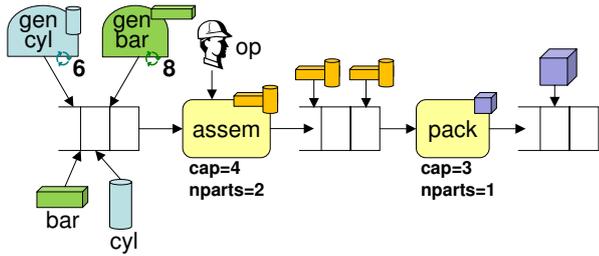


Fig. 33 A factory model.

one part of the appropriate type into a conveyor. The rules schedule themselves, hence they can be considered as periodic, atomic activities. This is modelled through the events and invocation edges shown below the rules. They use the production rate of the generators as the average of a poisson distribution for the scheduling.

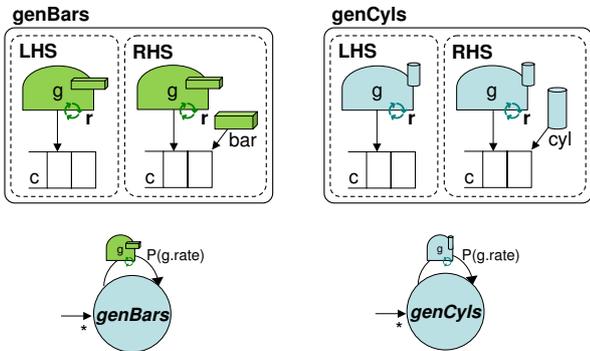


Fig. 34 Rules for generators (top) and scheduling (bottom).

Fig. 35 presents to the left the rules for the assembler machines. These need to have an operator for starting their work, in which case they assemble one cylinder and one bar from their input conveyor, provided that the machines have enough capacity. The assembly is initiated by rule *assemble_init*, which schedules the production of the assembled part for the given machine by rule *assemble_end*. Hence, both rules together represent an activity given by an initial and a final event, representing the behaviour of assembler machines. The right of the figure shows the activity using an abstraction which depicts the rules making an activity enclosed in a rounded rectangle. The probability function for the scheduling de-

pends on the load of the particular assembler machine: the higher the load in a machine, the bigger the assembling time.

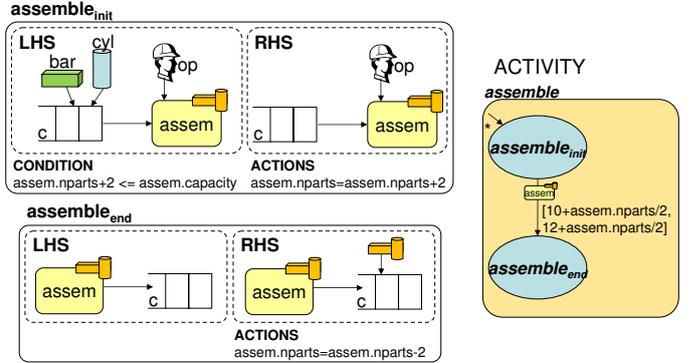


Fig. 35 Rules for assembler machines (left) and scheduling (right).

The behaviour of packaging machines, the transport of parts between adjacent conveyors, and the movement of operators between machines can be represented as activities as well. For example, Fig. 36 shows the activity that moves parts of any type (we use an abstract object) across two adjacent conveyors, whereas Fig. 37 shows the activity that moves an operator between two transformer machines of any type (we use abstract machines) provided that the target machine has some part waiting in an input conveyor but has no operator (NAC).

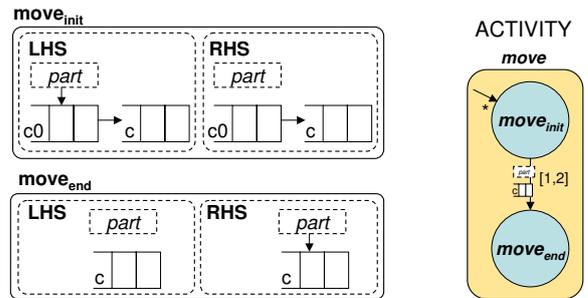


Fig. 36 Rules for moving parts across conveyors (left) and scheduling (right).

Fig. 38 shows the whole scheduling grammar of the factory simulator. We require non-interruptible activities. Moreover, following the “three-phase approach” shortcut introduced in Sec. 6, we include the initial event of every activity in the set *act* of activities so that they do not need to be explicitly invoked by the others, but their start event is scanned after the execution of the rules in the set. The grammar includes a final rule *end* with empty LHS and RHS and scheduled at time 1000, which signals the simulation end.

Fig. 39 shows some metrics for the example. We use the one on the left to measure the load of the con-

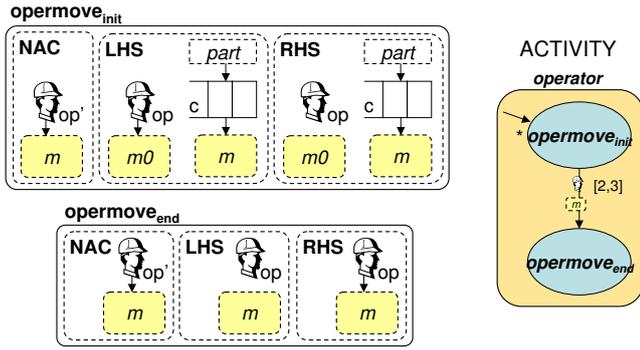


Fig. 37 Rules for moving operators between machines (left) and scheduling (right).

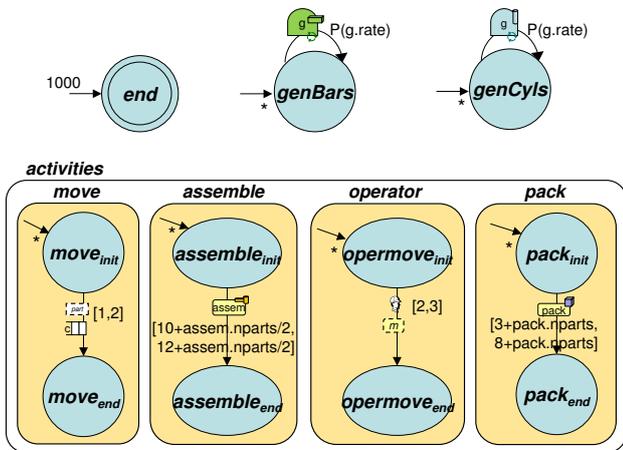


Fig. 38 Scheduling grammar of the factory simulator.

veyors. We can use the *weighted average* aggregation function over a derivation d to obtain the average load: $wavg(d, a, m_0, satlevel)$. We can also investigate the so-called *zero-entries*, that is, the total time a conveyor is empty. This can be done either by performing the weighted sum of the satisfied intervals and then subtracting from the total: $t(S_n) - wsum(d, a, m_0, sat)$, or by performing the weighted sum using the inverse of the **sat** function ($\overline{sat} = 1 - sat$) as follows: $wsum(d, a, m_0, \overline{sat})$.

