Goal-Oriented Requirements Animation

Hung Tran Van, Axel van Lamsweerde
Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
{tvh, avl}@info.ucl.ac.be

Philippe Massonet, Christophe Ponsard
Requirements Engineering Group
CETIC Research Center
B-6041 Charleroi (Belgium)
{phm, cp}@cetic.be

Abstract

Requirements engineers need to make sure that the requirements models and specifications they are building do accurately capture what stakeholders really want. Requirements animation has been recognized to be a promising approach to support this. The principle is to simulate an executable version of the requirements model and to visualize the simulation in some form appealling to stakeholders. Most animation tools available to date simulate operational models. Such models in general do not directly reflect the objectives, constraints and assumptions stated declaratively by stakeholders. It is also not possible to focus the animation on particular portions of a complex model relevant to some specific concern.

The paper describes a tool aimed at overcoming such limitations by animating goal-oriented requirements models. The tool automatically generates parallel state machines from goal operationalizations, instantiates those machines to specific instances created by users at animation time, executes them from concurrent events input by multiple users, monitors property violations at animation time, and visualizes concurrent simulations in terms of animated scenes in the domain.

1. Introduction

Animation is a well-established technique for checking whether software specifications meet the real intents and expectations of stakeholders. It consists in showing how an executable model of a software-based system behaves in response to external events and user inputs. Typically, an executable model is built from the software specifications; the software behavior is simulated by executing that model; the simulation is visualized on a textual or graphical model representation by highlighting the current model element being executed. Animation thus allows modellers to show the presence of problems, not their absence.

We will call *simulation* the execution of a model and *animation* the visualization of a simulation in some graphical form. Much work has been done in this area since the idea was originally proposed [2]. The contributions mainly differ by (a) how far the model is from the underlying requirements, (b) how far the visualization is from phenomena within the software environment, (c) how the simulation works (through direct execution of the model or preliminary translation to some executable form), and (d) how interactive and controlled the simulation can be.

For example, the model may be formulated in some prototyping language [1] and may even reuse program fragments [20]. An operational model may also be formulated at several levels of abstractions where interaction consists in asking whether specific behaviors can happen [3]. At a more abstract level, the model may be specified in equational sub-sets of state-based languages such as Z, VDM or B, and then either interpreted directly or translated into some logic or functional programming language – e.g., [16, 33, 37, 24, 13, 39]. As behavior is the primary focus of animation, many efforts have been devoted to explicit event-based descriptions [29] and state-machine models [11, 18, 23, 14, 38, 30, 31]; see also [36] for a comprehensive comparison of commercial tools emanating from this research. In particular, the animators described in [15, 38, 30] allow both control of the simulation through events in the world (such as pushing a control panel button) and visualization of the simulation through phenomena in the world (e.g., new values displayed on a plane altimeter). Other types of model may be animated as well; for example, the tool described in [17] animates specifications of the obligations and permissions of the various agents making the system.

Even though progress in this area is impressive the

current state of the art in *requirements* animation is limited in several respects. Such limitations provide the motivation for our work.

- *Animation of design behaviors:* many animation tools help exploring the *solution* space by animating design models of the "machine". As in [14, 17, 38], we are interested instead in exploring the problem space in the "world" [19, 34] in order to check whether a requirements specification is adequate with respect to the stakeholders' real needs.
- *Distance between the animated model and its underlying requirements:* most animation tools presuppose the availability of some executable model that correctly "implements" declarative requirements generally left implicit. Building an operational model from a declarative one may be a difficult, error-prone task (as illustrated in [6]); little support is provided for this.
- *Unfocussed animation:* while compositional animation is sometimes supported [30], we are not aware of tools able to animate particular model portions that are relevant to some specific concern or implement some underlying goal. Unfocussed animation may be a problem especially in the case of complex models.
- *Lack of property checking:* while most model checking tools allow traces to be animated during exploration, animators in general do not detect violations of desirable prperties on the fly *during* animation. Exceptions include [14, 17]; such tools however provide no support for suggesting what properties should be checked and how such properties are inter-related within a declarative model.
- *Lack of roundtrip animation:* animators that execute model translations in general provide no means for pointing out inadequacies back in the original model.

This paper describes an animation tool aimed at addressing the above limitations altogether. The key features of our animator are the following.

- The executable model being animated is a set of parallel state machines generated automatically from goal operationalizations. The latter are produced systematically from goal refinement graphs using semantics-preserving derivations [26].
- The animation is goal-oriented; it is focussed on the set of behaviors to achieve some goal(s) selected by the user from the goal model. Such goals define the *scope* of the animation. The scope may cover small or large portions of the behavior model dependent on whether the goals selected in the goal refinement graph are fine-grained or high-level, respectively. During model building, scoping allows incremental animation of partial models associated with specific goals; during analysis of entire models, it allows the animation to be restricted to some specific concern(s).
- The animator may detect violations of goals, domain properties or assumptions thanks to a fully reworked version of our runtime goal monitor [8]; the latter operates here at animation time, in full synchronization with the simulation, and no longer requires event traces to be recorded and analyzed.
- Stakeholders may decide on the configuration of instances of agents and entities to be animated by creating/deleting such instances at animation time.
- The animator visualizes simulation runs both in terms of UML state diagrams and animated scenes in the world (the latter visualization is based on the technology developed for LTSA [30]).
- Various forms of parallelism and interaction are supported. At the product level, animations of various domain objects may proceed in parallel (as in [30]). At the process level, multiple stakeholders may interact with the animator to input concurrent events (as in [17] but stakeholders may now be distributed over the internet). The animator may react to single events, replay pre-recorded scenarios, and proceed through traces forward or backwards.
- The animator is integrated in the FAUST formal analysis suite [35], a toolset that currently also includes a goal model checker; it is implemented in Java on top of the Objective*r* environment supporting the KAOS goal-oriented RE method [32, 22].

