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

Emmanuel Letier e.letier@cs.ucl.ac.uk
Department of Computer Science, University College London and London Software Systems, Gower Street, London, WC1E 6BT, UK

Jeff Kramer, Jeff Magee {jk, jnm}@doc.ic.ac.uk
Department of Computing, Imperial College London and London Software Systems, 180 Queen's Gate, London, SW7 2BZ, UK

Sebastian Uchitel suchitel@dc.uba.ar
University of Buenos Aires/CONICET and Department of Computing, Imperial College London - London Software Systems, Intendente Güiraldes 2160, Buenos Aires, C1428EGA, Argentina

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 reasoning about software behaviour at the architectural level.

The paper relates 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 can be used for the formal analysis and animation of KAOS operation models in LTSA.

The paper also provides insights into the two complementary formalisms, and shows that the use of synchronous temporal logic for requirements specification hinders a smooth transition from requirements to software architecture models.

Keywords Goal-Oriented Requirements Engineering, Labelled Transition Systems, Method Integration, Requirements Animation, Requirements Analysis.

1 INTRODUCTION

Goal orientation is a recognized paradigm for elaborating, structuring and analysing software requirements [Chu00, Lam00b, Lam04, Myl06]. *Goals* are prescriptive statements of intent 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 should be 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 formal analysis 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 [Lam98a] 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 latter can be used for the *automated formal analysis* of software specifications [Che99, Gia99a, Gia03, Mag00] and provides the basis for the downstream activities of software architectural design [Ng96, Gia99b], program verification [Gia04, Gia05], specification based testing, etc.

Our work is motivated by this complementarity. *A first contribution of the paper is to present a technique for transforming goal-oriented requirements models into event-based transition systems*. This transformation requires bridging the gap resulting from the different modelling paradigms that goal-oriented models and event-based transition systems follow: goal-oriented models are declarative, state-based, timed, synchronous models, while event-based transition systems are operational, event-based, untimed and asynchronous [Let05]. *The paper presents an automatic technique for transforming state-based specification of operations, derived from goals according to the KAOS method, into event-based transition systems analysable by the LTSA toolset.* The derived model can be used to carry out formal analysis of KAOS operation models in LTSA: *i)* incompleteness of a KAOS operation model with respect to goals can detected by model-checking the derived event-based transitions system against goals; *ii)* inconsistencies and implicit requirements in a KAOS operation model can be detected as deadlocks; *iii)* animation of the KAOS operation model can be performed using the standard animation features of LTSA.

A second contribution of the paper is to discuss and exemplify limitations of combining the formalisms that are currently in use for goal-oriented requirements engineering and for modelling behaviours at the architecture level. We will show that the gap between the synchronous and asynchronous modelling approaches has an adverse impact on the state space of the generated event-based model, and is an obstacle to deriving event-based models directly from goals, as opposed to deriving them from the operations derived from the goal. These problems raise important questions concerning the choice of formalisms to be used to support a combined elaboration of system requirements and software architecture models and suggest that using an asynchronous temporal logic formalism for modelling goals merits exploration.

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 used in Section 4 to translate KAOS operation models into event-based transitions system. Section 5 describes how to use the derived model for the formal analysis and animation of KAOS operation models. Section 6 presents the application of our

techniques on the safety injection system of a nuclear power plant. Section 7 discusses scalability and limitations of the approach. Related work is discussed in Section 8.

2 BACKGROUND

This section introduces the goal-oriented modelling language and the event-based transition system on which our technique is built. Language constructs are illustrated on a simplified model of the mine pump control system [Kra83]. Water percolating into a mine is collected in a sump to be pumped out of the mine. The purpose of the mine pump control system is to prevent sump overflows. The pump control software is responsible for switching a pump on when the water in the sump reaches some high-level and switching it off when the water goes below some low level. In order to avoid explosion, the pump must also be turned off when there is methane inside the mine. A full goal-oriented model of the mine pump control system is available in [Let01].

2.1 Goal-Oriented Modelling with KAOS

Operational software requirements are derived gradually from the underlying system goals. The word "system" here refers to the software-to-be together with its environment. The derivation proceeds according to the following steps [Lam01].

- *Goal modeling:* A goal refinement graph is elaborated first by identifying relevant goals from input material (such as interview transcripts and available documents) – typically, by looking for intentional keywords in natural language statements, and by asking *why* and *how* questions about such statements in order to identify higher and lower level goals, respectively.
- *Object modeling:* UML classes, attributes and associations are derived systematically from the goal specifications.
- *Agent modeling:* Agents are identified together with their potential monitoring/control capabilities; alternative assignments of goals to agents are explored.
- *Operationalization:* Operations and their domain pre and postconditions are identified from the goal specifications; strengthened pre-, post- and trigger conditions are derived so as to ensure the corresponding goals.

The above steps are ordered by data dependencies and, of course, intertwined. Each step may be guided by the use of heuristics and derivation patterns associated with specific tactics [Dar96, Let02a, Let02b, Let04]. Additional parallel steps of the method handle goal mining from scenarios [Lam98b, Dam06], the management of conflicts between goals [Lam98a], the management of obstacles to goal satisfaction [Lam00a], and the evaluation of alternative system designs with respect to their impact on the degree of goal satisfaction [Let04].