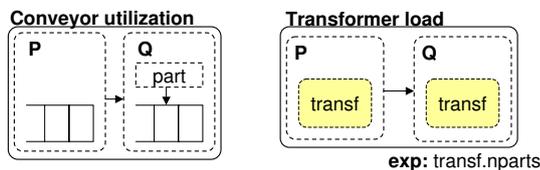


Fig. 39 Some metrics of the factory simulation.

The metric to the right of Fig. 39 measures the load on a transformer machine (either assembler or packaging) by using an expression that returns its number of parts. We can use this metric to obtain the maximum load of a machine during a derivation: $\max(d, a, m_0, \text{sum}_{\text{exp}})$. In this case, it is irrelevant which aggregation function

we use over the constraint expression sum_{exp} as there is exactly one match from Q for every match from P .

In conclusion, in this case study we have seen that our approach enables the definition of higher-level constructs, like activities. As a difference from our solution, approaches based on rule delays [12,23,45] represent each activity as a single rule, and therefore would not be able to represent activities with initial observable behaviour as required by this system.

9 Implementation

In this section, we present an executable, algebraic prototype for performing simulations of graph-based systems implementing the notions described in this paper using Maude [8]. We have chosen Maude due to its reflection-based facilities for defining rewriting strategies, which proved to be very useful for implementing the control flow constructs proposed in this work. The representation of a graph-based system as a rewrite theory in Maude [4,38] is introduced in Section 9.1; the specification of the notion of flow grammar and the untimed simulation scheme are defined in Section 9.2; the notion of scheduling grammar and the event scheduling scheme are defined in Section 9.3; and the specification and evaluation of metrics are considered in Section 9.4.

9.1 Graph-based Systems in Maude.

Maude programs are composed of functional modules and system modules. While functional modules are used to represent algebraic data types (specifying their types using sorts and subsorts) and their operations, a system module specifies local concurrent transitions in a system, whose states are algebraically characterized by the functional module. A functional module is defined as an expression $\text{fmod } n \text{ is } (\Sigma, E) \text{ endfm}$, where:

- n is the name of the module,
- Σ is a signature specifying sorts s with the clause $\text{sort } s \text{ .}$, subtyping relationships between sorts s_1 and s_2 with the clause $\text{subsort } s_1 < s_2 \text{ .}$ (i.e. s_1 is a subsort of s_2), and operation signatures of the form $\text{op } f : s_1 \dots s_n \rightarrow s \text{ .}$, where s_i, s are sorts in Σ , and
- E is a collection of (possibly conditional) equations of the form $(c) \text{eq } t = t' \text{ (if cond) .}$, where t and t' are terms with variables over the signature Σ and cond may be a conjunction of equations and memberships, and (possibly conditional) memberships of the form $(c) \text{mb } t : s \text{ (if cond) .}$, where t is a term as above, s is a sort in Σ , and cond is a condition as above. Additionally, E also contains axioms for some binary operations, such as associativity, commutativity and identity.

A system module $\text{mod } n \text{ is } (\Sigma, E, R) \text{ endm}$ extends a functional module with (possibly conditional) rewrite

rules R of the form $(c)r1\ t \Rightarrow t' \text{ (if cond) } .$, where t and t' are terms as above, and cond may be a conjunction of equations, memberships and rewrites. Moreover, we can also use so-called matching equations of the form $t := t'$ as an atomic formula in a condition cond , where t is a term with variables over the signature Σ that are instantiated by matching the term t against the canonical form of the subject term t' , where the term t must contain free variables satisfying certain technical requirements [8], which are not relevant for the presentation of the implementation in this section.

In this work, we use functional modules for representing meta-models as data types, for representing models as directed graphs and for encoding the deterministic procedures that form part of the simulation scheme; and we use system modules representing the non-deterministic aspects of the simulation scheme.

9.1.1 Type Graphs and Instance Graphs. A meta-model is syntactically formalized as an order-sorted signature Σ and a set of equations [5]. Σ can be split into a generic part providing constructs (sorts, subsorts and operators) for defining directed graphs of objects, and a meta-model-specific part containing type-relative information (metaclass names and properties). In the generic part, an object is represented by a triple $\langle o : C \mid PS \rangle$ using the operator $\text{op} \langle \cdot \rangle : \text{Oid Cid PropertySet} \rightarrow \text{Object}$. so that o is a unique object identifier¹, C is a class name, and PS is a record where each field represents either a slot (an attribute value) or a reference (a collection of pointers to other objects). Objects can be grouped in collections of objects by means of an associative, commutative union operator (denoted by juxtaposition) $_$ with identity none . A model is syntactically represented as an object collection that is wrapped with the operator $\langle \langle _ \rangle \rangle$ shown in Listing 1.

```

1 subsort Object < ObjCol .
2 op none : -> ObjCol .
3 op _ : ObjCol ObjCol -> ObjCol [assoc comm id: none] .
4 op <<_>> : ObjCol -> Model .

```

Listing 1 Syntactic representation of model.

In the meta-model-specific part of the signature Σ , there is a sort for each metaclass name and there is a constant defining this sort if the metaclass is not abstract. Metaclass inheritance is formalized by means of subsort relationships between the sorts corresponding to each metaclass. Taking into account the resulting subsort ordering, maximal sorts in this ordering are declared as subsorts of the sort cid . In addition, properties defined for each metaclass are encoded as constructors of the sort Property defining fields for the record PropertySet . Listing 2 shows the encoding of the domain-specific part of the signature corresponding to the meta-model in Fig. 4.

¹ We define identifiers using literals prefixed with a quote, such as 'a' or '1', using the built-in datatype Oid , which is defined as a subsort of the sort oid .

```

1 sorts Container Node Channel Message Reply Request .
2 subsorts Container Message < Cid .
3 subsorts Node Channel < Container .
4 subsorts Reply Request < Message .
5 --- Concrete metaClasses
6 op Node : -> Node . op Channel : -> Channel .
7 op Reply : -> Reply . op Request : -> Request .
8 --- Node properties
9 op isInit : -> PropName . op isFinal : -> PropName .
10 --- Channel properties
11 op in : -> PropName . op out : -> PropName .
12 --- Message properties
13 op at : -> PropName . op size : -> PropName .

```

Listing 2 Encoding of meta-model in Fig. 4.

