Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 133 (2005) 237–254
www.elsevier.com/locate/entcs

Early Verification and Validation of Mission Critical Systems

C. Ponsard, P. Massonet, A. Rifaut and J.F. Molderez [1]

CETIC Research Center, Charleroi, Belgium

A. van Lamsweerde and H. Tran Van [2]

Université catholique de Louvain, Department of Computer Science, Belgium

Abstract

Our world is increasingly relying on complex software and systems. In a growing number of fields such as transportation, finance, telecommunications, medical devices, they now play a critical role and require high assurance. To achieve this, it is imperative to produce high quality requirements. The KAOS goal-oriented requirements engineering methodology provides a rich framework for requirements elicitation and management of such systems.
This paper demonstrates the practical industrial application of that methodology. The non-critical parts are modelled semi-formally using a graphical language for goal-oriented requirements engineering. When and where needed (ie. for critical parts of a system) the model can be specified at formal level using a real-time temporal logic. That formal level seamlessly extends the semi-formal level which can also help hide the formality for the non-specialist.
To ensure at an early stage that the right system is being built and that the requirements model is right, validation and verification tools are applied on that model. Early verification checks help to discover missing requirements, overlooked assumptions or incorrect goal refinements. State machines generation from operations provides an executable model useful for validation purposes or for deriving an initial design. Acceptance test cases and runtime behavior monitors can also be derived from the model.
The process is supported by an integrated toolbox implementing the above tools by a roundtrip mapping of KAOS requirements level notations to the languages of formal technology tools such as model-checkers, SAT engines or constraint solvers. A graphical visualization framework also significantly helps validation using domain-based representations.

Keywords: Requirements Engineering, Goal-orientation, Early Verification, Validation, Animation, Monitoring, Acceptance Tests

doi:10.1016/j.entcs.2004.08.067

1 Introduction

Complex software and systems are pervasive in today's world. In a growing number of areas they come to play a critical role as their incorrect behavior may lead to catastrophic loss in terms of cost, damage to the environment, or even human life.

To produce highly reliable systems it is recommended to use formal methods. They refer to mathematically based languages, techniques and tools for specifying and verifying such systems. Even though there are some success stories, formal methods have not yet gained wide industrial adoption - the main obstacles being the significant investment from learning a difficult technology and often a psychological resistance to mathematics. To overcome those problems, alternative approaches are now being explored such as lightweight formal methods [4] and invisible formal methods [17]. The former is a targeted application limited in scope and analysis to reach relevant conclusions at a minimal cost. The latter aims at providing sufficiently convenient, powerful, and useful technologies for practitioners to adopt them willingly.

To achieve high assurance, a number of studies have also stressed the importance of the requirements phase [5][6]. Several severe failures can in some way be traced back to a requirements problem [11][12]. At present, most requirements engineering approaches rely on structured natural language or semi-formal notations such as UML which lack precise semantics and thus have poor reasoning capabilities. Formal methods are generally applied to software specifications, while requirements consider the composite system which consists of the software and its environment. Often the used approach is missing fundamental requirements engineering issues such as the capture of rationale, adequate guidance for requirements elaboration and support in the exploration of alternatives. In [23], it is argued that goals offer the right kind of abstraction to address such inadequacies, notably in the context of high assurance systems; that is systems for which compelling evidence is required that the system delivers its services in a manner that satisfies safety, security, fault-tolerance and survivability requirements [15].

The key activity in goal-oriented requirements engineering is the construction of the goal model. Goals are objectives the system under construction must achieve. Goal formulations thus refer to intended properties to be ensured. They are formulated at different levels of abstraction from high-level, strategic concerns (such as "serve more passengers" for the train transporta-

[1] e-mail: {cp,phm,ari,jfm}@cetic.be
[2] e-mail: {avl,tvh}@info.ucl.ac.be
[3] This work is financially supported by European Union (ERDF and ESF) and Walloon Region (DGTRE).

tion system we will consider throughout this paper) to low-level technical concern (such as "acceleration command delivered in time"). Goal models also allows analysts to capture and explore alternative refinements for a given goal. The resulting structure of the goal model is a AND-OR graph.

