Deriving Event-Based Transition Systems from Goal-Oriented Requirements Models

Emmanuel Letier
Dpt. d'Ingénierie Informatique, Université catholique de Louvain
Louvain-la-Neuve, Belgium
eletier@info.ucl.ac.be

Jeff Kramer, Jeff Magee, Sebastian Uchitel
Department of Computing, Imperial College London
and London Software Systems, U.K.
{jk, jnm, su2}@doc.ic.ac.uk

Abstract. *Goal-oriented methods are increasingly popular for elaborating software requirements. They offer systematic support for incrementally building intentional, structural, and operational models of the software and its environment. Event-based transition systems on the other hand are convenient formalisms for modelling and reasoning about software behaviours at the architectural level.*
The paper combines these two worlds by presenting a technique for translating formal specification of software operations built according to the KAOS goal-oriented method into event-based transition systems analysable by the LTSA toolset. The translation involves moving from a declarative, state-based, timed, synchronous formalism typical of requirements modelling languages to an operational, event-based, untimed, asynchronous one typical of architecture description languages. The derived model is used for the formal analysis and animation of KAOS operation models in LTSA.
The translation process provides insights into the two complementary formalisms and raises questions about the use of synchronous temporal logic for requirements specification.

Keywords Goal-Oriented Requirements Engineering, Event-based Transition Systems, Fluent Temporal Logic, Requirements Animation, Model-Checking.

1 Introduction

Goal orientation is a recognized paradigm for elaborating, structuring and analysing software requirements [Lam00b, Lam01, Chu00]. *Goals* are desired system properties whose satisfaction requires the cooperation of *agents* (or active components) in the software and its environment. Goals may refer to functional or non-functional concerns and range from high-level, strategic concerns (such as "avoid explosion" for the safety control mechanisms of a nuclear power plant) to low-level, technical ones (such as "safety injection overridden when block switch is on and pressure is less than 'Permit'").

Event-based transition models on the other hand are convenient formalisms for modelling and reasoning about software behaviours at the architectural level. They describe a system as a set of interacting components where each component is modelled as a state machine and interactions between components occur through shared events. Such models provide the basis for a wide range of automated analysis techniques, notably deadlock detection, model animation and model verification through model checking [Mag99].

Integrating goal-oriented requirements elaboration methods and specification techniques founded on event-based transition systems provides clear benefits:

- the former provides a *systematic method* for modelling the real-world goals of a system, structuring them in a refinement hierarchy [Dar96, Chu00, Let02a], reasoning about their conflicts [Lam98] and exceptions [Lam00a], reasoning about the impact of alternatives on non-functional goals [Chu00, Let04], and gradually deriving a specification of software operations that ensures the goals [Let02b];
- the later can be used for the *automated formal analysis* of software specifications and provides the basis for the downstream activities of software architectural design, program verification, specification based testing, etc [All97, Mag99].

Our work is motivated by this complementarity. The paper describes a technique for transforming KAOS specification of operations, derived from goals according to techniques described in [Let02b], into event-based transition system analysable by the LTSA toolset.

The derived model can then be used to carry out formal analysis of KAOS operation models in LTSA: (i) incompleteness with respect to goals can detected by model-checking the derived event-based transitions system against goals; (ii) inconsistencies and implicit requirements can be detected as deadlocks or violation of a time progress property; and (iii) animation can be performed using the standard animation features of LTSA.

Similar capabilities for animating and model-checking KAOS operation models have been developed in another prototype tool [Tra04, Pon04]. Animation there is preformed by translating KAOS operations into a special-purpose state-machine formalism, called goal

based state-machine, and model-checking by translating KAOS models into the symbolic model-checker NuSMV [Cim02]. No support is available there to detect inconsistencies and implicit requirements in KAOS operations.

In this paper we follow a different approach by translating KAOS models into *event-based* transition systems. Event-based formalisms are a natural choice for representing behaviours at the architecture level [All97, Ber91, Mag95]. At this level, one is concerned with the interactions between components in terms of messages sent and received by components and service invocations initiated or accepted by components. These messages and invocations are most naturally modelled as events. Our main reason for choosing an event-based formalism as target language is to facilitate the transition from goal-oriented, state-based requirements models to event-based architecture models.

In [Del03], we have investigated the possibilities of tranlsating KAOS operations into tabular specificatin analysable by the SCR* toolset [Hei96, Hei98]. However, a semantic incompatibility between the two languages concerning the "synchrony hypothesis" made it impossible to derive an SCR specification whose behaviours are exactly the same as those of the source KAOS model. The translation to event-based transition system described in this paper preserves exactly the semantics of the source KAOS model.

The rest of the paper is organized as follows. Section 2 introduces background material on KAOS and LTSA. Section 3 presents a technique for deriving event-based transitions systems from assertions written in fluent temporal logic, the formalism used in LTSA to specify state-based temporal properties over event-based systems. This technique is then used in Section 4 to translate goal-oriented operation models into event-based transitions system. Section 5 describes how to use the derived model for the formal analysis and animation of the KAOS model. Section 6 discusses limitations of an alternative approach in which LTS would be derived from KAOS goals instead of operations.