The model in Fig. 5 is represented by including a term as shown in Listing 3.

```

1 << < 'a : Node | isInit : true, isFinal : false >
2 < 'b : Channel | in : 'a, out : 'c >
3 < 'c : Node | isInit : false, isFinal : false >
4 < 'd : Channel | in : 'c, out : 'e >
5 < 'e : Node | isInit : false, isFinal : true >
6 < 'f : Channel | in : 'e, out : 'g >
7 < 'g : Node | isInit : false, isFinal : false >
8 < 'h : Channel | in : 'g, out : 'a >
9 < 'i : Channel | in : 'e, out : 'j >
10 < 'j : Node | isInit : false, isFinal : false >
11 < 'k : Channel | in : 'j, out : 'g >
12 < 'l : Request | at : 'b >
13 < 'm : Reply | at : 'i > >>

```

Listing 3 Encoding of model in Fig. 5.

9.1.2 Production Rules. GT rules are encoded as rewrite rules of the form $r1 \langle \text{label} \rangle : \langle \text{LHS} \rangle \Rightarrow \langle \text{RHS} \rangle$ and $[c]r1 \langle \text{label} \rangle : \langle \text{LHS} \rangle \Rightarrow \langle \text{RHS} \rangle \text{ [if } \langle \text{NAC-LIST} \rangle]$ where $\langle \text{label} \rangle$ is an identifier for the rule, $\langle \text{LHS} \rangle$ and $\langle \text{RHS} \rangle$ are terms with variables of sort Model , and $\langle \text{NAC-LIST} \rangle$ is a list of negative application condition statements separated by the conjunctive connective \wedge . As an example, Listing 4 shows rule *send*, which moves a message from a node to a channel unless the message is a request and the node is terminal (NAC RequestInTerminal in line 6), or the message is a reply and the node is initial (NAC ReplyInInitial in line 9).

```

1 cr1 [send] : << < N : Node | PS1 >
2 < M : Message | at : N, PS3 >
3 < C : Channel | in : N, PS2 > OC >> =>
4 << < N : Node | PS1 > < M : Message | at : C, PS3 >
5 < C : Channel | in : N, PS2 > OC >>
6 if nac("RequestInTerminal", NodeId(N) MessageId(M),
7 < N : Node | PS1 > < M : Message | at : N, PS3 >
8 < C : Channel | in : N, PS2 > OC)
9 /\ nac("ReplyInInitial", NodeId(N) MessageId(M),
10 < N : Node | PS1 > < M : Message | at : N, PS3 >
11 < C : Channel | in : N, PS2 > OC) .

```

Listing 4 Encoding of rule *send* from Fig. 6.

NACs are defined with the boolean operation nac that receives as arguments an identifier for the NAC, an environment containing assignments for some variables in the NAC, and the collection of objects that constitutes the syntactic representation of the model. A NAC is defined with an equation by providing the pattern that is forbidden in the third argument, which must return false. Listing 5 shows the definition of the NACs in the previous rule *send*. By default, the equation in line 2 assumes that a NAC fails if no pattern corresponding to

a specific NAC can be matched, by returning true. The operations in line 4 define the mechanism to instantiate variables in the equations defining NACs, and the equations in lines 5 and 9 define the NACs used in the rule *send*.

```

1 op nac : String Environment ObjCol -> Bool .
2 eq nac( Name, E, OC ) = true [owise] .
3
4 ops NodeId MessageId : Oid -> EnvVar .
5 eq nac("RequestInTerminal", NodeId(N) MessageId(M),
6 < N : Node | isFinal : true, PS1 >
7 < M : Request | at : N, PS3 >
8 < C : Channel | in : N, PS2 > OC ) = false .
9 eq nac("ReplyInInitial", NodeId(N) MessageId(M),
10 < N : Node | isInit : true, PS1 >
11 < M : Reply | at : N, PS3 >
12 < C : Channel | in : N, PS2 > OC ) = false .

```

Listing 5 Encoding of NACs for rule *send*.

All rules in Figs. 8 and 10 are defined in the Appendix B.

9.2 Untimed (Logical) Simulation

In this section, we present the Maude simulation engine that enables the simulation of flow grammars, as presented in Section 4.

9.2.1 Flow Grammar. Both invocation and cancellation edges are represented as tuples $\text{flowEdge}(SC, PR, TG)$, where: SC and TG correspond to labels identifying the source and target rules, respectively; and PR is a binary relation defining the parameter passing structure. A flow grammar is a tuple $\text{flowGrammar}(MOD, End, I, C, G)$ where: MOD is the module containing the encoding of the production rules of the flow grammar as rewrite rules; End is a set of rule labels identifying those rules that terminate the simulation process; I and C are sets of flow edges constituting the sets of invocation and cancellation edges, respectively; and G is a term of sort $Model$ representing the initial graph.

9.2.2 Simulation Data Types. An event is given as a tuple $\text{edge}(FE, M, G)$, where FE is the flow edge whose target rule has been applied, M is the match of the LHS of the target rule over the host graph, and G is the graph resulting from the application of this rule with the match M . For implementation purposes, the notion of simulation environment emerges and is defined as a term $\text{env}(FG, SS)$, where FG is a flow grammar and SS is the simulation state.

9.2.3 Simulation Scheme. The parameter passing in invocation and cancelling edges involves the capability of both defining parameterised rules and instantiating (some of) their arguments with specific values, constraining the number of events that can be enabled by the target rule. We have chosen Maude for implementing the proposed simulation scheme because it provides facilities for

handling these mechanisms through its meta-level operations by means of which rules can be applied in an explicit way.

The main simulation scheme presented in Def. 4 is implemented by three rewrite rules defined over a term of the form $\text{env}(FG, SS)$. The first rule, shown in Listing 6, constitutes the core mechanism for firing events and it applies when there are non-terminating events enabled in the simulation state component events , as checked in the condition in line 10. In line 11, the operator processChoice is used to split the set of currently enabled events into a pair of disjoint sets. The event to be fired is chosen from the first component. For untimed simulation, this operator defines a pair where the first component is the original set of events and the second component is empty. However, this operator is crucial for handling timed events as we explain in the next section. This rule can only be applied if the chosen event is not final. This is handled by the conditions in line 12, which check that the label of the rule that was used to create the event is not in the set End of final rule labels.

The event is fired by replacing the host graph G_1 by the precomputed resulting graph G_2 obtained from the event E in the component model , through the matching equation in line 13. The set of new events newES (resp. cancelling events cancES) is obtained by applying the function computeEvents , in line 14 (resp. 16), for each invocation edge (resp. cancellation edge) whose source rule component coincides with the rule that was used to create the event E .