The focus of this section is on presenting those aspects of the KAOS language that are necessary to understand the mapping to event-based transition systems to be presented in Section 4. A more complete introduction to the language and elaboration method can be found in [Lam01] and [Let01].

2.1.1 Specifying Goals in LTL

As introduced before, a *goal* is a declarative property to be satisfied by the system under consideration. An example of a goal for the mine pump control system is the following:

Goal Avoid[SumpOverflow]

Def. The sump should not overflow

FormalDef []¬(WaterLevel ≥ 'Overflow')

This specification fragment introduces a goal named Avoid[SumpOverflow] (Avoid is a keyword that refers to the temporal pattern of the goal). This goal is defined by some natural language statement for communication with stakeholders and by some formal definition expressed in linear temporal logic for formal reasoning.

The formal assertions defining goals are formed from state variables (such as WaterLevel) that correspond to object attributes and relationships in the application domain object model. A system state is a mapping that assigns a value to each state variable. Linear temporal logic assertions are formed from state variables, standard Boolean operators, qualitative linear temporal logic operators X (next), [] (always), <> (eventually), U (until) and W (Awaits) and real-time operators $[]_{\sim d}$ P , $<>_{\sim d}$ P and P $U_{\sim d}$ Q where $\sim \in \{<, \leq, >, \geq\}$ and $d \in$ Nat. An interpretation for an LTL formula is a history *h: Nat* $\rightarrow$ *State* that maps to each position $i \in$ *Nat* the global system state at that position. KAOS models assume a discrete-model of time in which consecutive states in a trace are always separated by a single time unit. The time unit corresponds to some arbitrarily chosen smallest possible time unit for the application domain. The notation (h,i) |= P is used to express that the LTL formula P is true at position i of the trace h. The semantics of the temporal logic operators is defined as follows [Man92, Hen98]:

- (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
- (h,i) |= P *U* Q *iff* (h, j) |= Q *for some* j ≥ i and (h, k) |=P for all k s.t. i ≤ k < j
- (h,i) |= P *W* Q *iff* (h,i) |= P *U* Q *or* (h,i) |= []P
- (h,i)|= $[]_{\sim d}$ P *iff* (h, j)|=P *for all* j ≥ i and |j-i| ~ d
- (h,i)|= $<>_{\sim d}$ P *iff* (h, j)|=P *for some* j ≥ i and |j-i| ~ d
- (h,i)|= P $U_{\sim d}$ Q *iff* (h, j)|=Q *for some* j ≥ i and |j-i|~ d and (h, k) |=P for all k s.t. i ≤ k < j

The Boolean operators have their usual semantics. 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.

2.1.2 Goals Refinement Structures

Goals are organized in AND/OR-refinement structures. *AND-refinement* links relate a goal to a set of subgoals (called *refinement*) and 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 relate a goal to an alternative set of AND-refinements; satisfying one of the refinements is a sufficient condition for satisfying the goal. High-level goals are recursively AND-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. An *assumption* is a goal assigned to an agent in the environment.

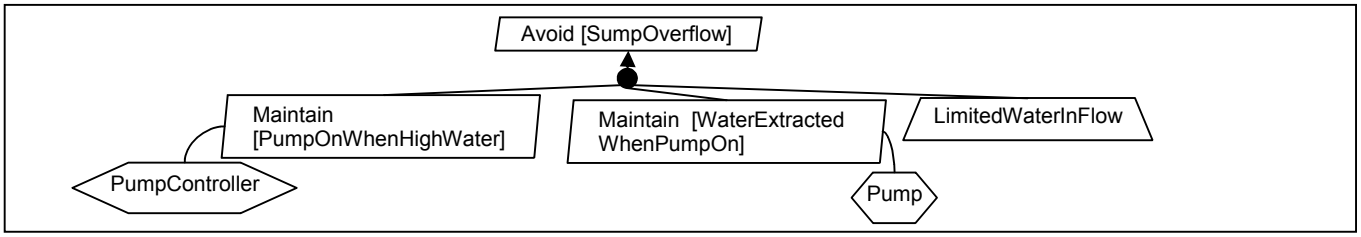

Figure 1. An example of AND-refinement and goal assignments for the pump control system

For example, in Figure 1 the goal Avoid[SumpOverflow] is AND-refined into the subgoals Maintain [PumpOnWhenHighWater], Maintain[WaterExtracted WhenPumpOn], and the domain property Maintain[LimitedWaterInFlow] (depicted by a trapezoid). The first subgoal states that the pump must be on when the water level is high. It is a terminal goal assigned to the mine pump controller and is therefore a requirement. The second subgoal states that when the pump is on water is extracted from the sump. It is a terminal goal assigned to the pump and is therefore an assumption. The domain property states that the maximal flow of water percolating into the mine is always less than the pump capacity to extract water. Formal definitions are skipped here for space reasons. Other requirements, not shown in Figure 1, for the mine pump controller include Maintain[PumpOffWhenLowWater] that is a subgoal of a higher-level goal called Avoid[PumpDamaged], and Maintain[PumpOffWhenMethane] that is a subgoal of Avoid[Explosion]. There is a conflict between the requirements Maintain [PumpOnWhenHighWater] and Maintain[PumpOffWhenMethane]. This conflict is resolved by weakening the first requirement into Maintain [PumpOnWhenHighWaterAndNoMethane] and propagating the change along goal refinement links by weakening the goal Avoid[SumpOverflow]. The result of the goal modelling step is an AND-OR refinement graph relating requirements, assumptions and domain properties to higher-level goals [Let01]. Figure 2 shows the formal definition for the three main requirements assigned to the pump controller agent.