The specific goal-oriented framework considered here is the KAOS methodology [2][24] which has a two level language: (1) an outer semi-formal layer for capturing requirements engineering concepts, structuring and presenting them; (2) an inner formal assertion layer for their precise definition and for reasoning about them.

The objective of this paper is to present an overview of the FAUST formal toolbox, to detail the individual tools and to show how, together, they provide powerful reasoning capabilities at an early stage of the system development. Figure 1 shows the main activities supported by the FAUST toolbox which are located at the top-level of the software lifecycle.

- *Early verification* is about making sure the system is correct, especially with respect to formal semantics of the goal model.
- *Validation* is about making sure the system being built is the system the user is expecting.
- *Artefact generation* is about automatically generating products used later in the software lifecycle such as acceptance test cases or run-time monitors.

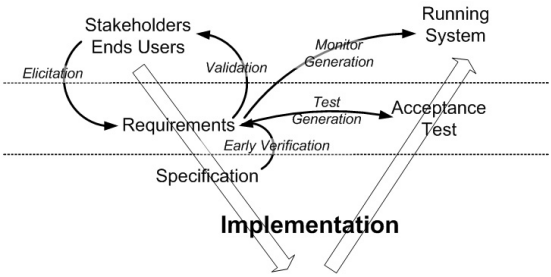

Fig. 1. Main activities supported by the FAUST toolbox

The FAUST toolbox was designed to make requirements engineering meet practical formal methods by:

- *seamlessly integrating semi-formal and formal descriptions*, allowing one to limit the scope of the latter to the critical parts of the system being modelled;
- *hiding most of the formal aspects*, eg. through the generation of animations based on domain-level representations;
- *providing goal-oriented formal tools* which encapsulate existing formal tech-

nology such as theorem provers, model-checkers, SAT-engines, constraint-solvers, etc.

- *reusing standard notations* where possible - for example, UML class diagrams, sequence diagrams, state machine diagrams.

The rest of this paper is structured as follows. Section 2 gives some background information on KAOS and its formal semantics. Section 3 gives an overview of the functionalities supported by the toolbox. Section 4 and 5 respectively detail two mature tools: the early analyzer (mainly about verification) and the animator (mainly about validation).

A demo of the tool in action on the train control system used as a running example in this paper can be downloaded at *http://www.cetic.be/∼faust/demo*. The system involves multiple trains moving along a circular single-track set of blocks with multiple stations, block signals, railroad crossing gates and cars. Only a subset of this model will be illustrated here.

2 Background on KAOS

A KAOS requirements model is composed of four sub-models: a goal model, an object model, an agent model and an operation model; these models are elaborated methodically using a goal-oriented approach, see [20].

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 the 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.

2.1 Building Goal Models

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 [2][3]. 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 [9]. 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 [2][8][14]. Keywords such as *Achieve*, *Avoid*, *Maintain* are used to name goals according to the temporal behavior pattern they prescribe.

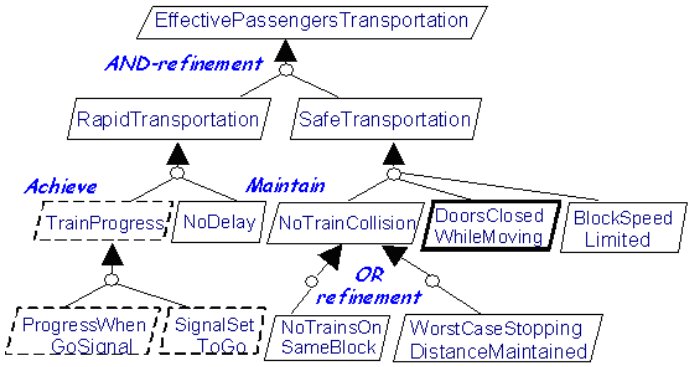

Fig. 2. Portion of a goal graph for a train control system

Figure 2 shows a goal model fragment of our 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:

$$(\forall tr : Train)tr.moving \Rightarrow tr.doorsClosed.$$

Note also the alternative refinement for the goal *NoTrainCollision* which leads to totally different designs when refined: block-based design for *NoTrainOnSameBlock* and speed control design (or moving blocks) for *WorstCaseStoppingDistanceMaintained*.