In line 4, the resulting set of enabled events is obtained by updating the remaining set of events, namely the union of ES_2 and ES_3 with the new graph G_2 through the function refreshOld . Given the module MOD containing the definition of production rules, the graph G_1 representing the current state of the system under simulation, and a set $\text{ES}_2 \text{ ES}_3$ of enabled events (with precomputed matches and output graphs), the function refreshOld checks whether each event in $\text{ES}_2 \text{ ES}_3$ is still valid by creating an event with the precomputed match and the same rule over the graph G_2 . The resulting set corresponds to those events that remain valid after firing event E and that have not been cancelled. The resulting set of enabled events is formed by adding the set newES of events that are enabled by event E over the system state G_1 .

Finally, the function updateGlobalTime in line 18 is used for updating the global clock when it is available, and the function recordData in line 8 is used to record simulation data to be used for analysis purposes by using (domain-specific) metrics.

```

1 crl env( flowGrammar(MOD, End, I, C, Model),
2 model(G1) events( ES1 ) SS1 ) =>
3 env(flowGrammar(MOD, End, I, C, Model), model(G2)
4 events(newES
5 applyCancEvents(
6 refreshOld( MOD, G2, ES2 ES3, empty ) ,
7 cancES)
8 recordData(E, SS4)
9 )
10 if ES1 /= empty /\

```

```

11 < E ES3, ES2 > := processChoice(ES1) /\
12 TGR := getTarget(getEdge(E)) /\ TGR in End = false /\
13 G2 := getTerm(E) /\
14 events(newES) SS2 := computeEvents(MOD, E,
15   nextEdges(I, TGR, empty), events(empty) inv SS1) /\
16 events(cancES) SS3 := computeEvents(MOD, E,
17   nextEdges(C, TGR, empty), events(empty) canc SS2) /\
18 SS4 := updateGlobalTime( E, SS3 ) .

```

Listing 6 Main rule of the implemented simulator.

A second rule is applied when the chosen event is final, in which case the simulation process ends after firing the event. A third rule deals with the case when there are no enabled events left in the simulation state component `events`, i.e. there is a *deadlock* since a terminating event cannot be fired.

9.3 Event Scheduling

In this section, we cover a number of variation points in the untimed simulation scheme that allow us to extend the untimed simulation scheme as an event scheduling scheme. This is achieved by enriching the simulation state with the set `ES` of scheduled events, with a component `clock(T)` with the global time of the simulation, and components `invClock(RVS1)` and `cancClock(RVS2)` representing the stochastic clock structures for invocation and cancellation edges.

The extensions concern the choice of the event to be fired according to its scheduling, the scheduling of events when an event is enabled according to the corresponding stochastic clock structure, the update of old scheduled events, the cancellation of scheduled events as dictated by cancellation edges, and the update of the global clock.

Event choice. The scheduling of events imposes an order over the events to be fired according to the relative time that is randomly sampled from a cumulative distribution function. A strict ordering is defined over the set of scheduled events, by means of which timed events are ordered by its time component. The function `processChoice` splits the set of scheduled events into a set of minimal timed events, from which one will be chosen to be fired.

Scheduling of events. The scheduling of events is achieved by enriching the function `computeEvents` in order to create a set of events whose flow edge corresponds to an invocation edge.

When there is a random variable assigned to the flow edge `FE` in the corresponding stochastic clock structure, the function `computeEvents` creates a set of events that are scheduled for an absolute time that is equal to the actual time of the simulation plus the current value of the random variable associated with the flow edge `FE`. The case for cancellation edges is analogous.

In the current implementation, we do not support FES policies, which is left for future work.

Updating and cancelling scheduled events. Once an event is fired, the current state of the system under simulation may change. The function `refreshOld` checks that remaining enabled and scheduled events are still valid and updates their matches and resulting graph as explained in the previous section. These equations are extended to deal with timed events.

The function for cancelling events is also extended to deal with timed events. In particular, the absolute time of the cancelled event should be greater or equal than the current time plus the relative time the cancelling edge indicates.

Updating the global clock. When a scheduled event is fired, the global clock is updated with the minimum time elapse provided by the time component of the event.

9.4 Metrics

The function `recordData` (line 8 in Listing 6) allows us to gather information in the simulation state during the simulation for its analysis a posteriori. As an example, Listing 7 provides the implementation of the metric that calculates the utilization time of channels (see the constraint in Fig. 26).

For this purpose, the dynamic part of the simulation state is extended with a set `waitingTimes` of tuples `cwt(C, TT, B, T)`, where `C` is the object identifier of a channel (i.e. graph P of the constraint), `TT` is the duration of the set of maximal satisfaction intervals, `B` indicates if the channel contained messages when the previous timed event was fired, and `T` is the time when the last event occurred. Whenever a timed event is fired, the function `update` in Listing 7 updates this set of tuples computing the duration. The equation in line 2 is applied when the channel was busy before firing the last event (i.e. the constraint was satisfied at the same match before the direct derivation) and updates the duration value by means of the expression $TT + (T' - T)$, where T' is the current global clock value. The equation in line 7 is applied when the channel was not busy before firing the last event but it contains a message now, as indicated by the pattern in the left side of the equation (i.e. the constraint has started to be satisfied after the last direct derivation) and it sets the boolean flag `B` to true. The equation in line 13 deals with the case where the channel is empty by setting the boolean flag `B` to false in the tuple `cwt(C, TT, B, T)`. By the end of the execution, this set will contain the utilization time of each channel.

```

1 op update : CWTSet Model Time CWTSet -> CWTSet .
2 eq update( cwt(O1, TT, true, T) CWTS1,
3   <<< O2 : Message | at : O1, PS > OC >>,
4   T', CWTS2 ) = update( CWTS1,
5   <<< O2 : Message | at : O1, PS > OC >>,
6   T', cwt(O1, TT + (T' - T), true, T') CWTS2 ) .
7 eq update( cwt(O1, TT, false, T) CWTS1,
8   <<< O2 : Message | at : O1, PS > OC >>,
9   T', CWTS2 ) = update( CWTS1,
10  <<< O2 : Message | at : O1, PS > OC >>,

```

```

11 T', cwt(O1, TT, true, T') CWTS2 ) .
12 eq update( none, Model, T', CWTS2 ) = CWTS2 .
13 eq update( cwt(O1, TT, B:Bool, T) CWTS1,
14 Model, T', CWTS2 ) = update( CWTS1, Model, T',
15 cwt(O1, TT, false, T') CWTS2 ) [owise] .

```

Listing 7 Measuring the utilization time of channels.

Note that, in the implementation, durations are associated with matches implicitly by using pattern matching in the set of recursive equations that define the function `update`. This is not the case in the graph transformation formalism, where matches are explicitly managed as first-order citizens. From the implementation point of view, Maude allows for a more economic approach where only the data that is relevant for simulation analysis purposes is kept.