Requirement Maintain[PumpOnWhenHighWaterAndNoMethane]
Responsibility Pump Controller
Def. If the water level is above high and there is no methane, then at the next time unit the pump must be on.
FormalDef [] (HighWater $\wedge \neg$ Methane $\rightarrow$ X PumpOn)

Requirement Maintain[PumpOffWhenMethane]
Responsibility Pump Controller
Def. If there is methane in the mine, the pump must be off within the next time unit.
FormalDef [] (Methane $\rightarrow$ X $\neg$ PumpOn)

Requirement Maintain[PumpOffWhenLowWater]
Responsibility Pump Controller
Def. If the water is below the low level, the pump must be off within the next time unit.
FormalDef [] (LowWater $\rightarrow$ X $\neg$ PumpOn)

Figure 2. Requirements for the mine pump controller

2.1.3 Deriving Operations From Goals

A goal assigned to some agent in the software-to-be is operationalized into functional services, called operations, to be performed by the agent. An *operation* is a relation over states. KAOS operations are specified in a state-based style in the spirit of Z [Woo96], VDM [Jon96], and B [Abr96] with additional language features specific to requirements models [Dar93]. When specifying an operation, an important distinction is made between *domain* pre/postconditions and *required* pre-, post- and trigger conditions necessary for achieving some underlying goal.

- 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 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.

To produce consistent operation models, a required trigger condition on an operation must imply the conjunction of its required preconditions.

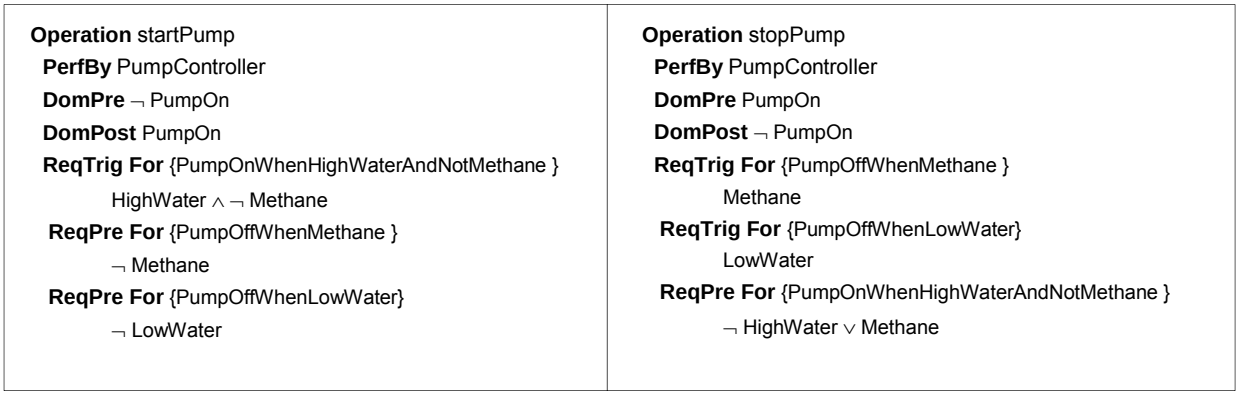

Figure 3. Operation model for the mine pump controller derived from the requirements in Fig. 1

Figure 3 shows the operation model for the mine pump controller derived from the requirements in Figure 2. In this model, the domain pre and postconditions for the operation startPump capture the domain property that every application of this operation corresponds to a transition from a state where the pump is not on to a state where it is on. The required trigger condition captures the requirements that this operation *must* be applied when the water level is high and there is no methane, and that this condition is necessary to satisfy the requirements PumpOnWhenHighWaterAndNoMethane. The two required preconditions on that operation state that this operation *may* be applied only if there is no methane and the water level is not low, and that these conditions are necessary to satisfy the requirements PumpOffWhenMethane and PumpOffWhenLowWater, respectively.

The formal semantics for KAOS operation models is defined 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 that satisfies the domain pre and postcondition of the operation:

$$[]\ (\ [|op|] \leftrightarrow \text{DomPre}\ (op)\ \wedge X\ \text{DomPost}\ (op)\ )$$

The right to left direction in this bi-implication corresponds to a form of frame axiom, similar to the one described in [Bor95], necessary to support an incremental elaboration of operation models from goals. Detailed justification can be found in [Let02b].