Goals refer to objects which may be incrementally derived from goal specifications to produce a structural model of the system (similar to 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 [9]. 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. The formal semantics of goal refinement is given in [3]:

Definition 2.1 (Semantics of Goal Refinement) *A set of goals $G_1, \ldots, G_n$ refines a goal G in a domain theory D if the following conditions holds:*

Completeness: $G_1, G_2, \ldots, G_n, D \models G$

Minimality: $\wedge_{j \neq i} G_j, D \not\models G \quad \forall i \in \{1, 2, \ldots, n\}$

Consistency: $G_1, G_2, \ldots, G_n, D \not\models false$

2.2 Operationalizing Goals

Goals are operationalized into specifications of operations to achieve them [2][10]. 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 $tr.doorsClosed$
DomPost $\neg tr.doorsClosed$
ReqPre for $DoorsClosedWhileMoving$: $\neg tr.moving$

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 *doorsClosed* appearing in the goal specification. The exact scope of the inputs and outputs of the operation is specified at entity/relationship level (eg. $tr : Train$) or more precisely at attribute level, using the / notation (eg. $tr : train/doorState$). 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 [10].

In [10] the semantics of the operation and operationalization are defined as follows:

Definition 2.2 (Semantics of operations) *For every operation op in the operation model, the predicate* $[|op|]$ *is defined as follows:*
$[|op|](arg_1, \ldots, arg_n, res_1, \ldots, res_n) \iff DomPre(op)$ *and* $\circ\, DomPost(op)$
where arg_i *denote inputs variables,* res_i *denote outputs and* $\circ$ *is the "next state" operator in linear temporal logic.*

Definition 2.3 (Semantics of pre-,trigger- and postconditions) *For every required condition R on an operation op in the operation model, the predicate* $[|R|]$ *is defined as follows*

$$\text{if } R \in ReqPre(op) \text{ then } [|R|] =_{def} (\forall *)[|op|] \Rightarrow R$$
$$\text{if } R \in ReqTrig(op) \text{ then } [|R|] =_{def} (\forall *)R \wedge DomPre(op) \Rightarrow [|op|]$$
$$\text{if } R \in ReqPost(op) \text{ then } [|R|] =_{def} (\forall *)[|op|] \Rightarrow \circ R$$

where $(\forall *)P$ *is the universal closure of* P.

Definition 2.4 (Correctness of goal operationalization) *A set* $R_1, \ldots, R_n$ *of required conditions on operations in the operation model correctly operationalize a goal G in the goal model iff the following conditions hold:*

$$Completeness: \quad R_1, \ldots, R_n \models G$$
$$Consistency: \quad R_1, \ldots, R_n \not\models false$$
$$Minimality: \quad G \models R_1, \ldots, R_n$$

The semantics of goal refinement and operationalization differ on an important point. While sub-goals refine their parent goal (ie. a model of the refined goals is also a model of the parent goal but not all models of the parent are covered), there is a logical equivalence between each operation specification and the enforced requirements. Moreover, that equivalence is independent of any domain property: it only relies on the requirement enforced on the system. This means that operations and requirements could interchangeably be given to the developer. Of course, it is often more indicated to provide them with both "views".

2.3 Producing Robust Requirements

The correctness of all refinements in a goal model does not ensure that the specification is consistent: inconsistencies can occur between goals. A *conflict* is a logical inconsistency between those goals. A *divergence* is a logical inconsistency under some (feasible) boundary condition. As opposed to goal refinement, checking inconsistencies is not a process local to a goal.

First-sketch models also tend to be overideal and are likely to be violated from time to time in the running system due to the unexpected behavior of agents. The lack of anticipation of such behaviors may lead to unrealistic, unachievable and/or incomplete requirements. Such exceptional behaviors are captured by formal assertions called *obstacles* to goal satisfaction.

Performing conflict and obstacles analysis is thus crucial for achieving high quality requirements. However it will not be detailed in this paper as the FAUST toolbox does not yet fully support them. The interested reader may refer to [22] and [21] for their formal definitions and how to discover and to handle them manually.