10 Related Work and Discussion

In this section we start by comparing our work with related approaches and finalize with a discussion of the advantages and limitations of our techniques.

10.1 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, [22] 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 [43], the authors develop a timed approach with the purpose of animating the execution of GT rules. Their rules are classified as internal or external events (the latter may be triggered by users) and the timing information is represented as additional attributes of the elements in the model. In [9], 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 meta-model) with timing elements for control purposes only.

In the second approach, [12] adapts concepts from timed Petri nets, so that rules are assigned a range, and rule executions are delayed with uniform probability in this range. The work of [23] takes concepts from stochastic Petri nets, so that rules are assigned a delay given by a negative exponential distribution. An important difference of these works with respect to ours is that, while time is assigned to rules in [12,23,45], we assign it to schedulings. Hence, our formalism distinguishes all possible occurrences of a rule as events, offering a more

refined control over the actions that manipulate the system state during simulation. This makes our approach flexible enough to model the other ones in a unified way, as we can model activities (interruptible or not) with a *start* and an *end* event, and hence activities may have an observable initiation (start event). In [45], 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 [6], elements in models can be assigned timed constructs like clocks or timers. The work of [36] provides a variety of high-level timed primitives, like periodic activities, and rules can manipulate the FES, mixing both control and data. An encoding of these primitives in Real-Time Maude is presented in [37], in which the state representation includes time and action objects metarepresenting the actions to be performed, and where, for example, the semantics of atomic rules with a duration (activities) is given by a pair of rewrite rules, one for *triggering* the activity, keeping a record of the actions to be performed, and another one to *realize* it, materializing the changes that were scheduled. In our approach, we deal with events explicitly, avoiding a temporary representation of actions in the state which have to be applied when a rule is realized. We provide a neat separation between data and control by extending graph grammars with invocation and cancelling edges between rules. At the conceptual level, this allows us to reuse GT theory to reason about conflicts between activities of different duration. At the implementation level, using Maude's reflection facilities, a flow grammar imposes a number of control constraints that are interpreted by the simulation engine. In addition, we also provide an event scheduling simulator that extends the untimed simulation scheme through a number of variability points as explained in Section 9.3. In this way, the simulation scheme is defined and implemented independently of the semantics of the rules.

With respect to the third approach, in [44], GT rules are embedded into the DEVS simulation formalism [47]. 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. In [13], the authors present an approach for the modelling and verification of time-dependent dynamic structures based on real-time statecharts, where operations with side effects are modelled as graph transformations. Models are extended with clocks, and rules may add or query clock instances using simple inequalities. The working scheme of such timed systems is conceptually similar to timed automata. While the approach is mainly used for *verification* of properties expressed as graphs, our proposal is directed to specify *simulations*. Hence, we incorporate

explicit event schedulings with arbitrary probability density functions and define a variety of metrics. Moreover, parallel processes may interact via cancelling edges.

Our work also relates to the models of computations proposed by the embedded systems and systems-on-chip communities [29]. However, although we follow the discrete-time model of computation, our approach is not based on modules (processes) and communication channels. Instead, our behavioural specifications are decoupled from the actual system state where they are executed, allowing its dynamic change. This is a distinguishing feature of our approach (and those based on GT [13,23,36,44]) with respect to approaches like that of Ptolemy II [26,16], where dynamic structure changes are difficult to model but they can be naturally expressed using GT rules. See the next subsection for a more detailed comparison with Ptolemy II.

There are many systems to perform visual discrete-event simulation [1,18,33,34,41]. Many of them are graphical front-ends for discrete-event languages. For example, the Arena tool is a graphical front-end for the SIMAN process interaction language [34]. This is a simulation language tailored to the manufacturing domain. Models are built interconnecting blocks that represent activities like queuing a transaction (QUEUE) or modelling a timed activity (DELAY). Still, simulation models use this general-purpose syntax instead of concepts of the domain, which makes models more difficult to build and understand by non-experts. Moreover, if the modeller needs a primitive not provided by the language, he may need to adapt his simulation model. Therefore, this approach is less suitable for MDE, where the use of DSMLs is fundamental. In contrast, our proposal gives modellers freedom to define the semantics of DSMLs by using GT rules to specify behaviour in a flexible way.

Finally, some works in the end-user development community are somehow close to our proposal. For instance, AgentSheets [35] is a tool for building agent-based simulations and games using a rule-based language.

10.2 Discussion

The work presented in this paper contributes to the current research state by proposing a new way to add time to GT, inspired by the event scheduling view of discrete-event simulation. Its distinguishing feature is that it does not add time to rules, but to events, using arbitrary probability density functions. This fine-grained formalism provides flexibility to model higher-level concepts like activities and processes. It also promotes extensibility of behaviours, as an existing grammar can be extended by adding new processes which may interact with the existing ones through invocation and cancelling edges. Extending a standard GT system is generally more difficult as each rule encodes its conditions for executability, so that adding new behaviour to a grammar implies adding new rules but also modifying the

existing ones. In our approach, the conditions for the executability of a rule can be placed in a different process which may cancel or start the execution of the rule.

Regarding the invocation scheme, we use invocation and cancellation edges to refine the concurrent semantics of graph grammars. In this way, we can pass the context of execution to the scheduled events. The advantage of this feature with respect to rule-based approaches like [12,13,22,23,36,45] is that we gain in efficiency, as matches are not sought from scratch but they are just completed starting from the received match. A more subtle difference with other languages for control flow [2,17,19], is that our invocation edges are enabling conditions for the application of rules, but the flow grammars are not to be interpreted as flow charts in traditional programming [3]. This feature greatly helps in modelling concurrency.

For example, in GReAT [2] a flow edge between two rules passes to the second rule all matches where the first rule was executed, resulting in a sequential processing of such matches (all matches of the first rule are processed before the matches of the second). In Fujaba's story diagrams [19] edges do not pass partial matches, but rules may use the same identifiers to match the same elements. Edges between rules move the execution control from one rule to the next, and only one rule owns the current execution point. In Henshin, different types of transformation units [17] (like independent, sequential or conditional) can be used to define the execution control flow of a grammar. Data dependencies between units and their subunits can be defined as well by passing partial matches.

In contrast, our basic mechanism are the invocation edges, which represent enabling conditions for a rule to be executed at a certain match, but does not imply a sequential processing. Hence, flow grammars do not specify a sequence of activities, but rule (event) dependencies, and moreover we may have several initial events to start different concurrent processes. A flow grammar does not imply a single point of execution, hence there is no "program counter" but a set of concurrently enabled rules, which can be picked for execution. Whereas the concurrent semantics of a flow chart is obtained by interleaving the activities in parallel flows that appear in fork clauses, in our flow grammars concurrency is explicitly characterized through dependencies between sets of events. This implies that our approach is more scalable when dealing with infinite server semantics, since we capture concurrency at a local level, avoiding the computation of the whole class of interleavings. Finally, in our approach we also consider cancellation of events, and we can control the level of parallelism through edge inscriptions (see Section 4.1).