The semantics of required pre, trigger and postconditions is defined as follows:

$$[]\ ([|op|] \rightarrow \text{ReqPre})$$

$$[]\ (\text{ReqTrig} \wedge \text{DomPre} \rightarrow [|op|])$$

$$[]\ ([|op|] \rightarrow\ X\ \ \text{ReqPost}\ (op))$$

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 is said to be a *complete operationalization* of a declarative requirement if satisfying all required conditions in the set guarantees the satisfaction of the requirement. Operationalisation patterns have been defined for automatically deriving complete goal operationalization for frequently occurring goal patterns [Let02b]. These patterns have been used in the above mine pump example to formally derive the operation model in Figure 3 from the requirements in Figure 2. The patterns guarantee the completeness of operationalizations while hiding formal reasoning from the requirements engineers. When a goal does not match an existing pattern, the operational requirements have to be derived and verified manually.

2.2 Behaviour Analysis with LTSA

2.2.1 Labelled Transition Systems

Our target event-based framework for modelling and reasoning about behaviours is that of Labelled Transitions Systems [Keller76] that provides a simple formalism for compositional reasoning in architectural context, and is supported by a tool that provides a wide range of analysis and animation capabilities [Mag99]. A Labelled Transitions System (LTS) models a system as a set of concurrent components where each component is characterized by a set of states and by the possible transitions between these states where each transition is labelled by an event. The global system behaviour is the result of the parallel composition of each component LTS so that the components execute asynchronously but synchronize on shared events.

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,
- $q_0 \in Q$ is the initial state.

In this paper we are concerned with the trace semantics of an LTS *M*, i.e. the set of sequences of observable events that the LTS can perform starting in its initial state. Given two LTSs *P* and *Q*, we denote *P*||*Q* the LTS that models their joint behaviour. The joint behaviour is the result of both LTSs executing asynchronously but synchronising on all shared message labels. This means that any of the two LTSs can perform a transition independently of the other LTS, so long as the transition label is not shared with the alphabet of the other LTS. Shared observable labels have to be performed simultaneously. Observability of labels signifies that the LTSs never synchronise on τ since they represent internal behaviour.

The formalism of LTS is untimed. The standard technique for modelling discrete-time systems as LTS consists in including explicit tick events signalling the regular ticks of a global clock to which each timed process synchronizes

[Ros98, Mag99]. When modelling a timed system in this way, one has to ensure that 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. An example of timed LTS is shown in Figure 4. It describes a system in which the events startPump and stopPump alternate and no more than one startPump or stopPump event can occur during a single time unit. This LTS corresponds to the model that our technique will derive from the domain pre and post conditions of the startPump and stopPump operations in the KAOS specification.

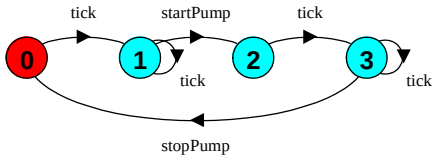

Figure 4. An example of timed LTS

2.2.2 Fluent Temporal Logic

The LTSA toolset has a model-checking feature that allows one to check the satisfaction of temporal logic properties by a given LTS model. 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 $Initially_{Fl}$

For example, fluents defining the state of the pump and whether the water level is above high may be defined as follows:

```
fluent PumpOn = <startPump, stopPump>
fluent HighWater = <raiseAboveHigh, lowerBelowHigh>
```

LTSA supports indexed sets of fluents. For example, the fluent Water[i] may be used to denote that the water level in the sump is i. This fluent could be defined by the following sets of initiating and terminating events:

```
fluent Water[i:0..Max] = <{raise[i-1], lower[i+1]}, {raise[i], lower[i]}>
```

A well formed FLTL formula is an LTL formula whose atomic propositions are fluents. The concrete syntax for FLTL assertions used in LTSA uses the ASCII symbols !, &&, and || for logical negation, conjunction and disjunction, respectively. The syntax for temporal operators is the same as in Section 2.1.1. For example, the assertion [] (HighWater -> X PumpOn) is a well formed FLTL formula.

The semantics of FLTL assertions is defined as follows. Given a set Φ of fluents and an event trace $tr: Nat \rightarrow A$, one can construct a state-based trace $StateTrace(tr): Nat \rightarrow A$ that gives the fluent values after the occurrence of each event in *tr* according to the following rule. 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.* $tr(k) \in Term_{Fl}$

An FLTL formula *P* is then said to be satisfied by an event trace *tr*, noted $tr \models P$, iff $StateTrace(tr) \models P$.

Finally, FLTL assertions can also 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 system 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.

2.2.3 From Synchronous to Asynchronous Temporal Logic

The sequences of states used to interpret FLTL assertions are different from the sequence of states used to interpret KAOS goals and operation models. 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. Zero, one or more events may occur between consecutive states separated by a single time unit. We refer to *synchronous temporal logics* those that are interpreted over sequences of states observed at a fixed time rate, and as *asynchronous* those that are interpreted over sequences of states observed after each occurrence of an event [Let05a].

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'. Confusion between the two variants of temporal logic may lead to important errors in the formal specification of system properties.

Consider for example the property requiring that when the water level is high, the pump must be on *at the next time unit*. In synchronous temporal logic, this property is specified as []($HighWater \rightarrow X\ PumpOn$). The same assertion in asynchronous temporal logic has a different meaning and does not specify the required property correctly because the X operator means 'after the next event' instead of 'at the next time point'. In asynchronous temporal logic, this assertion does not impose any time bound on when the pump must be on. It is also not closed under stuttering and should therefore be avoided as its satisfaction is not preserved by refinements or extensions of the event-based models. The synchronous interpretation of this assertion on the other hand is closed under stuttering, in the sense that adding silent events between time points does not change the satisfaction of the formula [Let05a].