The paper is organized as follows. Section 2 summarizes some material on goal-oriented RE used in the sequel. Section 3 outlines the client-server architecture of our animator. Section 4 presents the server side by describing the techniques used to generate parallel state machines from goal operationalizations, instantiate them for simulation, and execute them. Section 5 briefly discusses the animation actors on the client side. Section 6 presents the animation watchdog whose role is to monitor property violations during the simulation. Section 7 describes the techniques used for synchronizing state machine executions with visualizations as animated statecharts and domain scenes.

A demo of the animator in action on the train control system used as a running example in this paper can be

downloaded at http://www.cetic.be/~faust/Animator.html. The system involves multiple trains moving along a circular single-track set of blocks with multiple stations, block signals, railroad crossing gates and cars.

2. Some Bits of Goal-Oriented RE

Our requirements models comprise four sub-models: a goal model, an object model, an agent model and an operation model; these models are elaborated methodically, see [22].

A *goal* is a prescriptive statement of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system. *Agents* are active components, such as humans, devices, legacy software or software-to-be components, that play some *role* towards goal satisfaction. Some agents thus define the software whereas the others define its environment. Goals may refer to services to be provided (functional goals) or to quality of service (non-functional goals). Unlike goals, *domain properties* are descriptive statements about the environment, such as physical laws, organizational norms or policies, etc.

Goals are organized in AND/OR *refinement-abstraction hierarchies* where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve fewer agents [4, 5]. In such structures, *AND-refinement* links relate a goal to a set of subgoals (called *refinement*) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links may relate a goal to a set of alternative refinements.

Goal refinement ends when every subgoal is *realizable* by some individual agent assigned to it, that is, expressible in terms of conditions that are monitorable and controllable by the agent [25]. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment.

Goals prescribe intended behaviors; they are optionally formalized in a real-time temporal logic [4]. Keywords such as Achieve, Avoid, Maintain are used to name goals according to the temporal behavior pattern they prescribe.

Fig. 1 shows a goal model fragment for a train control system. The leaf goal Maintain[DoorsClosedWhileMoving] may be annotated with the following temporal logic assertion stating that in every future state the train doors shall be closed when the train is moving:

$$\forall \text{ tr: Train : tr.Moving } \Rightarrow \text{ tr.doorsState = 'closed'}$$

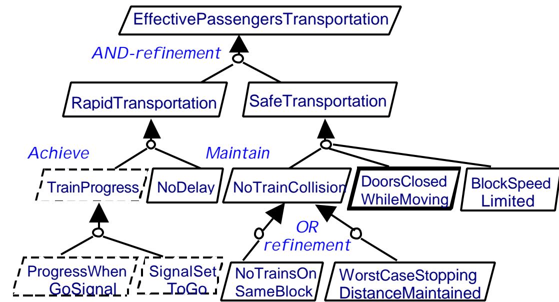

Figure 1: Portion of a goal graph for train control

Goals refer to *objects* which may be incrementally derived from goal specifications to produce a structural model of the system (represented by UML class diagrams). Objects have *states* defined by the values of their attributes and associations to other objects. They are passive (entities, associations, events) or active (agents). Agents are related together via their interface made of object attributes and associations they *monitor* and *control*, respectively [34]. In the above formalization of the goal DoorsClosedWhileMoving, Moving and doorsState are attributes of the Train entity declared in the object model. If the goal DoorsClosedWhileMoving is assigned to the TrainController agent, the latter must be able to monitor the attribute Moving and control the attribute doorsState of trains.

A goal specification prescribes a set of intended system behaviors, where a behavior is defined as a temporal sequence of system states.

Goals are *operationalized* into specifications of operations to achieve them [4, 26]. An *operation* is an input-output relation over objects; operation applications define state transitions along the behaviors prescribed by the goal model. In the specification of an operation, an important distinction is made between (descriptive) domain pre/postconditions and (prescriptive) pre-, post- and trigger conditions required for achieving some underlying goal(s):

- a pair *(domain precondition, domain postcondition)* captures the elementary state transitions defined by operation applications in the domain;
- a *required precondition* for some goal captures a permission to perform the operation *only if* the condition is true;
- a *required trigger condition* for some goal captures an obligation to perform the operation *if* the condition becomes true provided the domain precondition is

true (to produce consistent operation models, a required trigger condition on an operation implicitly implies the conjunction of its required preconditions);

- a *required postcondition* defines some additional condition that any application of the operation must establish in order to achieve the corresponding goal.

For example, the operation OpenDoor is among the operationalizations of the goal DoorsClosedWhileMoving; it may be partially specified as follows:

Operation OpenDoors
Input tr: Train; **Output** tr: Train/doorsState
DomPre tr.doorsState = 'closed'
DomPost tr.doorsState = 'open'
ReqPre for DoorsClosedWhileMoving :
$\neg$ tr.Moving
ReqTrig for RapidExit&Entrance:
AtStation (tr) $\wedge \neg$ tr.Moving

A *goal operationalization* is a set of such specifications. For example, the operationalization of our goal DoorsClosedWhileMoving includes specifications of all operations impacting on the satisfaction of this goal, that is, the *DomPre*, *DomPost*, *ReqPre*, *ReqTrig* and *ReqPost* conditions for the operations OpenDoors, CloseDoors, StartTrain and StopTrain; these operations impact on goal satisfaction as their specification captures changes of values of the state variables Moving and doorsState appearing in the goal specification.