Regarding the heterogeneity of models of computation, we discuss the similarities of our approach with Ptolemy II [26,16]. Ptolemy II provides a block diagram language [14] to coordinate, via wiring constructs,

the execution of synchronous, concurrent software modules that may be given as precompiled blocks specified in other languages. The execution model of a system specified in this language is a deterministic actor-based model, where the semantics of a system at a given point is provided by the least fixed point of the vector-valued function that defines the behaviour of the system in terms of the functions of the blocks and their connectivity. Ptolemy’s abstract semantics [26] enables a stepwise implementation of synchronous/reactive models, discrete-event models and continuous-time models. There is a clear analogy with our approach since our flow grammars can be used to simulate synchronous/reactive systems and scheduling grammars can be used to simulate discrete-event models and continuous-time models. This is possible because scheduling grammars are parameterised with respect to the probability density function, which can be defined over a discrete domain or over a continuous domain that provides the delay for an event. Discrete-event systems in Ptolemy II are based on the notion of super-dense time in which an actor is scheduled to be fired at a given tag (t, n) , where t is a timestamp and n is a natural number, so whenever there are several actors scheduled at the same timestamp t , the second component provides the order in which they have to be fired, achieving a deterministic behaviour. In our approach, events scheduled at the same timestamp are not ordered and the choice of the next event to be fired, among several instantaneous events, is non-deterministic in order to cover all possible behaviours. In Ptolemy II, pre-compiled blocks specified in different languages can be used in a system by encapsulating them in actors following a black-box philosophy. Our approach allows the domain expert/simulationist to exploit the domain-specific syntax of the corresponding model by using graphical syntax for defining both the system state and the behaviour of the system following a white-box philosophy. The black-box approach used in Ptolemy II also means that it is not possible to determine whether an actor breaks data precedences, whereas we have shown how to use GT theory to analyse this problem in our approach. Ptolemy II provides exact and heuristic algorithms to improve the efficiency of the execution of systems by exploiting their concurrent behaviour [14]. In our case, this effect is achieved by explicitly defining invocation edges that may carry contextual information. Hence, a domain expert can define heuristics in this way for the corresponding application domain. As stated above, the topology of systems in Ptolemy II is fixed at design time while we have shown how dynamic reconfiguration of systems can be simulated in our approach.

A crucial difference with other timed approaches to GT [12, 22, 23, 36, 45] is that we do not add time to the rules, but to events. This is consistent with the view that rules represent schemas of atomic events with no duration. As discussed in Section 6, adding time to rules

makes them have a duration, so that rules become activities with unobservable initiation, and only a final action given by the application of the rule. Adding time to events is therefore more *primitive*, and as illustrated by the example of Section 8.2 permits defining higher-level primitives like activities. Therefore, our approach can be used to express the semantics of other formalisms.

Our proposal is especially suited to complement MDE approaches, which explicitly define DSMLs and models thereof. In these approaches, it would be more time consuming to encode the semantics of models using an external simulation language. Instead, we propose a direct means to define the timed semantics of the DSMLs. Hence, we retain the best properties of both GT and discrete-event simulation worlds: on the one hand, we follow an MDE approach based on meta-models to define the syntax of the DSMLs, and use declarative GT rules to specify their semantics; on the other hand, we adopt the efficient time handling schemes of the event scheduling approach.

11 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. *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. We have also defined several ways to define and apply domain-specific metrics, and developed GT theory to detect errors in the scheduling of parallel matches. The visual nature of GT makes the approach suitable in application domains where simulation is used. We have also described an implementation in Maude, which shows the feasibility of our proposal.

In the future, we plan to work on analysis methods taken from both Event Graphs theory [42] and GT theory. In particular, the analysis of rule independence is of special interest for the optimisation of FES policies, e.g. to model non-interruptible activities. We would also like to extend our implementation in two ways. First, we are planning to provide a graphical front-end as well as facilities for the graphical representation of the obtained metrics. Second, we would like to use the verification capabilities of Maude, like model-checking and reachability analysis, to investigate properties of the simulation models. A usability study of this enhanced implementation is left for future work.

Acknowledgements. Work partially sponsored by the Spanish Ministry, under project “Go Lite” (TIN2011-24139) 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 previous versions of the paper.

References

1. AnyLogic. <http://www.xjtek.com/>.
2. D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. *ECEASST*, 1, 2006. See also <http://www.isis.vanderbilt.edu/tools/GReAT>.
3. M. Bohl and M. Rynn. *Tools For Structured and Object-Oriented Design*. Prentice Hall Press, 7th edition, 2007.
4. A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In *FASE'09*, volume 5503 of *LNCS*, pages 18–33. Springer, 2009.
5. A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22:269–296, 2010.
6. A. Boronat and P. C. Ölveczky. Formal real-time model transformations in MOMENT2. In *FASE'10*, volume 6013 of *LNCS*, pages 29–43. Springer, 2010.
7. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems, 2nd Ed.* Springer, 2008.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007. See also <http://maude.cs.uiuc.edu>.
9. J. de Lara. Meta-modelling and graph transformation for the simulation of systems. *Bulletin of the EATCS*, 81:180–194, 2003.
10. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.
11. J. de Lara, E. Guerra, A. Boronat, R. Heckel, and P. Torrini. Graph transformation for domain-specific discrete event time simulation. In *ICGT'10*, volume 6372 of *LNCS*, pages 266–281. Springer, 2010.
12. J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3–4):297–326, 2010.
13. T. Eckardt, C. Heinzemann, S. Henkler, M. Hirsch, C. Priesterjahn, and W. Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. *Computer Science - Research and Development*, in press:1–20, 2011.
14. S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.
15. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
16. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003.
17. C. Ermel, E. Biermann, J. Schmidt, and A. Warning. Visual modeling of controlled emf model transformation using henshin. *ECEASST*, 32, 2010.
18. ExtendSim. <http://www.extendsim.com>.
19. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *TAGT*, volume 1764 of *LNCS*, pages 296–309. Springer, 2000. See also <http://www.fujaba.de>.
20. G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, 2001.
21. L. Gönczy, M. Kovács, and D. Varró. Modeling and verification of reliable messaging by graph transformation systems. *ENTCS*, 175(4):37–50, 2007.
22. S. Gyapay, D. Varró, and R. Heckel. Graph transformation with time. *Fundam. Inform.*, 58(1):1–22, 2003.
23. R. Heckel, G. Lajos, and S. Menge. Stochastic graph transformation systems. *Fundam. Inform.*, 74(1):63–84, 2006.
24. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling. Enabling Full Code Generation*. Wiley-IEEE CS, 2008.
25. L. Lambers, H. Ehrig, and F. Orejas. Conflict detection for graph transformation with negative application conditions. In *ICGT'06*, volume 4178 of *LNCS*, pages 61–76. Springer, 2006.
26. E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, pages 114–123. ACM, 2007.
27. L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf. A visual control flow language for model transformation systems. In *IASTED Conf. on Software Engineering*, pages 194–199. IASTED/ACTA Press, 2006. See also <http://avalon.aut.bme.hu/~tihamer/research/vmts/>.
28. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.
29. 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.
30. M. Naeem, R. Heckel, F. Orejas, and F. Hermann. Incremental service composition based on partial matching of visual contracts. In *FASE'10*, volume 6013 of *LNCS*, pages 123–138. Springer, 2010.
31. R. E. Nance. A history of discrete event simulation programming languages. *SIGPLAN Not.*, 28:149–175, 1993.
32. OCL. <http://www.omg.org/spec/OCL/2.3/Beta2/>.
33. C. D. Pegden. SIMIO: a new simulation system based on intelligent objects. In *Winter Simulation Conference*, pages 2293–2300, 2007. See also <http://www.simio.com>.
34. C. D. Pegden and D. A. Davis. Arena: a SIMAN/cinema-based hierarchical modeling system. In *Winter Simulation Conference*, pages 390–399, 1992. See also <http://www.arenasimulation.com/>.
35. A. Repenning, A. Ioannidou, and J. Zola. AgentSheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), 2000. See also <http://www.agentsheets.com>.