3 An Overview of the FAUST Toolbox

This section presents the main formal activities supported by the FAUST toolbox. Those activities and their flow of data are depicted in figure 3:

- **the early analyzer** checks the correctness about the goal-model. In case of error, it will produce a counter-example trace which can be replayed in the animator tool. It can also be used for validation purposes in order to produce constrained animation traces.
- **the FSM compiler** generates finite state machines from the operation model. These can be used later on in the development life-cycle or executed in the simulator for validation purposes.

- **the Simulator** is the engine with runs the finite state machine instances. It takes care of capturing the input commands from the person controlling of the animation, to trigger the relevant transitions and to notify the impacted visualization components of the changes which occurred.
- **the Animator** is a visual tool allowing one or more humans to (possibly concurrently) interact with a simulation or to replay an existing trace. It proposes different ways of viewing/controlling the designed system: symbolic FSM representations and graphical domain-based visualizations.
- **the Test Data Generator** generates acceptance tests which can be played on the developed system to check if it meets the requirements. They can also be directly played in the animator.
- **the Monitor Compiler** generates a monitor able to detect the violation of goals, requirements or assumptions. Those monitors can also be deployed at run-time as well as within the animator.

The toolbox is deployed as a formal extension of the *Objectiver* requirements engineering tool [18]. This integration allows the FAUST formal tools to rely on a powerful set of services for KAOS modelling, persistency, consistency checks and document generation.

The next section will detail the Early Analyzer and the Animator (including the Simulator, the FSM Compiler and the Monitor)

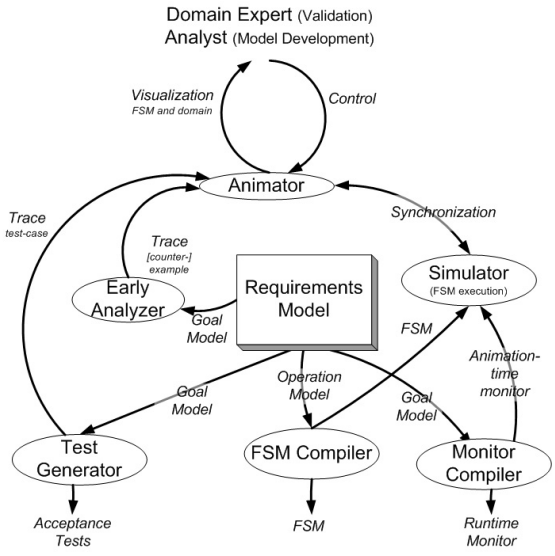

Fig. 3. Formal Activities

4 The Early Analyzer

In the incremental process of building four complementary models in parallel, errors will often occur in the formalization of the informal statements made by the stakeholders. The main purpose of the analyzer is to verify their formal consistency at an early stage of the process. Another use of the tool is to produce possible system histories for validation purposes.

4.1 Early Verification

All formal analysis is based on the formal semantics of the elements (goals, conflicts, obstacles, requirements, objects, operations, ...) contained in the four models. The analyzer can, most of the time, automatically formally verify the validity or invalidity of those elements on a given finite domain.

As a example we will consider the verification of a goal refinement. Given the semantics of goal refinement (see section 2), the following analysis can be made to prove the validity of the three required conditions:

- *Completeness:* Find a trace satisfying $G_1 \wedge G_2 \wedge \ldots \wedge G_n \wedge D \wedge \neg G$ if no trace is found, the refinement is "proved" (ie. on the considered finite domain).
- *Minimality:* To "prove" minimality (on the considered finite domain), no trace should be found for: $\wedge_{j \neq i} G_j \wedge D \wedge \neg G,\ \forall i \in \{1, 2, \ldots, n\}$
- *Consistency:* To "prove" consistency (on the considered finite domain), find a trace satisfying $G_1 \wedge G_2 \wedge \ldots \wedge G_n \wedge D$

The verification of the conditions related to operationalization, conflict, or obstacle is very similar to what has been shown for the goal refinement and will not be detailed here. Let us simply illustrate the operationalization of the already stated requirement *DoorClosedWhileMoving*. Most analysts will enforce the requirement by strengthening the preconditions of the operations possibly leading to the unsafe state: the operation to start the train and to open doors (see figure 4). It is also required to specify a safe initial state. However while trying to verify the equivalence between the operational model and the requirements, the early analyzer will produce a counter-example trace which is the following (considering a single train $tr\#1$)

1. $tr\#1.doorClosed \wedge \neg tr\#1.moving$
2. $\neg tr\#1.doorClosed \wedge tr\#1.moving$

   Loop goes back to state 1.