We assume in this paper that operationalizations have been derived from goal specifications. For every goal specification pattern, inference rules are available for the formal derivation of a correct and complete set of *DomPre*, *DomPost*, *ReqPre*, *ReqTrig* and *ReqPost* conditions on operations to achieve the corresponding goal [26]. Goal monitoring at animation time may be used to check whether the operationalization is actually correct and complete, see Section 6.

3. Architecture of the FAUST Animator

Fig. 2 outlines the client-server architecture of our distributed multi-user animator. The *animation server* comprises two pairs of components: one for simulation of goal operationalizations and the other for monitoring of property violations. On the simulation side, a state machine compiler generates goal-based state machines (GSM) from the operation and object models retrieved through queries on the specification repository. The server's simulation engine executes those state machines in parallel during animation. On the monitoring side, another compiler generates claim machines from real-time temporal logic specifications of the goals, domain properties or assumptions to be monitored; the monitoring engine executes those claim machines during animation.

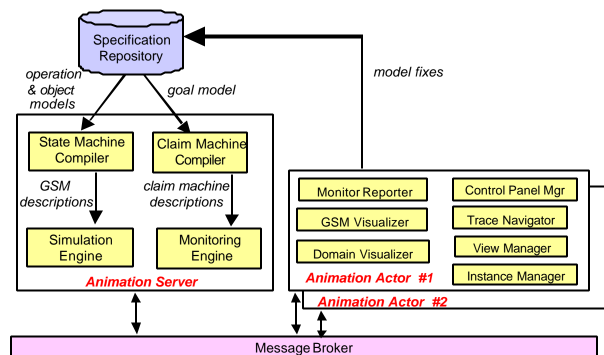

Figure 2: Animator architecture

The client, called *animation actor*, is integrated in the Objectiver front end [32]. It supports various features such as the selection of the goals from the goal model that define the animation scope, the creation of object instances to be animated within that scope, the selection of properties to be monitored during animation, the initialization of the animation, the visualization of the corresponding state machines during their execution, the visualization of the corresponding scenes in the world, event entry, and the saving, reload and replay of animation scenarios. Also on the client side, a monitor reporter logs all animation events about properties being monitored and signals any violation.

Following the event-based architectural style, the server and its clients communicate via a message broker; synchronization between them is ensured by event broadcast and notification protocols.

4. Simulating Goal Operationalizations

Within the server, goal operationalizations are first compiled into GSM state machines. The simulation engine then instantiates these machines and executes their instances through a GSM virtual machine based on a lazy behavior scheme [6] (see the *NextState* function below). The GSM virtual machine also provides services for the animation actors to interact with the simulation. This overall picture is now made further precise.

4.1. Goal-based state machines

Our GSM simulation formalism has been chosen to be (a) traceable back to the goal/operation specification language, (b) expressive enough to reflect KAOS models, (c) executable, (d) compositional and amenable to parallel animations, and (e) statechart-like to allow

visualizations of simple UML state diagrams [11].

Let V denote a set of state variables v of type $TY(v)$. A state s of v is defined by a function s: $V \rightarrow TY(V)$. Let $State(V)$ denote the set of all possible states of V. A *state machine M over a set of state variables V* is a transition system composed of the following items:

- a set $Var(M)$ of state variables such that $Var(M) \subseteq V$;
- a set $Init(M)$ of initial states ($Init(M) \subseteq State(Var(M))$);
- a set $Trans(M)$ of transitions each taking the form of a tuple

  $$\langle src, Grd, ev, Trig, Oe, tgt \rangle$$

  where src and tgt are the transition's source and target states, respectively, Grd is a guard condition, ev is a triggering event, $Trig$ is a trigger condition and Oe is a set of output events.

Let G denote a set of goals and $OP(G)$ denote an operationalization of the goals in G. A *GSM state machine with intentional scope G* is a state machine over the set of variables appearing in $OP(G)$.

GSM machines may be composed to run in parallel. Inter-machine communication and synchronization is achieved through event broadcast.

4.2. Generating GSMs from goal operationalizations

The generation rules used for mapping a goal operationalization $OP(G)$ to GSM machines with scope G are based on the semantics of goal operationalization [26]. For lack of space we only mention the main ones here.

- *Rule 1*: Every entity or agent from the object model referenced in $OP(G)$ is mapped to a GSM machine.
- *Rule 2*: For each such entity/agent, we call *behavioral attribute* (or *behavioral association*) any attribute (association) whose state is specified in $OP(G)$ to change by application of some operation; a behavioral attribute/association defines a state variable local to the associated GSM (in case of an attribute) or shared with other GSMs (in case of an association). Every behavioral attribute or association of an entity/agent referenced in $OP(G)$ is mapped to a concurrent substate of the corresponding GSM.
- *Rule 3*: Every non-behavioral attribute/association referenced in $OP(G)$ is mapped to an internal variable of the corresponding GSM.
- *Rule 4*: Every operation op specified in $OP(G)$ is mapped to a state transition in the GSM(s) associated with its codomain; this state transition is characterized by the following items:
  - a source state defined by op's *DomPre*,
  - a target state defined by op's *DomPost* ∧ *ReqPost*,
  - a triggering event labeled by op's name
  - a guard defined by op's *ReqPre*,
  - a trigger condition defined by op's *ReqTrig*.