2 Background

Our presentation will rely on the safety injection system for a nuclear power plant [Cou93, Heit96]. The purpose of this system is to prevent or mitigate damages to the core and coolant system on the occurrence of a fault such as a loss of coolant. The ESFAS (Engineered Safety Feature Actuation System) is a software component that monitors steam pressure in the coolant system; if the pressure falls below some 'Low' value, a safety injection signal is sent to safety feature components, the function of which is to cope with the incidents. A manual block button allows operators to override safety injection during a normal start-up or cool down of the power plant reactor. The manual block must be automatically reset by the system when the pressure rises above a value 'Permit'.

2.1 Goal-Oriented Modelling with KAOS

A KAOS model is specified as a composition of four submodels: a *goal model* in which the goals to be achieved are described together with their refinement/conflict links; an *object model* in which the application objects involved are described together with their relationships and attributes; an *operation model* in which the services that operationalize the goals are described; and an *agent model* in which the agents are described together with their interfaces and responsibilities with respect to goals and operations. The reader may refer to [Lam01, Let01] for a full description of the KAOS method.

2.1.1 Specifying Goals in LTL

A *goal* is a declarative property to be satisfied by the system under consideration. The word "system" refers to the software and its environment. An obvious goal for the nuclear power plant is:

Goal Avoid[Explosion]
Def. The nuclear reactor should not explode.
FormalDef [] ¬ Explosion

This specification fragment introduces a goal named Avoid[Explosion] (Avoid is a keyword that refers to the temporal pattern of the goal). Every goal is defined by a natural language assertion for communication with stakeholders and a linear temporal logic assertion for formal reasoning.

Linear temporal logic assertions are formed from state variables, standard Boolean operators, and qualitative linear temporal logic operators X (next), [] (always), <> (eventually), U (until) and W (Awaits). Real-time and past operators are allowed as well but will not be used in this paper. State variables (such as Explosion) correspond to object attributes and relationships in the application domain object model.

The semantics of the temporal logic operators is defined following [Man92]. Let *V* be a set of state variables. A system state is a mapping that assigns a value to each state variable. An history is a mapping *h: Nat* → *State* that maps each position *i* to the global system state at that position. The notation (h,i) |= P is used to express that the LTL formula P is true at position i of history h. The temporal operators we will use in the paper are defined as follows:

- (h,i) |= X P *iff* (h, i+1) |=P
- (h,i) |= [] P *iff* (h, j) |=P *for all* j ≥ i
- (h,i) |= <> P *iff* (h, j) |=P *for some* j ≥ i

The notation @P is also used as a shorthand for (¬ P ∧ X P). A LTL formula P is said to be satisfied by a trace h, noted h |= P, if it is satisfied at the initial position, i.e. (h,0) |= P.

Goals are organized in AND/OR-refinement structures. *AND-refinement* links relate a goal to a set of subgoals (called *refinement*); this means that satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. *OR-refinement* links relate a goal to an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition for satisfying the goal. High-level goals are recursively refined into subgoals until each terminal goal is realizable by some individual agent, in the sense that it is defined in terms of variables that are monitored and controlled by the agent [Let02a]. A *requirement* is a terminal goal assigned to an agent in the software-to-be. For example, the gradual refinement of the goal Avoid[Explosion] generates, among others, the following requirement assigned to to the ESFAS component:

Req Maintain[SafetyInjectionOverridden Iff Block and Pressure LessThan Permit]

Def: Safety injection should become overridden if and only if block is pushed while the steam pressure is less than permit.

FormalDef: [] (@Overridden ↔ block ∧ Pressure<'Permit')

Responsibility ESFAS

The reader may refer to [Let02c, Lam04] for a full KAOS elaboration of the requirements for the safety injection control system.

2.1.2 Modelling Operations

A goal assigned to some agent in the software-to-be is operationalized into functional services, called *operations*, to be performed by the agent. Operations are specified in a state-based style by pre, post- and trigger conditions. An important distinction is made between *domain* pre-/postconditions, which capture the elementary state transitions defined by operation applications in the domain, and *required* pre/post/trigger conditions, which capture additional requirements on operation application to ensure that the goals are met. A *required precondition* for some goal captures a permission to perform the operation when the condition is true. A *required trigger condition* for some goal captures an obligation to perform the operation when the condition is true provided the domain precondition is true. A *required postcondition* defines some additional condition that any application of the operation must establish in order to achieve the corresponding goal.

Consider the following operation that operationalize the requirement Maintain[SafetyInjectionOverridden Iff BlockAnd PressureLessThanPermit].

Operation overrideSI
DomPre ¬ Overridden
DomPost Overridden
ReqPre/TriggerFor {SafetyInjectionOverridden Iff BlockAndPressureLessThanPermit}:
block ∧ ¬ Pressure<'Permit'

In this specification, the domain pre and postconditions capture the domain property that every application of overrideSI corresponds to a transition from a state where safety injection is not overridden to a state where it is overridden. The ReqPre/Trigger condition means that the condition is both a required pre- and a required trigger condition. It captures that the operation must be applied if block occurs and the steam pressure is above permit, and can only be applied in this circumstance.