That counter-example is easy to understand: the operations *Start* and *OpenDoors* where triggered in the same state and a "race condition" leading to the unsafe state is allowed by the system. In this case the counter-example

is pretty simple to understand for an analyst but, in a more complex system, they can become fairly complex to analyze. The animator will greatly help in that task and is definitely necessary when system behaviors have to be shown to some stakeholder for validation purposes (see section 5).

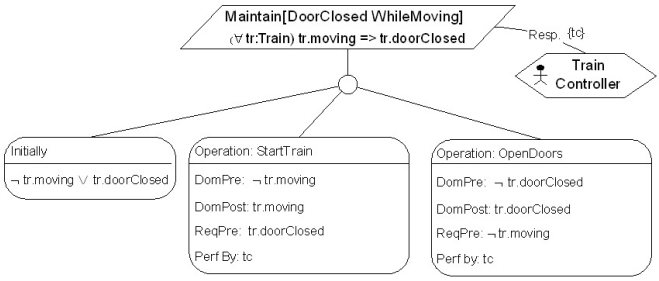

Fig. 4. A tentative operationalization for $DoorClosedWhileMoving$

To fix the synchronization problem, one approach is to explicitly forbid that behavior for example by stating the following assertion:

$$(\forall tr : train) tr.moving \Rightarrow \neg@\neg tr.doorClosed$$

whose responsibility should of course be examined. Another way is to enforce the postconditions rather that the preconditions. In fact the latter approach is documented as a well-known operationalization pattern [10] and if the analyst had looked in that library in the first place, the mistake would not have occurred. In both cases, the formal analyzer stops returning counter-examples.

4.2 Validation

The analyzer can automatically exhibit examples of behaviors of the system and its environment satisfying an objective, or showing the occurrence of a conflicting situation or an obstacle, or satisfying the properties associated to an object, or showing the occurrence of an operation.

For example, to show a situation where an obstacle O prevents an objective G, *"train progress with no delay on crossing block"*, to be fulfilled, it suffices to show a trace (sequence of states) of the system and its environment satisfying the formal definition of the domain properties D, *"trains cannot cross over cars"*, but not satisfying the formal definition of the objective G.

When the trace automatically generated by the analyzer is explained to the stakeholders (e.g. the trace exhibits a car blocked on the crossing block), they can confirm that this situation shows the existence of an obstacle (e.g. *"car crash on crossing block"*), or this situation points out a bad formal definition of the objective that should be modified (e.g. *"train progress with no delay*

on crossing block when no car on the crossing block"), or that some domain property is bad or missing (e.g. *"no crossing block is allowed, only bridges are"*).

4.3 Mapping onto Standard Formal Technology

The analyzer must automatically produce traces and proofs from formulas expressed in a first-order linear real-time temporal logic when requested by the user. The formulas used in each analysis are very small compared to what would have been done in a specification analysis activity. The analyzer takes advantage of that as follows :

1. For each request, a specific formal tool can be chosen which is most adapted to the kind of analysis and formulas used. For instance, traces are easier to find using bounded model checking : if interpreted symbols (eg. integer arithmetic) are used in the formulas, constraint programming techniques are preferred, whereas if no interpreted symbols are used, SAT-based techniques are very efficient.
2. Small formulas allow a simple and under-optimized mapping between the analyzer and the other tools : this results in a correct code that is easily adapted to new tools.
3. The running times are kept very small (often seconds, rarely minutes). If no answer is obtained, the analyst can ask to use another tool or make another local analysis.

So, the analyzer mainly helps to use different well-known formal tools, doing the forward and backward translation between the KAOS formalism and the formalisms of those tools. The difference with others is that the splitting of analysis and consolidation of their result is naturally integrated into the goal-oriented methodology.

The tools used are the BDD-based engine and the SAT-based engines of NuSMV [1] and the CLP engine of Oz [16]. Experiments with Alloy [7] show that it is better to use tools that provide high level input formalisms which optimize their mapping into SAT, BDD,... It is planned to use Alloy, some automatic theorem provers, and well-known theorem provers such as SteP or PVS (mainly to use their powerful decision procedures).