Fig. 3 illustrates the use of these GSM generation rules for the train control example introduced in Section 2. The arrow there indicates generation of the right-hand part (a statechart fragment) from the left-hand part (a portion of the goal, object, and operation models represented here in textual format). The attributes *doorsState* and *Moving* of the *Train* entity from the object model are seen to be behavioral (see the specification of operations *OpenDoors* and *StartTrain*), and result in concurrent substates of the statechart associated with the *Train* entity. The attribute *Speed* appears from the specification to be non-behavioral and will result in an internal variable of that state machine (not shown in Fig. 3). The state machine transitions in the right-hand part of Fig. 3 are derived from operations referencing *Train* in their Output clause; they are annotated by information derived according to Rule 4 from the specifications of the corresponding operations. Note that the generated state machines are "first-order" in that they are parameterized on instance variables. Other information may be generated such as output events associated with a transition; this is not detailed here.

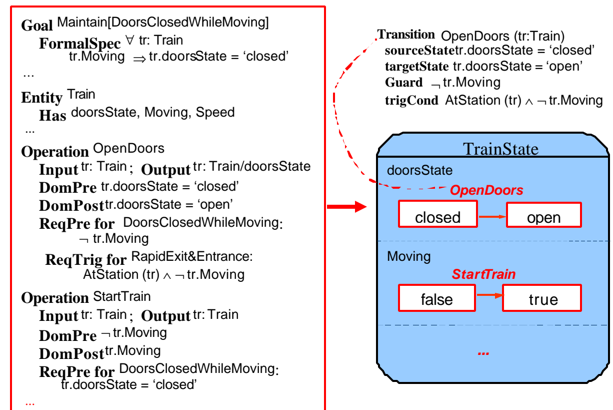

Figure 3: From goal operationalizations to GSM machines

4.3. Instantiating and executing GSM machines

The GSM machines generated within some goal scope at compile time are instantiated at animation time according to the entity/agent instances created by animation actors. First-order assertions on variables are propositionalized on the fly according to these instances. The GSM instances are executed by the GSM virtual machine using the *NextState* function.

The *NextState* function controls the traversal of GSM instances under a *lazy behavior* scheme, that is, a

transition is fired only when it is really obliged to do so – because a triggering event occurs or one of the transition's required trigger conditions becomes true. This choice has been made to remove agent non-determinism [6]. The simultaneous occurrence of multiple events on different transitions, corresponding to triggering events or trigger conditions becoming true, results in concurrent activation of these transitions; the *NextState* function controls such parallelism. It also handles clock ticking for the animator's stepwise execution.

We outline the definition of the *NextState* function without providing all details for lack of space. The definition relies on the following quantities. Let *T* denote a set of transitions, *CS* a global configuration of states and *E* a set of events.

- *Triggered*(*E*) is the set of transitions that are triggered by an event *e* in *E*;
- *Obliged*(*CS*) is the set of transitions that are triggered by the satisfaction of trigger conditions in configuration *CS*;
- *Enabled*(*T*, *CS*) is the set of transitions in *T* whose source state is in *CS*;
- *Permitted*(*T*, *CS*) is the set of transitions whose guard is satisfied in configuration *CS*;
- *Consistent*(*T*) is the maximal set of transitions in *T* whose target states are not conflicting with each other (that is, their instantiated target state predicates are not inconsistent with each other);
- *EvGenerated*(*T*) is the set of output events generated when all transitions in *T* are executed;
- *newConfig*(*T*, *CS*) is the new global configuration produced by executing all transitions in *T* from configuration CS.

The lazy behavior *NextState* function is then defined as follows.

```
NextState (E, CS):
  T =  Triggered (E) ∪ Obliged (CS) ;
  T' =  Enabled (T, CS) ∩ Permitted (T, CS) ∩ Consistent (T);
  if T' ≠ ∅ then
        E' = EvGenerated (T');
        CS' = newConfig (T', CS);
        NextState (E', CS');
  endif
  return
```

According to the *NextState* function, two transitions found to be inconsistent with each other are both discarded; there will thus never be multiple subsets of transitions with nonconflicting target states. A more liberal strategy of keeping one of them while discarding the other could be adopted at the price of some arbitrariness on the selection of which transition to keep [27]; in cases where our strategy is felt too restrictive the user of the animator may control a more liberal, non-arbitrary selection by moving the animation one step back and removing only one of the conflicting transitions.

5. Interacting with the Simulation

The animation actor in Fig. 2 is a collection of services allowing users to access the animation server over the net and interact with the model being executed on the server. Communication and synchronization is based on an event send/notify mechanism (see Section 3). This section outlines some of the services provided by the actor.

To simulate GSMs within some goal scope, the participating actors of the animation session need to define an instantiation scheme, e.g., two instances *tr1*, *tr2* for the *Train* entity. This is done through the actor's *Instance Manager*.

Each animation user (actor) may have some specific view on the system; for example, the view of a train conductor might be restricted to specific attributes and associations relevant to her; the attribute gateState of the RailroadCrossing entity would be relevant in that view whereas the attribute Speed of the Car entity might not. This corresponds to the view concept in KAOS [21] and is supported by the *View Manager*. When the scope associated with the generated GSMs comprises several goals, the View Manager may also be used to restrict the focus of the animation further on some specific goal within that scope.

Thanks to the *Trace Navigator*, users may also walk through animation scenarios forward or backwards, and save them as scenarios for later replay.