The formal semantics for KAOS operation models is given by translating KAOS operations into temporal logic assertions [Let02b]. Let [|op|] be a predicate denoting the application of the operation op in the current state. This predicate holds over the pair of states satisfying the operation domain pre/post conditions:

[] ([|op|] ↔ DomPre (op) ∧ X DomPost (op))

The semantics of required pre-, trigger- and post-conditions is defined as follows:

[] ([|op|] → ReqPre)

[] (ReqTrig ∧ DomPre → [|op|])

[] ([|op|] → X ReqPost (op))

KAOS operation models are interpreted over sequences of system states where consecutive states are separated by a single time unit. Zero, one or more operations may occur between two consecutive states. This *non*-interleaving semantics is required by the semantics of trigger conditions as immediate obligations. With an interleaving semantics, an operation model would be inconsistent when the trigger conditions of two (or more) operations are true at the same time.

A set of required pre-, trigger-, and postconditions on operations is said to be a *complete operationalization* of a goal if satisfying all required conditions in the set guarantees the satisfaction of the goal.

A KAOS operation model is said to be *consistent* if every reachable state has a successor that satisfies all domain pre/post conditions and required pre/post/trigger conditions. There are different ways in which a KAOS operation model may contain inconsistencies. For example, if the required trigger condition of an operation does not imply its required preconditions, the system might be in a state in which the required trigger condition is true and the required precondition is not, so that the operation must be applied and may not be applied at the same time.

KAOS operation models may also contain *implicit requirements* that are due to interactions between

requirements on different operations. For example, a trigger condition on one operation may prevent another operation from being applied even if all required preconditions on this operation are true. Implicit required conditions do not necessarily entail inconsistencies. An operation has an *implicit required precondition* when the actual condition for which the operation is allowed to occur is weaker than its stated domain and required preconditions. An operation has an *implicit required trigger condition* when the actual condition for which the operation must occur is stronger than its stated required trigger conditions.

2.2 Behaviour Analysis with LTSA

Our target event-based formalism is that of Labelled Transitions Systems (LTS) [Mag99].

2.2.1 Labelled Transition Systems

Let *Act* be the universal set of observable events and let τ denote a local action that is unobservable by a component's environment. An LTS *M* is a quadruple $<Q, A, \delta, q_0>$ where *Q* is a finite set of states, $A \subseteq Act$ is the communicating *alphabet* of *M*, $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ is a labelled transition relation, and $q_0 \in Q$ is the initial state.

The semantics of an LTS *M* is a set of sequences of events (observable or τ) that the LTS can perform starting in its initial state. The parallel composition operator "||" is a commutative and associative operator that combines the behaviour of two LTSs by synchronizing the events common to their alphabets and interleaving the remaining events.

Discrete-time systems can be modelled by including an explicit tick event signalling the regular ticks of a global clock to which each timed process synchronizes. When modelling a timed system, one has to ensure the LTS model does not indefinitely prevent time from progressing. This can be verified automatically in LTSA by checking that the LTS model does not deadlock and that tick events may occur infinitely often in every infinite execution of the LTS i.e. it satisfies the following "progress property"

progress TimeProgess = {tick}.

2.2.2 Fluent Temporal Logic

Fluent Linear Temporal Logic (FLTL) provides a uniform framework for specifying and model-checking state-based temporal properties to be satisfied by event-based transition systems [Gia03]. The motivation for this formalism is that properties to be satisfied by an event-based transition system are often much easier to specify if they can refer to system states in addition to referring to events.

A *fluent Fl* is a state predicate defined by a set of initiating events $Init_{Fl}$, a set of terminating events $Term_{Fl}$, and an initial value $Initially_{Fl}$ that can be either true or false. The sets of initiating and terminating events must be disjoint. By default, the initial value of a fluent is false. The concrete syntax for fluents in LTSA is the following:

fluent $Fl = < Init_{Fl}, Term_{Fl} >$ initially $Initially_{Fl}$

Given a set Φ of fluents and an event trace $tr: Nat \rightarrow A$, one can construct an associated state-based trace $StateTrace(tr): Nat \rightarrow A$ that gives the fluent values after the occurrence of each event in *tr*. Such state-based trace is defined as follows: for every position $i \in Nat$ and every fluent $Fl \in \Phi$, *Fl* is true at position *i* of *StateTrace(tr)* iff either of the following conditions holds

(a) *FL* holds initially and no terminating event has occurred before position *i*:

$Initially_{FL}$ *and there is no* $k \in Nat, 0 \leq k \leq i$ *s.t.* $tr(k) \in Term_{Fl}$

(b) some initiating event has occurred before position *i* and no terminating event has occurred since then:

there is some $j \in Nat, j \leq i$, *s.t.* $tr(j) \in Init_{Fl}$

and there is no $k \in Nat, j < k \leq i$, *s.t.* $tr(k) \in Term_{Fl}$

A well-formed FLTL formula is an LTL formula whose atomic propositions are fluents. An FLTL formula *P* is said to be satisfied by an event trace *tr*, noted $tr \models P$, iff $StateTrace(tr) \models P$.

FLTL assertions can refer to event occurrences. For every event *e* in an LTS model there is an implicit fluent, also noted *e*, whose set of initiating events is the singleton event {e} and whose set of terminating events contains all other events in the alphabet A:

fluent e = <e, A-{e}> Initially false

According to this definition, the fluent associated with an event *e* becomes true the instant *e* occurs and become false with the first occurrence of a different event.