The different mappings replace infinite domains with finite ones (eg. Alloy), and replace the infinite time structure with a bounded one (eg. bounded model checking). The analyst must interpret the results obtained with caution. For instance, no bounded trace can be found if a counter constrained to augment indefinitely is modeled. The analyst will often foresee this because the analyzed formulas are small.

5 The Requirements Animator

Animation of goal-oriented requirements specifications is intended to help non-technical stakeholders validate them by interacting with the specification instantiated to examples they are familiar with. The animator tool is based on the KAOS operational model. It is composed of the following components:

1. *a state machine compiler* producing finite state machine descriptions from the KAOS operation descriptions. In this paper, those state machines are called Goal-Oriented State Machine (GSM) to emphasize their origin.
2. *a simulator* allowing one or several analysts to instantiate and animate those GSMs, either interactively or using previoulsy computed traces.
3. *an animator interface* including a control panel and a rendering engine, including a generic GSM viewer and a toolbox for designing user-level visuals.
4. *a animation watchdog* monitoring for the violation of the goals that are in the animation scope.

5.1 GSM Generation from Goal Operationalizations

The requirements animator requires an executable model. Finite state machines have that property and also a number of other interesting qualities for animation purposes: they can be traced back to goal/operation specification language, they are compositional to support parallel behavior and they have widely accepted statechart-like notations for visualizing them. The generation algorithm takes as input a goal scope (allowing one to analyze partial models) and outputs a set of flat finite state machines. The main steps of the algorithm is depicted in Figure 5. For the full algorithm, the reader may refer to [19].

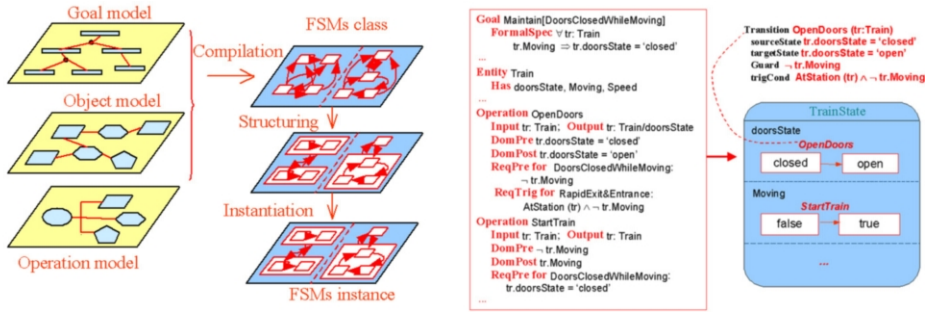

Fig. 5. State Machine Generation: principle (left) and example (right)

5.2 The Simulator

The simulator is responsible for instantiating the GSMs and managing those instances. It can be controlled using the following functionalities which are available from the animator interface.

1. *Instance explorer:* for viewing the objects involved and create new instances;
2. *Operation editor:* for triggering GSM transitions by applying enabled operations;
3. *State viewer:* for a raw overview of the current GSM instances
4. *Replay tool:* for going back and forward in the produced animation trace

5.3 The Animator Interface

The role of this component is to provide an adequate interface for a comprehensive interaction with the animation. It includes a number of ways for visualizing and controling the GSM instances. It is also deployed on a client-server architecture with multiple synchronized interfaces being connected to the same server. This allows for multiuser validation: each user impersonating a specific agent of the system which could be misbehaving (possibly intentionally such as intruders when designing a security system).

For a comfortable validation, visualization facilities are provided under two forms:

1. a symbolic GSM viewer allows the analyst to display the GSM instances in their classical statechart-like form.
2. domain-level graphical representations can be mapped onto the GSM, those are currently based on the Scene Beans framework [13].

Figure 6 shows such graphical representation for the counter-example discussed in section 4.2. The first state is on the left of the picture and the second on the right. Both pictures show: a symbolic GSM visualization (bottom right), a scene showing train doors (top left), a scene of a train track (top right, not relevant here) and a big control panel (in the foreground) where the trace generated by the analyzer is being replayed. The state on the left shows a stopped train with closed doors and the state on the right shows a moving train with its doors opened.