Advanced users may also edit commands in a *Simulation Command Language* (SCL) and send them to the server. SCL is a script language that can be used, e.g., to create scenarios to be executed by the server, save such scenarios for later rerun, etc. SCL primitives include SET or GET the current value of some state variable, GO to launch multiple events in parallel, and trace rerun primitives such as GONEXT, GOBACK, RESET, RUN. For example, the following SCL script expresses a scenario where train instance #1 enters a station block, stops at station #1, open doors and then tries to start without closing doors:

```
[LeaveBlock(vh:=Train@1,bl:=Block@3)]GO
[EnterBlock(vh:=Train@1,bl:=StationBlock@1)]GO
```

```
[Stop(vh:=Train@1)]GO
[OpenDoors(tr:=Train@1)]GO
[Start(vh:=Train@1)]GO
```

The end-user, of course, does not use SCL; she submits input events and provides animation scenarios by interaction through input event panes or domain-specific control panels.

The animation actor provides other services for, e.g., control panel management, and an API for interfacing with the GSM simulator, monitor, and visualization engines.

6. The Animation Watchdog: Monitoring Property Violations at Animation Time

During animation of a requirements model, the user may want to know whether some specific properties are being violated. Our animation watchdog accepts a list of properties to be monitored and issues a warning when a property from that list is being violated (the warning is specific to that property). The watchdog is intended mainly for two different types of use:

- to debug incorrect goal operationalizations, when the property being monitored is a requirement (as defined in Section 2), by pointing out inadequate pre-, trigger-, or post-conditions; or, similarly, to debug incorrect goal refinements, when the property being monitored is a coarser-grained goal;
- to monitor whether the history of environmental events input to the animator conforms to assumptions on the environment's behavior, when the property being monitored is an expectation (as defined in Section 2) specified in the goal model or a domain property or hypothesis specified in the object model.

Our watchdog is a fully reworked version of a general-purpose monitoring tool we built before for goal/obstacle monitoring [8]. The current version is more efficient as it monitors traces containing relevant events only; such traces are handled on the fly without need for trace storage and analysis.

The idea is to represent temporal logic specifications by claim machines (CM) and execute such machines in parallel and synchronously with our GSM machines in order to detect possible mismatches at some point during their parallel execution. Our CM machines have to meet the following requirements.

- Accept the KAOS goal specification formalism, that is, a full first-order linear temporal logic equipped with real-time constructs (see, e.g., [21]).
- Handle finite-length traces (as observation time is always bounded), in chronological order and without backtracking.
- To ensure cheap monitoring, log and maintain a minimal amount of relevant information for trace checking; avoid explicit state polling; be event-oriented.

Trace checking algorithms are generally based on variants of the tableau algorithm and generate deterministic finite state automata [28, 9, 7, 12]; they meet most of our requirements except for quantifiers and real-time constraints. We therefore designed an extension based on deterministic time automata with some specific instantiation mechanism for their execution. We just sketch the idea here; technical details are outside the scope of this paper.

A *claim machine* (CM) is a finite automaton representing a first-order, real-time, linear temporal logic assertion as follows.

- A CM *state* is labelled with an assertion representing the obligation to be fulfilled on the rest of the trace. It may have an internal clock set to some initial value when entering the state and decreased while staying in that state.
- A CM *transition* is labelled with an assertion representing the guard condition for the transition to be activated. The disjunction of transition guards from a given state must yield **true** while their pairwise conjunction must yield **false**. CM states in general have a cycling transition (exceptions include the **false** state and states associated with the temporal "next" or "previous" operators).

Fig. 4 shows the claim machine for our goal DoorsClosedWhileMoving (the outer universal quantification is left implicit). The automaton simply loops in the invariant state unless it gets broken.

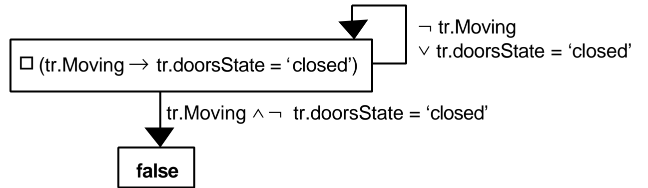

Figure 4: Claim machine for *DoorsClosedWhileMoving*

Consider now the real-time goal Achieve [CrossingGateClosedWhenTrainApproaching], specified as follows:

$\forall$ tr: Train, cb: CrossingBlock

On (tr, prev(cb)) $\Rightarrow \Diamond_{=d}$ cb.gateState = 'closed'

where *On* is an association between trains and blocks and *prev* is a function yielding the previous block. The

claim machine for this goal is shown in Fig. 5. It has an initial state waiting for a train to be on a block preceding a crossing block. The intermediate state is clocked and waits either for gate closing (in which case it goes back to the initial state) or for clock expiration (in which case a violation is detected).

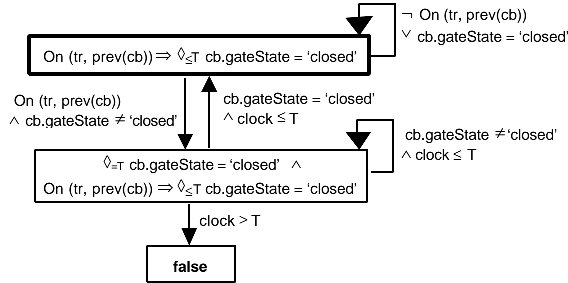

Figure 5: Claim machine for *GateClosedWhenTrainApproaching*

Efficient deployment of several claim machines is achieved using a transition-based encoding of time automata that avoids explicit state representation – as used in some intrusion detection systems that are able to tackle large amount of data in real-time [10].