Specifying immediate response properties in asynchronous temporal logic is not trivial. Consider again the property requiring the pump to be turned on immediately when the water level becomes high. It may come as a surprise that the asynchronous assertion [] ($HighWater \rightarrow PumpOn$) does not correctly capture this property, because, contrary to what is required, this assertion prevents the occurrence of an event that makes HighWater true from a state where HighWater and

PumpOn are both false, i.e. it prevents water to rise above high if the pump has not previously be turned on. The problem is due to the fact that in an asynchronous trace with interleaving semantics, the event startPump may not occur concurrently with changes in water level. In a synchronous temporal logic however, such immediate response property can be specified easily as []($HighWater \rightarrow$ PumpOn) or [] (HighWater $\rightarrow$ X PumpOn), depending on whether the response (PumpOn) has to occur within the same or within the next time frame as the triggering condition (HighWater), respectively.

In [Let05], we have defined an encoding of synchronous temporal logic into asynchronous temporal logic. The translation function $Tr: FLTL_{Sync} \rightarrow FLTL_{Async}$ is defined recursively as follows:

Tr([] P)	=	[] (tick -> Tr(P))
Tr(<> P)	=	<> (tick $\wedge$ Tr(P))
Tr(P U Q)	=	(tick -> Tr(P)) U (tick $\wedge$ Tr(Q))
Tr (X P)	=	X ($\neg$ tick W (tick $\wedge$ Tr(P)))

Boolean operators remain unchanged (i.e. Tr(not P) = not Tr(P), etc.).

For example, this encoding translates the synchronous temporal logic assertion []($HighWater \rightarrow$ X PumpOn) into the following asynchronous one:

```
[](tick -> (HighWater -> X(!tick W (tick && PumpOn))))
```

This mapping allows LTSA modellers to use a goal-oriented requirements elaboration process à la KAOS for the incremental identification, elaboration and specification of formal properties expressed in synchronous temporal logic to be model checked in the LTSA toolset. With this approach, 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. In this paper, we present a technique for automatically deriving these fluent definitions and LTS models from KAOS operation models.

3 FROM FLTL ASSERTIONS TO LTS MODELS

Our procedure for transforming KAOS models into LTS will rely on the following more general technique for transforming FLTL assertions into LTS.

The 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 that do not reach the ERROR state satisfy the assertion. Removing the error states and the transitions leading to it from a property LTS associated to an assertion ϕ yields an LTS model that captures all traces on the alphabet of ϕ that satisfy ϕ. When the assertion is not a safety property, it is not possible to generate an LTS whose behaviours are equivalent to the ones accepted by the assertion.

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.

For example, Figure 5 shows the LTS model generated from the synchronous temporal assertion [](HighWater -> X PumpOn). This LTS model accepts the sequence of events <aboveHigh, tick, startPump, tick>, but does not accept the sequence <aboveHigh, tick, tick>. In the initial state (state 0), HighWater and PumpOn are both false. In state 4, no tick is allowed to occur because HighWater is true at the last occurrence of tick and PumpOn is currently false.