The concrete syntax for FLTL assertions used in LTSA uses the ASCII symbols !, &&, and || for logical negation, conjunction and disjunction, respectively.

2.2.3 From Synchronous to Asynchronous TL

FLTL assertions are interpreted over sequences of system states observed after each occurrence of an event, whereas KAOS models are interpreted over sequences of system states observed at a fixed time rate so that zero, one or more events may occur between consecutive states. We have called *synchronous temporal logics* those that are interpreted over sequences of states observed at a fixed time rate, and *asynchronous* those that are interpreted over sequences of states observed after each occurrence of an event [Let05].

Temporal logic operators have very different meanings in synchronous and asynchronous temporal logics. For example, 'X P' in an asynchronous temporal logic means 'P holds after the next event', whereas in a synchronous temporal logic it means 'P holds at the next time unit'. Similarly, '[] P' in an asynchronous temporal logic means 'P holds after each event' whereas in a synchronous temporal logic it means 'P holds at each time point'.

In [Let05], we have defined an encoding of synchronous temporal logic into asynchronous temporal logic. As an example, this encoding translates the KAOS goal definition

[] (@Overridden ↔block ∧ PressureLessThanPermit)

into the following asynchronous FLTL assertion analysable in LTSA:

```
[] (tick ->
 ((!Overridden && X(!tick W (tick && Overridden))
 <-> block && PressureLessThanPermit)))
```

This mapping allows LTSA modellers to use a goal-oriented requirements elaboration process à la KAOS for the incremental identification, elaboration and specification of the formal properties to be model-checked with the LTSA toolset. However, LTSA modellers are still required to specify LTS behaviour models in FSP (the process algebra used in LTSA to concisely specify LTSs) and to provide the fluent definitions that relate the predicates involved in the goal definitions to the events appearing in the FSP model. The purpose of this paper is to present a technique for automatically deriving the fluent definitions and LTS models from KAOS operations.

3 From FLTL Assertions to LTS Models

Our translation from KAOS operation model to LTS will use a technique for translating FLTL assertions into LTS that we describe in this section. The existing technique for model checking a FLTL assertion ϕ in LTSA involves constructing a Buchi automata B that recognizes all infinite event-based traces that violate ϕ and checking that the synchronous product of B with the LTS to be verified is empty. When the assertion is a safety property, the Buchi automata can be viewed as a "property LTS", i.e. an LTS with an ERROR state so that executions leading to the error state correspond to undesired system behaviours. All executions of this LTS that do not reach the error state satisfy the assertion. Removing the error states and the transitions leading to it yields a LTS that captures all traces on the alphabet of ϕ that satisfy ϕ. We have extended LTSA so that this LTS can be generated using the keyword `constraint` in front of a safety FLTL assertion. If the FLTL assertion is not a safety property, an error message is generated.

LTSs derived from FLTL assertions can be composed as any other process. In the FLTL framework, a formula is said to be closed under stuttering if the satisfaction of the formula by a trace is unaffected by the insertion or removal of silent events τ from the trace. It can be shown that if two safety properties P and Q are closed under stuttering then the parallel composition of their derived LTSs is equivalent to the LTS derived from their logical conjunction, i.e.

(constraint P || constraint Q) ≡ constraint (P ∧ Q).

The proof of this property is omitted here due to space limitation.

4 From KAOS Operations To LTS

The derivation of LTS models from KAOS operations is composed of the following three steps. These steps can be fully automated with user interaction required only to bound the infinite scope of the KAOS model if necessary. However, we have not built an implementation of this translation.

4.1 Identifying Fluents

The first step consists in identifying fluents from KAOS state variables. Boolean state variables, such as Overridden, are trivially mapped to a fluent with the same label. Integer and enumerated state variables are mapped to parameterized fluents where the fluent parameter records the state variable value. For example, the state variable Pressure whose values are in the integer range PressureRange yields the following fluent declaration:

fluent Pressure [i: PressureRange]

If a KAOS state variable has an infinite range, this range needs to be bounded by some maximal value in order to be analysable in LTSA. First-order KAOS models in which state variables correspond to attributes and relationships of an object model can be handled by bounding the number of instances of each object in a way similar to the technique used in Alloy [Jac00]. KAOS object models however are much simpler, and less expressive than the one allowed in Alloy (in particular they do not allow transitive closure), so that the mapping from KAOS object models to fluents is straightforward and allows fluents to be easily interpreted back as KAOS state variables.

4.2 Deriving Fluents Definitions

Fluents' initiating and terminating events are then derived from the domain postconditions. We assume that domain postconditions are given as conjunctions of positive or negative occurrences of fluents, i.e. disjunctions are not allowed. Since domain pre/postconditions corresponds to elementary state transitions, this assumption is generally satisfied. If it is not satisfied our derivation process cannot be applied.

Rule for deriving fluent definitions: An operation is part of the initiating (resp. terminating) events of a fluent if and only if there is a positive (resp. negative) occurrence of the fluent in the operation domain postconditon.

This rule also assumes that a single fluent does not have both positive and negative occurrences in the domain postcondition of an operation. This assumption can always be satisfied by simply replacing a domain postcondition in which a fluent appears both positively and negatively by 'false'.