The generation process takes two other dynamic issues into account for greater efficiency:

- the reconstruction of state information based on observed transitions (e.g., from a transition guard @P, the predicate P is inferred on the target state),
- the progressive instantiation of parameterized automata to handle quantification. For example, an automaton instance for the goal DoorsClosedWhileMoving will dynamically get started the first time an event referencing a new train is observed. For the goal CrossingGateClosedWhenTrainApproaching, an automaton will be generated for each occurring combination of train/crossing events.

The compiled claim machines are thus passed to the *Monitoring Engine* (see Fig. 2) where they are dynamically instantiated according to the entity/agent instances created by the animation actors at animation time. The instantiated CMs are then run concurrently with the GSM instances; they keep synchronized with the latter by listening to GSM transition events. Property violation is notified when a CM **false** state is reached.

7. Synchronizing Simulations and Domain Visualizations

Simulations are easier to follow by stakeholders and analysts when convenient graphical visualizations are provided. We first present the visualizations supported by our animator and then discuss the key issue of synchronizing multiple visualizations with GSM executions in a consistent way.

7.1. Visualizing simulations

Two types of graphical representation of GSM runs are provided:

- animated UML state diagrams;
- animated scenes in the world.

Simulation visualization is made flexible by letting the user select the appropriate combination of GSMs to be displayed.

Fig. 6 shows three state diagrams for the Train instance Train#1, composed in parallel, in which the current state is highlighted for each of them. These concurrent substates correspond to Train#1's *On* association, *doorsState* attribute and *Moving* attribute, respectively. The state assertions and transition guards may be viewed just by dragging the cursor over the corresponding states and transitions, respectively.

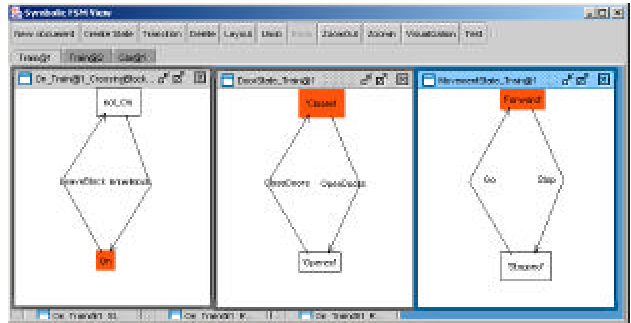

Figure 6: Visualization of Train#1's concurrent substates

The two lower right images in Fig. 7 are snapshots of corresponding scenes in the world that in particular exhibit the state of the railroad crossing and train doors, respectively.

7.2. Keeping simulations and visualizations consistent

To keep our multiple visualizations synchronized with GSM executions, we could have defined and then hardcoded the necessary mapping between them for each model considered. This turns out to be tedious and error-prone. In order to avoid such manual hardcoding, we have developed a tool for interactive specification and generation of mapping/synchronization schemes that ensure consistency.

Back to Fig. 7, a mapping/synchronization snapshot is suggested between GSM instances Train#1, Gate#1 and

Car#1 on the left part and their visual counterparts on the right part, namely, concurrent substates of a UML statechart (upper right part) and two domain-specific visualizations (lower right part): a bird's eye view of a railroad crossing and a view of the doors of a train car. Note that those different snapshot representations are consistent; the train is moving with its doors closed, the railroad crossing gate is closed, and the car is stopped and waiting.

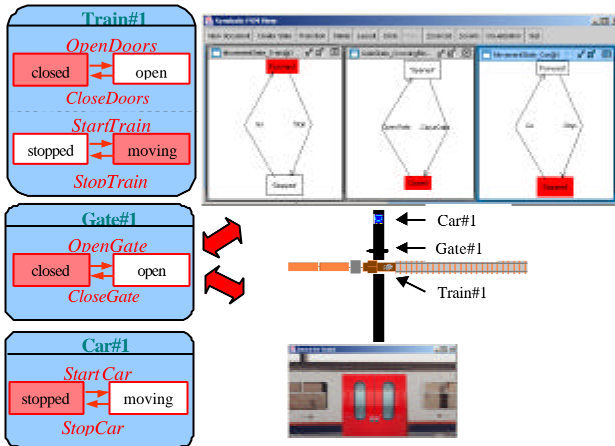

Figure 7: Synchronizing animations

At *scene specification* time, the definition of a mapping requires some domain-specific *mapping rules* to be specified between each GSM entity/agent, state and transition on the one hand and their visual counterpart in the world on the other hand. Such mapping rules have to be defined both at the *class* and *instance* levels as suggested in Fig. 8. For example, the Train GSM class with openDoors and closeDoors transitions needs to be instantiated before a complete mapping may be defined for animating Train#1's doors opening and closing. Train instance Train#1 needs to be mapped to a specific DoorsAnimation#1 so that openDoors/closeDoors transitions are mapped to animated doors opening/closing of *this* train instance.

At *scene animation* time, an event-based integration mechanism is provided for synchronizing GSM executions with their visual counterpart. The mapping tool listens to simulation events and dispatches them to each visual animation that has registered for them. These events are then translated into corresponding visual actions according to the specified mapping rules (see lower part of Fig.8 from right to left). To keep the visualizations synchronized with the simulation, each animated transition must be synchronized with the corresponding transition in the simulation; the latter is instantaneous whereas the former have some duration. The mapping rules must thus also involve events indicating the end of an animated transition for every simulation transition. The mapping tool listens to such events; any new simulation transition is blocked until the end events have been received from all animated transitions. This achieves the required consistency between GSM simulations and their animations. In addition, a compatible initial state must be provided for every GSM instance and its various visualizations so that they all start in states consistent with each other.