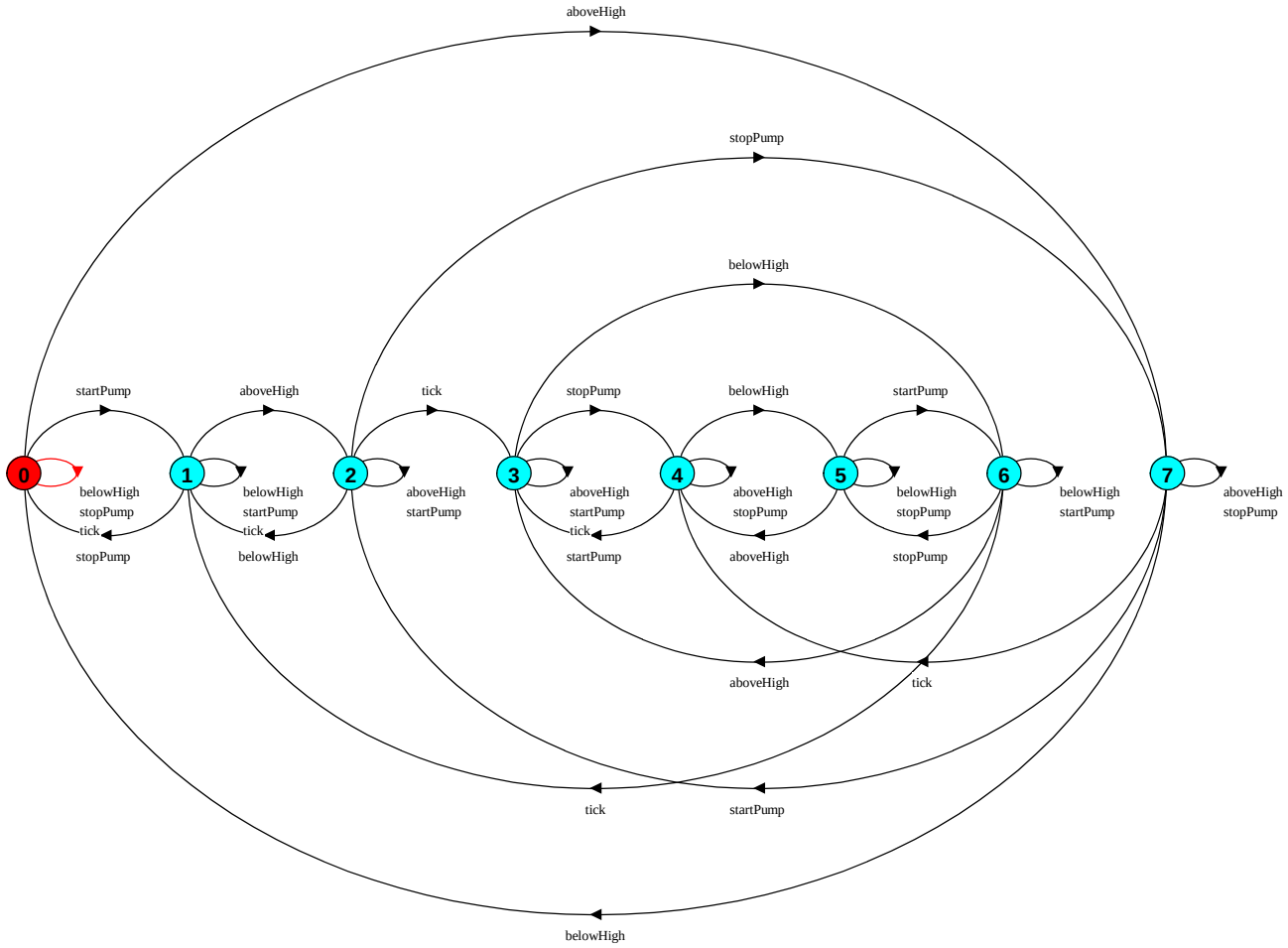

Figure 5. LTS derived from [](HighWater -> X PumpOn)

In order to compose LTSs derived from FLTL assertions in a meaningful way, it is desirable that the composition of two "constraint LTSs" is equivalent to logical conjunction. The following proposition establishes that this is the case for formulae that are closed under stuttering. 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.

Proposition. Let P, Q be two safety properties in FLTL. If P and Q are closed under stuttering, then for all traces tr, tr |= P ∧ Q if and only if tr is accepted by (constraint P || constraint Q).

Proof. We have to show that a trace tr satisfies P ∧ Q if and only if it is accepted by (constraint P || constraint Q). Let *tr* be an event trace. We have that *tr* |= P ∧ Q if and only if *tr* |= P and *tr* |= Q. Consider that *tr* |= P. If we replace in *tr* every occurrence of an event not in the alphabet of P by a silent action τ, the values of the fluents involved in P remain unchanged in *tr* and the resulting trace still satisfies P. Since P is closed under stuttering, removing these

silent actions from the trace does not change the satisfaction of P. Similarly, if *tr* |≠ P, the resulting trace still violates P. We therefore have that *tr* |= P iff $tr_{|alpha(P)}$ |= P, where alpha(P) denotes the set of all initiating and terminating events for the fluents appearing in P and $tr_{|alpha(P)}$ denotes the projection of tr on that alphabet . By definition of the constraint construct, we also have that $tr_{|alpha(P)}$ |= P iff $tr_{|alpha(P)}$ is accepted by the LTS 'constraint P'. Therefore, *tr* |= P iff $tr_{|alpha(P)}$ is accepted by the LTS 'constraint P'. Similarly, since Q is closed under stuttering, we have that *tr* |= Q iff $tr_{|alpha(Q)}$ is accepted by the LTS 'constraint Q'. On the other hand, the definition of the parallel composition operator implies that *tr* is accepted by (constraint P || constraint Q) iff $tr_{|alpha(P)}$ is accepted by 'constraint P' and $tr_{|alpha(Q)}$ is accepted by the LTS 'constraint Q'. We have therefore established that *tr* |= P ∧ Q if and only if *tr* is accepted by (constraint P || constraint Q).

In the following sections, the FLTL assertions that we translate into LTS are assertions encoding the semantics of the KAOS operation model and are all safety properties that are closed under stuttering.

4 FROM KAOS OPERATIONS TO LTS

We now assume that a KAOS goal model has been transformed into an operation model according to the technique described in [Let02b]. The procedure for transforming a KAOS operation model into a LTS is composed of the following 5 steps.

Step 1. Bounding the KAOS model and identifying fluents

The first step consists in transforming a KAOS model that has possibly an infinite state space into a finite "fluent-based" KAOS model. The result of this step is a finite state KAOS model in which all variables correspond to LTSA fluents and all assertions (i.e. the pre-, post-, and trigger conditions) correspond to FLTL assertions on these fluents.

This step is performed as follows.