For example, the domain postconditions for the operation overrideSI in Section 2.1.2 and its dual operation enableSI yield to the following definition:

```
fluent Overridden = < overrideSI, enableSI >.
```

4.3 Deriving Labelled Transition Systems

Labelled transition systems are then derived from the domain preconditions and the required pre-, trigger-, and post-conditions.

Domain preconditions are translated according to the following template:

constraint DomPre_<Operation> =
[]((tick && !<DomPre>)->X(!<Operation> W tick))

The assertion (in asynchronous FLTL) states that if the domain precondition of an operation is not true at an occurrence of a tick then that operation cannot occur at least until the next occurrence of a tick. This constraint is a safety property that is closed under stuttering (despite the use of X). LTSs derived from domain preconditions can therefore be combined through parallel composition according to the proposition in Section 3.

Furthermore, since a transition between two states in a KAOS trace is a *set* of operations, a single operation cannot occur more than once between two observable states (i.e. between two occurrences of tick). This is modelled by an additional process that is combined with the LTSs derived from the domain preconditions.

As an example, the translation of the domain pre- and post-conditions for the operations overrideSI and enableSI yield the LTS shown in Figure 1.

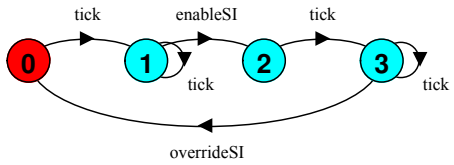

Figure 1. LTS derived from domain pre/post conditions

Required pre/trigger/post conditions are then translated according to the following constraint template:

constraint ReqPre_<Operation>_For_<Goal>
= []((tick && !<ReqPre>) -> X (!<Operation> W tick))

constraint ReqTrig_<Operation>_For_<Goal>
= []((tick && <ReqTrig> && <DomPre>)
-> X (!tick W <Operation>))

constraint ReqPost__<Operation>_For_<Goal>
= [](action -> (! tick W (tick && ReqPost)))

These assertions encode in FLTL the temporal logic semantics of KAOS operation models given in Section 2.1.2. All these assertions are safety properties that are closed under stuttering.

In our running example, the complete KAOS model for the ESFAS component has 4 operations and 8 required pre- and trigger-conditions. A total of 12 FLTL constraints are therefore translated into LTSs to derive the event-based model for this component. Our KAOS model also includes operations specifying the behaviour of the nuclear reactor and cooling system in the software environment. Generating the global model requires translating and composing 50 FLTL constraints. We have also applied the technique to the mine pump control system [Kra83] whose KAOS model [Let01] has a size similar to that of the safety injection system. Although both models are translated "by hand", the translation follows strictly the procedure described in the paper.

The resource-consuming steps in our derivation are the translation of FLTL assertions to Buchi automaton and the parallel composition of all derived automaton. The complexity of each translation is exponential in the size of the FLTL formula, but since each formula corresponds to a single required condition on an operation, each formula remains small. In our experience, this translation runs into a state explosion problem if a required condition is composed of more than 6 fluents, which is rarely the case. The most delicate part is the composition of all LTSs derived from the required conditions. Even if the state space of the final system is manageable, the intermediate state space generated during parallel composition might run into a state explosion. However, as will be seen in the following section, we can take advantage of the goal-oriented structure of the KAOS operation model to slice the LTS model based on the portion of the goal model to be analysed. The formal analysis of the model therefore does not require composing the LTSs derived from all the required conditions.

5 LTSA Analysis of KAOS Operation Models

Formal analysis and animation of KAOS operation models can be performed on the derived LTS models.

5.1 Checking Goal Operationalization

An important check on the KAOS operation model consists in verifying whether a set of required pre-, trigger- and post-conditions {R1, ..., Rn} form a complete

operationalization of a goal G. This can be performed with LTSA by model-checking whether the translation of G in asynchronous FLTL is satisfied by the model obtained from the composition of the LTS derived from the required condition R1, ..., Rn and the LTSs derived from the domain preconditions of the operations involved. Only the portion of the LTS model necessary for the verification is generated. If the KAOS model has been bounded in order to generate the finite-state LTS, the completeness of the goal operationalization in the bounded scope does not ensure that the operationalization is complete for an unbounded scope.

In our running example, we can verify that the required pre and trigger condition on the operation overrideSI in Section 2.1.2 is a complete operationalization of the goal Maintain [SafetyInjection Overridden Iff Block And Pressure LessThan Permit]. If we remove the term "!PressureAbovePermit" from the required pre/trigger condition on the operation, the goal operationalisation is no longer correct. This is automatically detected by the tool that generates the following error trace:

```
tick    Overridden
block
raiseAbovePermit
tick    block && PressureAbovePermit
overrideSI
tick    Overridden && PressureAbovePermit
```

An error trace is a sequence of events annotated with the fluents that holds at every occurrences of tick (since KAOS goals are synchronous assertions, the satisfaction of a goal depends only on fluents values when tick occurs). The error trace here shows that if block occurs when pressure is above permit (at the next to last occurrence of tick), the operation overrideSI is applied, leading to a state where Overridden is true which violates the goal. This error trace helps identifying the missing term in the required preconditions of the last operation.