While the GSM visualization is generic, developing new domain-level views requires some work: the graphical scene has to be described by assembling graphical animation primitives which defines the available "sprites", how they can move or change and which variables are associated with them. The

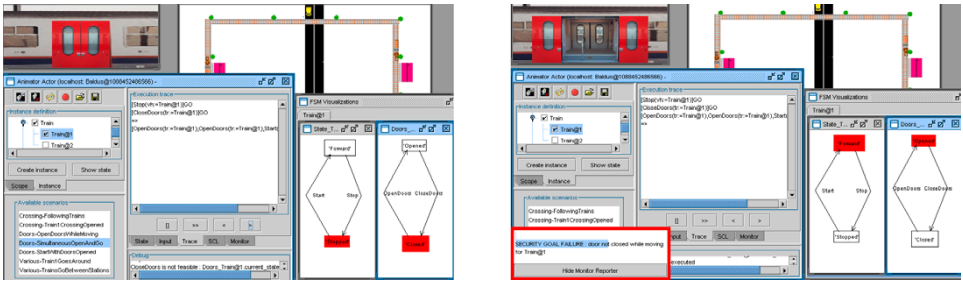

Fig. 6. Graphical Representations for the counter-example scenario of section 4.1

SceneBeans framework used here provides a good level of abstraction and is described in a simple XML format. It allows the designer to produce a new view and to interface it with the simulation engine in a few minutes for simple scenes (such as the train doors) to a few hours for more complex one (such as the global track view). A mapping tool helps in this process of connecting the view to the underlying model.

5.4 The Animation Watchdog

As analysts or validating users will generally want to check for goal satisfaction while playing with the system, the simulator also contains an animation watchdog. This tool monitors all the goals/requirements/assumptions being in the animation scope and reports any violation. It relies on another kind of finite state machine capable of accepting finite goal traces. Those are compiled at the same time the GSM are generated but are running completely independent of them. They are only synchronized on initialization and through the listening to the state changes occurring in the system.

Back to our example, the second and unsafe state in figure 6 shows a monitor popup in the bottom left part of the picture. This popup report the violation that just occurred with respect to the goal *DoorClosedWhileMoving*. Such a tool can thus greatly help pointing out defects but, of course, cannot provide any assurance the model is free from undesired behaviors: that is why the animator and the early analyzer are complementary tools.

The algorithm used for monitoring is detailed in [19]. As it is designed to scale up, it is not restricted to animation but can also to be deployed at runtime in real systems.

6 Discussion and Conclusions

The FAUST toolbox is aimed at achieving formal assurance at an early stage while avoiding obstacles to the application of formal methods and hiding for-

mal notations as much as possible.

The KAOS methodology has been applied for more than ten years on industrial case in many different domains such as telecommunication, steel industry, press and publishing. Typical requirements documents range from a few dozen to a few hundreds of goal and are successfully managed by Objectiver tool, the KAOS CASE tool [18]. The FAUST toolbox now extends the tool with key features for the analysis of mission-critical systems. The tight integration allows the analyst to go formal in an incremental way, only when and where needed, keeping the formal part small and manageable. The identification of patterns at semi-formal level can also help as their formalization can be proved once for all.

The example presented in this paper is an excerpt from a larger railways signaling specification including a level-crossing model built with the input of the Belgian railway company in the process of the replacement of a large number of level-crossings. The initial work document was based on operational description through state machine diagrams with the safety properties left in some annex. Our approach showed how to start from such properties to guide the elaboration of the requirements document and then to produce state machines satisfying them.

The Early Analyzer has also shown promising results in air traffic control for reasoning about air conflict detection. The goal model was composed of more than 200 concepts (goals, requirements, conflicts and obstacles) of which a subset was formalized in an incremental way, first reasoning without considering temporal aspects, then introducing real-time constraints and finally taking agent loads into account. The discovery of recurring domain-specific patterns also helped in the structuring and formalization processes. It is now considered to use the Monitor and the Animator in the same domain for classifying air-traffic data, to detect some incidents and near-misses and visualize them.