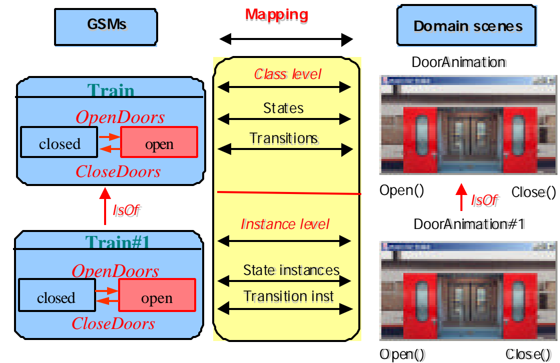

Figure 8: Mapping GSMs to visualisations

7.3. Controlling simulations from environment input

Our mapping tool has been designed to listen also to visual events in the environment (e.g., pressing a button to open Train#1's doors) and translate them into transition requests for the simulation engine using contextual information (e.g., the doors are closed).

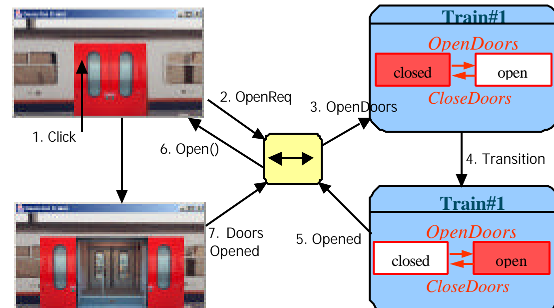

Figure 9: Interacting with world visualizations

Fig. 9 illustrates this; see steps 1-7 from upper left part to right. Clicking on closed doors of Train#1 (step 1) sends an openDoor request captured by the mapping tool (step 2), resulting in an OpenDoors transition request on Train#1 (step 3), sent to the simulation engine in the form of a SCL script. As explained in Section 4, the *NextState* function is then applied (step 4); in case of transition firing the simulator produces an event indicating that Train#1's GSM instance has made the

transition to the Open state (step 5). The latter event is captured by the mapping tool which translates it into a call to the Open() command of the corresponding world visualization (step 6). When the visual transition is finished, an event is sent indicating that the doors are open (step 7).

To make our mapping/synchronization tool as independent of specific technologies as possible, we have defined a simple meta-model of graphical animation toolkits based on notions such as animated graphical classes, static scenery elements, commands and events. This model needs to be specialized for some specific graphical technology to be used. Our current implementation is based on SceneBeans [30].

7.4. Instantiating domain visualizations with the mapping

As seen in Section 5, our animator makes it possible to create multiple entity/agent instances dynamically at simulation time. This feature should ideally be propagated in world scenes, e.g., to explore animations with 2 train instances and 4 blocks in the domain first and then with 3 train instances and 5 blocks next. This turns out to be non-trivial with technologies such as SceneBeans as it requires dynamic composition of graphical animations from parts *after* the graphical mapping and the number of instances have been defined. This is one of the visualizer's features we are still working on.

8. Conclusion

The contribution of our requirements animator lies in the unique combination of a number of distinguishing features, namely,

- the generation of parallel state machines from goal operationalizations to reduce the gap between stakeholders' requirements, often stated declaratively, and an operational model whose construction may be far from obvious;
- the goal-oriented animation of those portions of a behavior model that achieve some goal(s) selected by the user from the goal model – such focussed animation is especially helpful for incremental model building and for analysis of complex models;
- the ability to monitor, at animation time, violations of goals, requirements on the software, expectations on the environments, and domain properties;
- the ability to define configurations of simulated objects dynamically at animation time;
- the visualization of animations as world scenes within a multi-user, distributed environment.

Our animator is used typically downstream to its companion goal model checker; the latter checks for correctness of goal refinements and operationalizations, and generates counter-example scenarios in case of incorrectness [35]. Both tools are built on top of the Objectiver goal-oriented RE environment that supports model elaboration through a graphical editor and model analysis through queries, traceability management, and generation of the requirements document from the model.

While we have extensive industrial experience with Objectiver for several years, our experience in using the animator has been limited to case studies so far. (The one used in this paper is a composition of several non-trivial benchmarks used in the literature.) Note that scaleability is a built-in feature of our animator as (a) goal scoping restricts the animation to specific portions of the behavior model, (b) such portions can still be restricted by specifying relevant views that reduce the number of state variables, and (c) the animation involves a restricted number of object instances selected by the user.

Our short-term plans include extensions to our current domain visualizer to support dynamic domain configurations and dynamic mapping to "interesting" initial states in the domain. In parallel, we expect to get feedback fairly soon from industrial projects to direct future enhancements. Such feedback will in particular help us assess which tool among the animator and the goal model checker is better in what context.

Acknowledgement. The work reported herein was partially supported by the European Union and the Regional Government of Wallonia (FAUST project, Objective 1 programme, RW Conv. n°EP1A12030000012 -130023).

References

[1] B. Auernheimer and R. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems", *IEEE Trans. Software Engineering* Vol. 12 No. 9, Sept. 1986.

[2] R.M. Balzer, N.M. Goldman, and D.S. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM SIG-SOFT Softw. Eng. Notes* Vol. 7 No. 5, Dec. 1982, 3-16.

[3] K. M. Benner et al, "Utilizing Scenarios in the Software Development Process", *Information System Development Process*, Elsevier Science, 1993, 117-134.

[4] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[5] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 – 4th ACM Symp. on Foundations of Software Engineering*, Oct. 1996, 179-190.