- For each KAOS *state variable with an infinite range*, the analyst has to choose some finite discrete range for that variable. For example, a real-valued variable WaterLevel could be restricted to integer values in the range [0..14]. First-order KAOS models in which state variables correspond to attributes and relationships of an object model can be bounded as well by limiting the maximum number of instances of each entity class. For example, a fragment of an object model for a library system composed of the classes User and BookCopy and the Borrowing relationship between them would yield the following declarations:

  ```
  range User = 0..MaxUser
  range BookCopy = 0..MaxBookCopy
  fluent Borrowing[u:User, b:BookCopy]
  ```

  where MaxUser and MaxBookCopy are constants, chosen by the analyst. After this step, all state variables in the KAOS model have a finite range and the following steps can be performed automatically.

- Each *Boolean state variable* in the KAOS model is mapped to an equivalent fluent. For example, the state variable PumpOn in our running example yields the same fluent in LTSA.

- Each *finite range integer variable or enumerated variable* is mapped to a parameterized fluent where the fluent parameter records the state variable value. For example, an integer state variable WaterLevel whose values are in the integer range 0..14 yield the following fluent declaration:

  fluent WaterLevel[i: 0..14]

  KAOS expressions involving proposition of the form "Var = x" where Var is a integer or enumerated variable and x is an integer or enumerated value are rewritten in the fluent-based KAOS model as "Val[x]". An expression of the form "Var ~ x" where Var is an integer variable of finite range R, x is an integer, and $\sim \in \{<, \leq, >, \geq\}$ is rewritten in the fluent-based KAOS model as "exists [i:R] Val[i] && (i ~ n) ". For example, the expression of the form "WaterLevel≤2" in the KAOS model is rewritten as "exists [i:0..14] WaterLevel[i] && (i ≤ 2) ". When processed by LTSA, such expression is automatically transformed into "WaterLevel[0] || WaterLevel[1] || WaterLevel[2]" .

The bounding technique described here may be seen as a very simplified version of the technique used in Alloy to transform object models into propositional temporal logic [Jac00]. However, we assume a much simpler (and less expressive) object model than the ones handled in Alloy so that the mapping from KAOS object models to fluents is straightforward and allows fluents to be easily interpreted as KAOS state variables. Therefore we do not need to provide an explicit mapping from fluents back to KAOS state variables.

Step 2. Deriving fluents defintions

The next step consists of deriving the initiating and terminating events for each fluent. Each operation in the KAOS model yields a corresponding event in the LTS model. For example, the KAOS operation startPump is associated to an LTS event with the same label. If the KAOS operation has parameters, the same parameters are used in the LTS events. For example, the operation borrow[u:User, b:BookCopy] of a library system is mapped to the corresponding parameterized LTS event.

The following rule is then used to automatically derive the initiating and terminating events of each fluent from the domain postconditions.

> **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 domain postcondition of the operation.

This rule relies on two assumptions. Firstly, it assumes that domain postconditions are given as conjunctions of positive and negative occurrences of fluents, i.e. disjunctions are not allowed in domain postconditions. Since domain pre- and postconditions correspond to elementary state transitions, this assumption is generally satisfied by KAOS operations. If this assumption is not satisfied, our derivation process cannot be applied. Secondly, we assume 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 a simple pre-processing step that replaces a domain postconditions in which a fluent appears both positively and negatively by 'false'.

For example, the domain postconditions for the operations startPump and stopPump in Figure 3 yields to the following fluent definition:

```
fluent PumpOn = <startPump, stopPump>
```

In most cases, KAOS operation models represent only the operations performed by software agents. In order to derive initiating and terminating events for fluents whose values depend on the occurrence of operations performed in the

environment, such operations need to be specified as well. For example, the KAOS model of the mine pump control system needs to contain operations specifying the changes in water level such as the following:

Operation aboveHigh

DomPre ¬ HighWater ∧ ¬ LowWater

DomPost HighWater

The operations belowHigh, aboveLow and belowLow are specified in a similar way. The expected initiating and terminating events for the fluents HighWater and LowWater are then automatically derived from these operations.

Step 3. Translating the domain preconditions

KAOS domain preconditions are translated according to the following template:

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

The FLTL assertion 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 event. 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.

The FLTL assertion corresponds to the semantics of domain precondition in the KAOS model. The correctness of the FLTL assertion however relies on an addition LTS process that ensures that every trace starts with a tick. Furthermore, in a KAOS trace, two successive states are separated by a *set* of events, i.e. each event may occur at most once between two states and the occurrences of events between to states are not ordered. We encode this in our model with an additional FSP processes constraining each fluent to change value at most once between two ticks.

For our mine pump example, Figure 4 shows the LTS model obtained by combining the processes derived from the domain pre and post conditions of the operations startPump and stopPump in Figure 3 and the additional constraints derived from the nature of KAOS traces.

Step 4. Translating the required pre-, trigger-, and post-conditions

The rules for translating required pre/trigger/post conditions are as follows:

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 rules encode in FLTL the temporal logic semantics of the KAOS operation models given in Section 2.1.2. All assertions expressed in these three rules are safety properties and are closed under stuttering. The LTSs derived from these assertions can therefore be combined through parallel composition as described in Section 3.

For example, the required pre condition for the goal PumpOffWhenMethane on the startPump operation, i.e. '! Methane', yields the following constraint:

constraint ReqPRE_On_startPump_For_PumpOffWhenMethane = [](tick && ! ! Methane) -> X(! startPump W tick))

Figure 6 shows the LTS generated from this constraint. As required by the KAOS semantics, this LTS prevents startPump to occur in states 2 and 3, because in these states Methane was true at the last occurrence of a tick.