In the future, the early analyzer will be extended to support checks addressing obstacles and conflicts. The Animator Mapping is also being improved and a new component for designing control panels (such as train/plane/automative cockpits) is being implemented. Other tools such as the the acceptance tests generator and the obstacle generator are in the implementation phase or being planned.

References

[1] A. Cimatti, E. Clarke, E. Giunchiglia, F.Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *Nusmv version 2: An opensource tool for symbolic model checking*, Int. Conf. on Computer-Aided Verification (CAV02, LNCS 2404), Denmark, July 2002.

[2] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas, *Goal-directed requirements acquisition*, Science of Computer Programming **20** (1993), no. 1-2, 3–50.

[3] R. Darimont and A. van Lamsweerde, *Formal refinement patterns for goal-driven requirements elaboration*, FSE-4 - 4th ACM Symp. on the Foundations of Software Engineering, San Francisco, October 1996.

[4] Steve Easterbrook, Robyn R. Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton, *Experiences using lightweight formal methods for requirements modeling*, Software Engineering **24** (1998), no. 1, 4–14.

[5] The Standish Group, *http://www.standishgroup.com/chaos*, 1995.

[6] European Software Institute, *European user survey analysis, report usv_eyr 2.1 espiti project*, January 1996.

[7] D. Jackson, *Automating first-order relational logic*, ACM SIGSOFT in Proc. Conf. Foundations of Software Engineering, November 2000.

[8] R. Koymans, *Specifying message passing and time-critical systems with temporal logic, lncs 651*, Springer-Verlag, 1992.

[9] E. Letier and A. van Lamsweerde, *Agent-based tactics for goal-oriented requirements elaboration*, 2001.

[10] E. Letier and A. van Lamsweerde, *Deriving operational software specifications from system goals*, FSE'10: 10th ACM S1GSOFT Symp. on the Foundations of Software Engineering, Charleston, November 2002.

[11] N.G. Leveson, *Safeware, system safety and computers*, Addison-Wesley, 1995.

[12] R. R. Lutz, *Analyzing software requirements errors in safety-critical, embedded systems*, IEEE International Symposium on Requirements Engineering (San Diego, CA), IEEE Computer Society Press, 1993, pp. 126–133.

[13] Jeff Magee, Nat Pryce, Dimitra Giannakopoulou, and Jeff Kramer, *Graphical animation of behavior models*, International Conference on Software Engineering, 2000, pp. 499–508.

[14] Z. Manna and A. Pnueli, *The reactive behavior of reactive and concurrent system*, Springer-Verlag, 1992.

[15] J. McLean and C. Heitmeyer, *High assurance computer systems: A research agenda*, America in the Age of Information, National Science and Technology Council Committee on Information and Communications Forum, Bethesda, 1995.

[16] T. Muller, *Promoting constraints to first-class status*, First International Conference on Computational Logic (CL00, LNAI 1861), London UK, July 2000.

[17] John Rushby, *Disappearing formal methods*, High-Assurance Systems Engineering Symposium (Albuquerque, NM), Association for Computing Machinery, nov 2000, pp. 95–96.

[18] The Objectiver Tool, *http://wwww.objectiver.com*.

[19] H. Tran Van, A. van Lamsweerde, P. Massonet, and C. Ponsard, *Goal-oriented requirements animation*, 12th IEEE International Requirements Engineering Conference, Kyoto (Japan), September 2004, accepted.

[20] A. van Lamsweerde, *Goal-oriented requirements engineering: A guided tour*, 2001.

[21] A. van Lamsweerde, R. Darimont, and E. Letier, *Managing conflicts in goal-driven requirements engineering*, IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development (1998).

[22] A. van Lamsweerde and E. Letier, *Handling obstacles in goal-oriented requirements engineering*, IEEE Transactions on Software Engineering, Special Issue on Exception Handling **26** (2000), no. 10.

[23] A. van Lamsweerde and E. Letier, *From object orientation to goal orientation: A paradigm shift for requirements engineering*, Radical Innovations of Software & System Engineering, Montery'02 Workshop, Venice(Italy), LNCS, 2003.

[24] Axel van Lamsweerde, *Requirements engineering in the year 00: a research perspective*, International Conference on Software Engineering, 2000, pp. 5–19.