Note that, thanks to our simple mapping from KAOS state variables and opertions to LTSA fluents and events, error traces generated by LTSA are easily interpretated as KAOS traces, without the need to explicitly map back LTSA traces to KAOS traces.

5.2 Checking Higher-level Goals

In addition to checking a single goal operationalisation, the model-checking feature of LTSA may be used to check the satisfaction of higher-level goals by operation models describing the behaviours of several agents in the software-to-be and its environment.

Since KAOS operation model typically represent only the operations performed by software agents, KAOS modellers first have to extend the operation model with operations in the environment. The goal refinement graph and obstacle model provides guidance for identifying and specifying these operations.

```
tick                    tick
start                   lowerPressure.6
tick                    stabilizeReactor
startUpReactor          tick
tick                    lowerPressure.5
raisePressure.0         tick
tick                    lowerPressure.4
raisePressure.1         tick
tick                    lowerPressure.3
raisePressure.2         block
tick                    tick
raisePressure.3         overrideSI
tick                    lowerPressure.2
raisePressure.4         tick
enableSI                lowerPressure.1
tick                    tick
raisePressure.5         explode
leakAppears             tick        Explosion
```

Fig. 2. Error Trace to Explosion

Consider for example, the high-level goal Avoid[Explosion]. Our operation model for the safety injection system is extended with environment operations such as explode identified from the high-level goal and leakAppears identified from an obstacle in our goal model. The explode operation specifies that an explosion occurs when the reactor is on and the steam pressure in the cooling system is null. Other environment operations include startUpReactor, stopReactor, stabilizeReactor, and coolDownReactor describing how the nuclear reactor changes states, and raisePressure, lowerPressure describing how the steam pressures varies according to the state of the reactor and the presence or not of a leak in the cooling system.

The satisfaction of a high-level goal is checked against the LTS derived from all required conditions operationalizing subgoals of the high-level goal and all domain operations concerning the fluents involved in this goal graph. The goal graph defines the scope of the operation model from which to derive the LTS model.

Checking the high-level goal Avoid[Explosion] generates an error trace displayed in 2 columns in Fig. 2. The trace shows that an explosion is possible if a leak occurs when the pressure is above 'Permit' (level 4) and the operator pushes the block button when the pressure is between 'Permit' and 'Low' (level 2).

5.3 Deadlock Analysis

As mentioned in Section 2.2.1, timed LTSs must not prevent time from progressing. This can be verified automatically by checking whether the model contains deadlock or violation of the time progress property. Our translation rules ensure that every deadlock and violations of time progress correspond to either an inconsistency or an implicit required precondition in the source KAOS operation model.

As a simple example, if the specification of a KAOS operation is inconsistent because one of its required

trigger condition does not imply all of its required preconditions, the tool will identify a deadlock and generate a sequence of events leading to a state where the required trigger condition is true and one of the required preconditions is false. Checking for deadlocks also allows one to detect more subtle inconsistencies in KAOS operations. For example, our initial operation for the reactor's operations contained two inconsistencies. In both cases, the inconsistency were due to interference between two operations that had trigger conditions that could be true at the same time but inconsistent domain postconditions. The deadlock trace generated by the tool was much helpful in locating the cause of the problem in the KAOS model.

Some deadlocks and violations of time progress are caused by implicit required precondition. For example, consider the operation lowerPressure and the required trigger condition 'Leaked' saying that lowerPressure must occur if there is a leak in the cooling system.

Checking the derived LTS model generates a deadlock trace in which 'Leaked' holds at the last occurrence of tick, followed by an occurrence of raisePressure leading to a state where no further occurrence of tick is possible because the required trigger condition on lowerPressure prevents tick from occurring until lowerPressure has occurred. Since lowerPressure and raisePressure cannot be applied simultaneously, the system is in a deadlock. The deadlock is due to an implicit required precondition '!Leaked' on raisePressure induced by the required trigger condition on lowerPressure.

In KAOS operation models, making implicit requirements explicit is not mandatory as it does not change the semantics of the model. It is however necessary to make them explicit to remove the deadlocks form the derived LTS model. When the implicit required preconditions are in software operations, making them explicit is also useful when moving towards an implementation of the operations, as it exposes hidden requirements that developers will have to take into account. The automated derivation of the implicit requirements could be performed for some restricted simple cases. The general case however is much more difficult to handle.

5.4 Goal-Based Animation

Our translation also allows one to animate KAOS operation models using the animation features of LTSA [Mag00]. The animator can be used to explore the behaviours of the model interactively with stakeholders or to replay error traces generated during formal analysis. The animation can be visualized as textual sequence of events (as shown in the paper) or as a graphical animation of a domain scene.

The animation of goal-derived LTS enjoys the key benefits of goal-based animations [Hun04]: (i) it is possible to animate partial operation models associated to specific goals; goals provide the scope of the animation, and (ii) the animator may automatically detect goal violations during its execution. In order to achieve this, the animated model is composed with goal monitors that are "property LTS" derived from goal definitions. Space limitation prevents us from illustrating this in the paper.