36. 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.
37. J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In *WRLA*, volume 6381 of *LNCS*, pages 174–190. Springer, 2010.
38. J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *SLE'09*, volume 5452 of *LNCS*, pages 54–73. Springer, 2009.
39. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
40. T. J. Schriber. *Simulation Using GPSS*. John Wiley & Sons, 1974.
41. T. J. Schriber and D. T. Brunner. Inside discrete-event simulation software: How it works and why it matters. In *Winter Simulation Conference*, pages 151–165, 2010.
42. L. Schruben. Simulation modeling with event graphs. *Commun. ACM*, 26(11):957–963, 1983.
43. T. Strobl and M. Minas. Specifying and generating editing environments for interactive animated visual models. *ECEASST*, 29, 2010.
44. E. Syriani and H. Vangheluwe. Programmed graph rewriting with DEVS. In *AGTIVE'07*, volume 5088 of *LNCS*, pages 136–151. Springer, 2008.
45. P. Torrini, R. Heckel, I. Ráth, and G. Bergmann. Stochastic graph transformation with regions. *ECEASST*, 29, 2010.
46. J. G. Vaucher and P. Duval. A comparison of simulation event list algorithms. *Commun. ACM*, 18:223–230, April 1975.
47. B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation, 2nd Edition*. Academic Press, 2000.

A Detecting Errors in Parallel Schedulings

This appendix details the developed theory to detect errors in the scheduling of parallel matches. Similar to critical pairs [15], there are two kinds of conflicts: *delete-use* and *produce-forbid*. The first one, already discussed in Section 4.1, detects if some element in the passed parameters is not preserved by the rule execution. As all matches share such elements, the first rule application will destroy the rest of the scheduled matches. The second conflict (produce-forbid) detects if the rule adds elements that will violate some of the rule's NAC, invalidating the rest of the scheduled matches.

As a difference with standard critical-pairs analysis, we only want to indicate a conflict if it violates the parallel semantics, allowing only the first match to be executed. That is, we do not report *potential* conflicts but real ones, produced in the passed parameters M_{ij} .

Delete-Use. The left of Fig. 40 shows the condition for a delete-use conflict, which is the absence of a morphism $m_k: M_{ij} \rightarrow K_j$ s.t. (1) commutes. The right of

the same figure shows the first step in the rule application at match m_1 . This rule application does not destroy a previous match $m_2: L_j \rightarrow G$ (with $m_1 \circ m_l = m_2 \circ m_l$), if there exists a morphism $m'_2: L_j \rightarrow D$ s.t. $d \circ m'_2 = m_1$ (see the left of Fig. 41). However, as the right of Fig. 41 shows, such morphism cannot exist because, in (weak) adhesive HLR categories, pushouts where one of the given morphisms is injective (like $l: K_j \rightarrow L_j$) are also pullbacks [15]. Then, as we would have that $m_1 \circ m_l = m'_2 \circ m_l$, by the pullback universal property, there would exist a morphism $u: M_{ij} \rightarrow K_j$, obtaining a contradiction. Hence, m'_2 cannot exist.

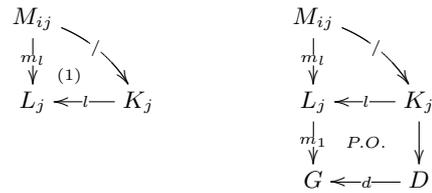


Fig. 40 Condition for delete-use scheduling conflict (left). First rule application step in delete-use scheduling conflict (right).

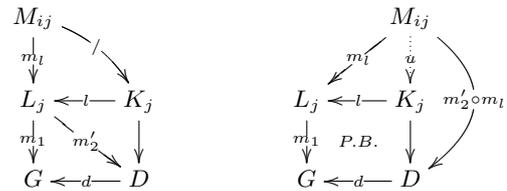


Fig. 41 A preserved match m_2 in delete-use conflict (left). Contradiction if match m_2 would exist (right).

Produce-Forbid. Fig. 42 shows the conditions for a produce-forbid conflict. This conflict exists if: (a) there exist a graph X and morphisms m_x, y s.t. the left square is a pushout, (b) there exist morphisms m_r and x s.t. (1) and (2) commute. The left pushout exists if the NAC N adds elements only to elements included in M_{ij} . m_r exists and (1) commutes if the rule preserves the elements in M_{ij} . x exists and (2) commutes if the rule adds the elements included in the NAC.

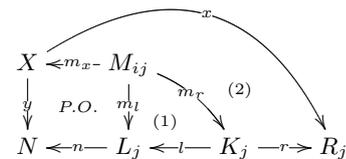


Fig. 42 Conditions for produce-forbid scheduling conflict.

We next proof that this condition induces a conflict as the rule cannot be applied twice at two matches sharing M_{ij} . Fig. 43 shows how, if the conditions of Fig. 42

are met, and the first rule application at match m_1 preserves the match $m_2: L_j \rightarrow G$ so that we have a morphism $m'_2: L_j \rightarrow D$, then the rule cannot be applied at $e \circ m'_2$ since there is a match $u: N \rightarrow H$ violating the NAC. Morphism u exists due to the pushout universal property, because we have $m_1^* \circ x \circ m_x = e \circ m'_2 \circ m_l$.