[6] R. De Landtsheer, E. Letier and A.van Lamsweerde, "Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models", *Proc. RE'03 – 2nd Intl. Joint Conf. on Requirements Engineering*, Sept. 2003. Expanded version in *Requirements Engineering Journal*, 2004.

[7] L. K. Dillon and Y. S. Ramakrishna, "Generating Oracles from Your Favorite Temporal Logic Specifications", *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering*, San Francisco, October 1996, pp. 106-117.

[8] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th Intl. Workshop on Software Specification and Design*, IEEE CS Press, 1998.

[9] M. Felder and A. Morzenti, "Validating Real-Time Systems by History-Checking TRIO Specifications", *ACM TOSEM*, Vol.3, No.4, October 1994.

[10] N. Habra, B. Le Charlier, A. Mounji, I. Mathieu, "ASAX: Software Architecture and Rule-base Language for Universal Audit Trail Analysis", *Proc. 2nd European Symp. on Research in Computer Security (ESORICS)*, Nov. 1992.

[11] D. Harel et al, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, Vol. 16 No. 4, April 1990, 403-414.

[12] K. Havelund and G. Rosu. "Synthesizing monitors for safety properties", *Proceedings of TACAS*, volume 2280, pages 342--356. Springer-Verlag, April 2002.

[13] D. Hazel, P. Strooper, O. Traynor, "Requirements Engineering and Verification using Specification Animation", *Proc. Automated Software Engineering Conf.*, 1998, 302-305.

[14] C. Heitmeyer, J. Kirby, and B. Labaw. "Tools for Formal Specification, Verification, and Validation of Requirements", *Proc. COMPASS '97*, June 1997, Gaithersburg, MD.

[15] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements", *Proc. CAV'98*, Vancouver, 1998.

[16] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods, and VDM*. Addison-Wesley, 1988.

[17] P. Heymans and E. Dubois, "Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements", *Requirements Engineering Journal*, Vol. 3 No. 3-4, 1998, 202-218.

[18] G. Holzman, "The Model Checker SPIN", *IEEE Trans. on Software Engineering* Vol. 23 No. 5, May 1997, 279-295.

[19] M. Jackson, *Software Requirements & Specifications - A Lexicon of Practice, Principles and Prejudices*. ACM Press, Addison-Wesley, 1995.

[20] B. Kramer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Trans. Software Engineering* 19(5), May 1993, 453-477.

[21] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering,* November 1998.

[22] A. van Lamsweerde , "Goal-Oriented Requirements Engineering: A Guided Tour", *Proc. RE'01 - 5th Intl. Symp. Requirements Engineering*, August 2001, pp. 249-263.

[23] K.G. Larsen, P. Petersson and W. Yi, "UPPAAL in a Nutshell", *Intl. Journal on Software Tools for Technology Transfer*, Vol. 1, Springer-Verlag, 1997, 134-152.

[24] Y. Ledru, "Specification and Animation of a Bank Transfer using KIDS/VDM", *Automated Software Engineering* Vol. 4 No.1, January 1997, 33 – 51.

[25] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, May 2002.

[26] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[27] N.G. Leveson, M. Heimdahl, H. Hildreth and J.D. Reese, "Requirements Specification for Process-Control Systems", *IEEE Trans. Software Engineering* 20(9), Sept. 1994.

[28] U. W. Lipeck, "Transformation of Dynamic Integrity Constraints into Transaction Specifications", *Theoretical Computer Science* Vol. 76, 1990, 115 - 142.

[29] D.C. Luckham et al, "Specification and Analysis of System Architectures Using RAPIDE", *IEEE Transactions on Software Engineering* Vol. 21 No. 4, April 1995.

[30] J. Magee, N. Pryce, D. Giannakopoulou and J. Kramer, "Graphical Animation of Behavior Models", *Proc. ICSE'2000: 22nd Intl. Conf. on Software Engineering*, Limerick, May 2000, 499-508.

[31] W. E. McUmber and B.H.C. Cheng, "A Generic Framework for Formalizing UML", *Proc. ICSE01: Intl. Conf. on Software Engineering*, May 2001, Toronto.

[32] http://www.objectiver.com.

[33] G. O'Neill, Automatic Translation of VDM into Standard ML Programs", *The Computer Journal*, March 1992.

[34] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming,* Vol. 25, 1995, 41-61.

[35] A. Rifaut, P. Massonet, J.F. Molderez, C. Ponsard, P. Stadnik, H. Tran Van and A. van Lamsweerde, "FAUST: Formal Analysis of Goal-Oriented Requirements Using Specification Tools", *Proc. RE'03,* Sept. 2003, 350.

[36] R. Schmid, J. Ryser, S. Berner, M. Glinz, R. Reutemann and E. Fahr, *A Survey of Simulation Tools for Requirements Engineering*. Special Interest Group on Requirements Engineering, German Informatics Society (GI), Aug. 2000.

[37] J. Siddiqi, I. Morrey, C. Roast and M. Ozcan, "Towards Quality Requirements via Animated Formal Specifications", *Annals of Software Engineering*, Vol. 3, 1997.

[38] J.M. Thompson, M.E. Heimdahl, and S.P. Miller, "Specification-Based Prototyping for Embedded Systems", *Proc. ESEC/FSE'99*, LNCS 1687, Springer, Sept. 1999, 163-179.

[39] M. Utting, Formal Methods Links: Animation of Formal Specifications, http:// www.cs.waikato.ac.nz/~marku/ formalmethods.html#animation.