6 Why not derive LTS from KAOS goals?

An alternative approach to combining goal-oriented requirements models and event-based transition systems would consists in deriving LTS directly from goals using the technique described in Section 3. KAOS modellers would still be required to provide operations domain pre- and post-conditions in order to be able to establish the necessary link between fluents and events, but they would be relieved from specifying the required pre-, trigger-, and post-conditions operationalising the goals.

Unfortunately, because KAOS goals are synchronous temporal logic assertions, goal-derived LTSs are not adequate models of behaviours: they constrain the occurrences of tick only, instead of constraining which operations components are allowed to perform at every stage of their executions [Let05].

Consider the following goal requiring a pump to be on when the steam pressure rises above 'Permit':

[] (PressureAbovePermit -> X PumpOn)

Fig. 3 shows the LTS derived from this goal. In the initial state (state 0), PressureAbovePermit and PumpOn are both false. Occurrences of tick are not allowed in states 4 and 6 because PressureAbovePermit was true at the last occurrence of tick and PumpOn is currently false, therefore an occurrence of tick from one of these states would violate the goal. However, this LTS is not an adequate model of system behaviour because all other events are allowed in every state even if their application results in a goal violation in the KAOS model. For example, in state 5, PressureAbovePermit was true at the last occurrence of tick and PumpOn is currently true, but the LTS still allows stopPump to occur (leading to state 4) although the goal requires the pump to remain on.

If we combine the LTS derived from the goals with those derived from domain preconditions, the resulting LTS model contains deadlocks that are due to the fact that the goal-derived LTSs do not prevent the occurrence of operations (such as the occurrence of stopPump from state 5) that leads to states where tick is not allowed; and the LTS derived from the domain precondition prevent the application of other operations (in our example, startPump) that would bring the system back into a state

where tick is allowed.

In order to avoid such deadlocks, the goal-derived LTSs should be modified so as to prevent an operation if its occurrence leads the system in a state from which no further occurrences of tick are possible. This means that the LTS should prevent transitions labelled with operations whose occurrences between the current observable state (i.e. at the last occurrence of tick) and the next observable state (the next occurrence of tick) would violate the goal. In KAOS terms, these operations occurrences are those for which the required preconditions for satisfying the goal are not satisfied. Deriving LTS from the required pre-, trigger- and post-conditions as described in Section 4 resolves the problem because the FLTL constraints encoding the semantics of domain pre- and required pre-conditions constrain the occurrences of the forbidden operations between the occurrences of tick.

7 Conclusion

Thanks to our derivation of event-based transition system from goal-oriented operation models, KAOS modellers may use LTSA to formally analyse and animate their operation models. The structure of the goal model provides the scope of the operation model to be analysed, thereby allowing for the analysis of partial models related to specific goals. Conversely, for LTSA modellers, our translation allows them to follow a goal-oriented process to elaborate the behaviour models to be analysed and animated.

The translation can be fully automated with user interactions required only to bound the size of the KAOS model (Section 4.1). In the current state, only the translation from FLTL constraints to LTS is implemented (Section 3). The derivation of fluents and FLTL constraint from KAOS operation (Sections 4.2 and 4.3) has not been implemented.

An important contribution of this work is to provide insights into differences between modelling approaches used in requirements engineering and software architecture analysis, particularly concerning the use of synchronous and asynchronous temporal logics. In Section 6, we have shown that the synchronous nature of KAOS goals is an obstacle to deriving adequate event-based models directly from goals. On the other hand, in [Let05] we have shown that important classes of requirements such as "immediate response" properties and some state invariants that are easily specified in synchronous temporal logic may be extremely difficult to specify correctly in asynchronous temporal logics.

The work reported in this paper has also helped clarifying semantic subtleties of the KAOS operation model, such as the assumption that an operation can occur at most once between two observable states (due to the fact that two consecutive states in a KAOS trace are separated by a *set* of operations) and the possibility of implicit required pre- and trigger conditions due to interferences between concurrent operations.

Based on the insights gained during this work, we believe that a promising approach for future work would be to consider specifying goals in an asynchronous temporal logic instead of the synchronous one. This would provide the benefit of being able to derive adequate LTS models directly from goals. This approach requires adapting the goal-oriented requirements elaboration techniques (such as goal refinement, conflict detection, etc.) currently defined in the synchronous framework to an asynchronous framework. The proposition of Section 3 allowing one to compose LTS derived from asynchronous safety assertions that are closed under stuttering will be much useful in that respect. This will also require finding adequate ways of specifying 'immediate response' properties in asynchronous temporal logic without using assertions that are not closed under stuttering.