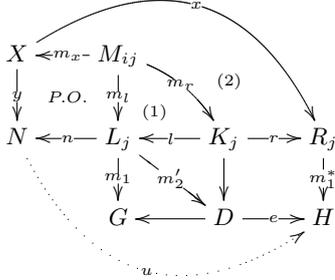


Fig. 43 Produce-forbid scheduling conflict.

B Encoding of the Flow Grammar for the Protocol Example

The Maude encoding of the rules in Fig. 8 for the protocol example is as follows:

```

1 rl [init] :
2   << < N : Node | isInit : true, PS > OC >> =>
3   << < N : Node | isInit : true, PS >
4   < freshOid() : Request | at : N, size : 256 > OC >> .
5
6 rl [end] :
7   << < N : Node | isInit : true, PS1 >
8   < M : Reply | at : N, PS2 > OC >> =>
9   << < N : Node | isInit : true, PS1 > OC >> .
10
11 crl [send] :
12   << < N : Node | PS1 > < M : Message | at : N, PS3 >
13   < C : Channel | in : N, PS2 > OC >> =>
14   << < N : Node | PS1 > < M : Message | at : C, PS3 >
15   < C : Channel | in : N, PS2 > OC >>
16   if nac("RequestInTerminal", NodeId(N) MessageId(M),
17   < N : Node | PS1 > < M : Message | at : N, PS3 >
18   < C : Channel | in : N, PS2 > OC)
19   /\ nac("ReplyInInitial", NodeId(N) MessageId(M),
20   < N : Node | PS1 > < M : Message | at : N, PS3 >
21   < C : Channel | in : N, PS2 > OC) .
22
23 ops NodeId MessageId ChannelId : Oid -> EnvVar .
24 eq nac("RequestInTerminal", NodeId(N) MessageId(M),
25 < N : Node | isFinal : true, PS1 >
26 < M : Request | at : N, PS2 > OC) = false .
27
28 eq nac("ReplyInInitial", NodeId(N) MessageId(M),
29 < N : Node | isInit : true, PS1 >
30 < M : Reply | at : N, PS2 > OC) = false .
31
32 rl [receive] :
33   << < N : Node | PS1 > < M : Message | at : C, PS3 >
34   < C : Channel | out : N, PS2 > OC >> =>
35   << < N : Node | PS1 > < M : Message | at : N, PS3 >
36   < C : Channel | out : N, PS2 > OC >> .
37
38 rl [reply] :
39   << < N : Node | isFinal : true, PS1 >
40   < M : Request | at : N, PS2 > OC >> =>
41   << < N : Node | isFinal : true, PS1 >
42   < M : Reply | at : N, PS2 > OC >> .
43
44 rl [lose] :
45   << < M : Message | at : C, PS2 >
46   < C : Channel | PS > OC >> =>

```

```

47   << < C : Channel | PS > OC >> .
48
49 crl [createConnection] :
50   << < N : Node | PS1 >
51   < C : Channel | in : O:Oid, PS2 > OC >> =>
52   << < N : Node | PS1 >
53   < C : Channel | in : N, PS2 > OC >>
54   if nac("notConnectedToAnotherNode",
55   NodeId(N) ChannelId(C),
56   < N : Node | PS1 >
57   < C : Channel | in : O:Oid, PS2 > OC)
58   /\
59   nac("notConnectedYet", NodeId(N) ChannelId(C),
60   < N : Node | PS1 >
61   < C : Channel | in : O:Oid, PS2 > OC) .
62
63 eq nac("notConnectedToAnotherNode",
64   NodeId(N) ChannelId(C),
65   < N : Node | PS1 > < C : Channel | PS2 >
66   < C2 : Channel | in : N, PS3 > OC)
67   = false .
68
69 eq nac("notConnectedYet", NodeId(N) ChannelId(C),
70   < N : Node | PS1 > < C : Channel | in : N, PS2 > OC)
71   = false .
72
73 rl [deleteConnection] :
74   << < N : Node | PS1 >
75   < C : Channel | in : N, PS2 > OC >> =>
76   << < N : Node | PS1 >
77   < C : Channel | in : 'null, PS2 > OC >> .
78
79 rl [channelCheck] :
80   << < C : Channel | PS > OC >> =>
81   << < C : Channel | PS > OC >> .

```

Listing 8 Maude encoding of the example rules.

Invocation and cancellation edges in the flow grammar in Fig. 10 are defined as constants IESet and CESet in our Maude encoding as follows:

```

1 op IESet : -> FlowEdgeSet .
2 eq IESet =
3   flowEdge('null, empty, 'init)
4   flowEdge('init, < 'N:Oid, 'N:Oid >, 'init)
5   flowEdge('init,
6   < 'N:Oid, 'N:Oid > < 'M:Oid, 'M:Oid >, 'send)
7   flowEdge('send,
8   < 'C:Oid, 'C:Oid > < 'M:Oid, 'M:Oid >, 'receive)
9   flowEdge('receive,
10  < 'N:Oid, 'N:Oid > < 'M:Oid, 'M:Oid >, 'send)
11  flowEdge('receive,
12  < 'N:Oid, 'N:Oid > < 'M:Oid, 'M:Oid >, 'reply)
13  flowEdge('receive,
14  < 'N:Oid, 'N:Oid > < 'M:Oid, 'M:Oid >, 'end)
15  flowEdge('receive, < 'N:Oid, 'N:Oid >, '
16  createConnection)
17  flowEdge('reply,
18  < 'N:Oid, 'N:Oid > < 'M:Oid, 'M:Oid >, 'send)
19  flowEdge('createConnection,
20  < 'C:Oid, 'C:Oid > < 'N:Oid, 'N:Oid >, 'send)
21  flowEdge('deleteConnection,
22  < 'C:Oid, 'C:Oid > < 'N:Oid, 'N:Oid >,
23  'deleteConnection)
24  flowEdge('null, empty, 'deleteConnection)
25  flowEdge('null, empty, 'channelCheck)
26  flowEdge('channelCheck, < 'C:Oid, 'C:Oid >, '
27  channelCheck)
28  flowEdge('channelCheck, < 'C:Oid, 'C:Oid >, 'lose) .
29
30 op CESet : -> FlowEdgeSet .
31 eq CESet = flowEdge('send,
32 < 'C:Oid, 'C:Oid > < 'N:Oid, 'N:Oid >,
33 'deleteConnection) .

```

Listing 9 Encoding of the Flow Grammar in Maude.