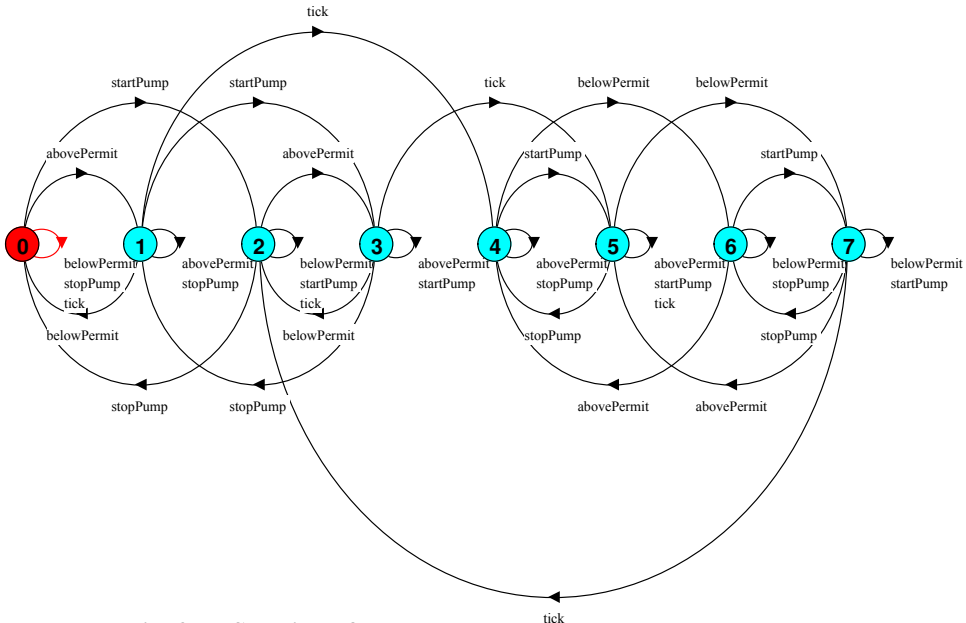

Fig. 3 LTS derived from []($PressureAbovePermmit -> X PumpOn)

ACKNOWLEDGEMENT. The work reported herein was partially supported by the Belgian "Fond National de la Recherche Scientifique" (FNRS) and EPSRC grant READS GR/S03270.

8 References

[All97] R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol. 6, No. 3, pp. 213-249, 97.

[Ber91] M. Bernardo, P. Ciancarini and L. Donatiello, Architecting Software Systems with Process Algebras, University of Bologna, UBLCS-2201-7, July 2001.

[Chu00] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 2000.

[Cim02] 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, *Proc. CAV02*, LNCS 2404, Denmark, July 2002.

[Cou93] P.-J. Courtois and D.L. Parnas, "Documentation for safety critical software", Proc. ICSE'93 - 15th Intl. Conf. on Software Engineering, 1993.

[Dar96] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'96*, San Francisco, Oct. 1996.

[Gia03] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems", in Proc. ESEC/FSE 2003, Helsinki, Finland, Sept. 2003.

[Hei96] C. Heitmeyer, R.D. Jeffords and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications", *ACM Trans. on Software Eng. and Methodology*, Vol. 5 No. 3, July 1996, 231-26.

[Hei98] C. Heitmeyer, J. Kirkby, B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for specifying and Analyzing Software Requirements", *Proc. CAV'98*, Vancouver, 1998, 526-531.

[Jac00] D. Jackson, Automating first-order relational logic, in Proc. FSE 2000, San Diego, Ca, USA, Nov. 2000.

[Kra83] J. Kramer, J. Magee, M. Sloman et al, CONIC: an Integrated Approach to Distributed Computer Control Systems, IEE Proceedings, Part E 130, 1, January 1983, pp. 1-10.

[Lam98] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development*, Nov 1998.

[Lam00a] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, October 2000.

[Lam00b] A. van Lamsweerde, Requirements Engineering in the Year 00: A Research Perspective, *22nd International Conference on Software Engineering*, Limerick, ACM Press, 2000.

[Lam01] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", Invited Minitutorial, *Proc. RE'01 - 5th Intl. Symp. Requirements Engineering*, Toronto, August 2001, pp. 249-263.

[Lam03] 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.

[Let01] E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*, Phd Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, May 2001.

[Let02a] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Press, May 2002.

[Let02b] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *FSE'10: 10th ACM Symp. Foundations of Software Engineering*, Charleston, November 2002.

[Let02c] E. Letier, *Goal-Oriented Elaboration of Requirements for a Safety Injection Control Syste*m. Research Report, Département d'Ingénierie Informatique, UCL, June 2002.

[Lam04] A. van Lamsweerde and E. Letier, From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering, in *Radical Innovations of Software and Systems Engineering in the Future*, M. Wirswing (ed.), Springer-Verlag, LNCS 2941, 2004.

[Let05] E. Letier, J. Kramer, J. Magee and S. Uchitel, Fluent Temporal Logic for Discrete-Time Event-Based Models, *Proc. ESEC/FSE 2005*, Lisbon, Sept. 2005.

[Mag95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, Specifying Distributed Software Architectures, *5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, Sept. 1995.

[Mag99] J. Magee and J. Kramer, *Concurrency - State Models & Java Programs*, Chichester, John Wiley & Sons, 1999.

[Mag00] Magee, J., Pryce, N., Giannakopoulou, D., and Kramer, J. "Graphical Animation of Behavior Models", *in Proc. of the 22d International Conference on Software Engineering (ICSE' 2000)*, Limerick, Ireland, June 2000.

[Pon04] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, H. Tran Van Early Verification and Validation of Mission-Critical System, *Proc. FMICS'04, 9th International Workshop on Formal Methods for Industrial Critical Systems*, Linz (Austria) Sept. 2004.

[Tra04] H. Tran Van, A. van Lamsweerde, P. Massonet, Ch. Ponsard, Goal-Oriented Requirements Animation, *Proc. 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004.