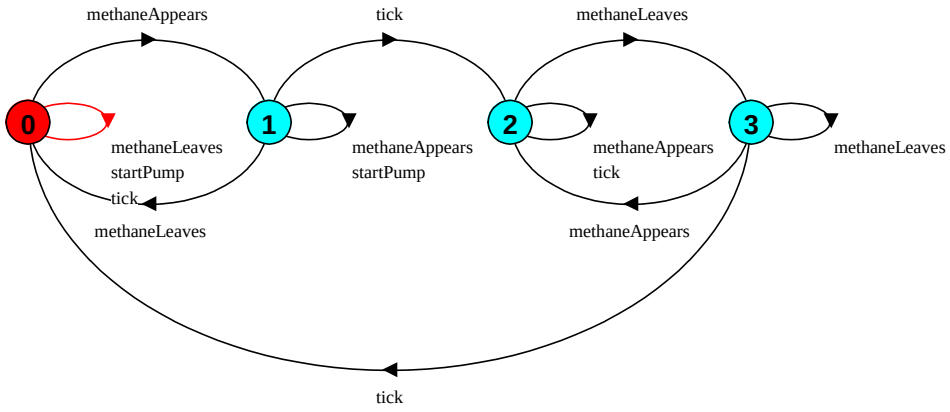

Figure 6. LTS derived from the required precondition on the operation startPump for the goal PumpOffWhenMethane

Step 5. Model composition

The LTS model for the software-to-be is then obtained by composing all LTSs derived from the domain and required conditions on its operations. For the mine pump controller, this involves composing the 2 LTSs derived from its operations' domain preconditions with the 6 LTSs derived from its required pre and trigger conditions. The derivation of these LTSs from constraints, their composition, and minimization takes less than a second on a standard computer (3 Ghz, 512 Mb). The resulting LTS model has 65 states. Complexity and scalability issues for larger models are discussed in Section 7.

5 LTSA Analysis of KAOS Operation Models

The derived LTS model can now be used to carry out formal analysis of the KAOS operation model using LTSA.

5.1 Checking Goal Operationalization and Higher-level Goals Satisfaction

When a KAOS operation model has not been derived from requirements using pre-proved operationalisation patterns, it is useful to verify formally whether a set of required pre-, trigger- and post-conditions {R1, ..., Rn} forms a complete operationalization of a requirement G. This can be done using the FLTL model-checking feature of LTSA by verifying whether the translation in FLTL of the KAOS requirement G is satisfied by the LTS derived from the required condition R1, ..., Rn and the LTSs derived from the domain preconditions of all operations involved. Note that only the portion of the LTS model necessary for the verification is generated. As usual, if the KAOS model has been bounded, the completeness of the goal operationalization in the bounded scope does not ensure that the operationalization is complete for an unbounded scope. If the operationalization is not complete, the tool generates an error trace.

For illustration purpose, suppose a mine pump model in which the requirement PumpOnWhenHighWaterAndNoMethane is operationalized by the required trigger condition "HighWater $\wedge \neg$ Methane" on startPump alone, i.e. the required precondition "$\neg$ (HighWater $\wedge \neg$ Methane)" on operation stopPump shown in Figure 3 has been omitted. We know from experience that omitting this required precondition is a frequent error when operationalizing similar goals without the use of pre-proved operationalization patterns. In order to check the goal operationalisation in LTSA, we derive a LTS model composed of the required trigger condition on startPump, and of the domain preconditions of the pump controller's operation. We then

model-check the derived LTS against the declarative requirements. The tool detects that the requirement is not satisfied and generates the following counter-example:

tick	
startPump	
aboveHigh	
tick	**HighWater && PumpOn**
stopPump	
tick	**HighWater**

The error trace is a sequence of events annoted with fluents that hold at every occurrence of tick. Since KAOS goals are synchronous assertions, the satisfaction of a goal depends only on fluents values when tick occurs. This error trace shows that the requirement is violated if the operation stopPump is performed from a state where HighWater and PumpOn hold, thereby indicating a missing required precondition on that operation. Note that, thanks to our simple mapping from KAOS state variables and operations to LTSA fluents and events, error traces generated by LTSA are easily interpreted as KAOS traces without the need to explicitly map back LTSA traces to KAOS traces.

In addition to checking a single goal operationalisation, the model-checking feature of LTSA may also 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. This allows one to verify with a single check the global correctness of a whole portion of a goal refinement graph and operationalisation. This will be illustrated on the case study in the following section.

5.2 Checking for Consistency and Implicit Requirements

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 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 both must be taken and may not be taken. As another example, an inconsistency may arise if two operations have domain postconditions that are logically inconsistent and have domain preconditions and required trigger conditions that can be true at the same time. In this case, the system may be in a state in which both operations must be taken, leading to an inconsistency because there is no next state that would satisfy both of their domain postconditions. Even if complete operationalisation patterns are used to derive the operation model, inconsistencies may occur if the operation model is derived from conflicting goals [Lam98a]. It is therefore important to be able to detect inconsistencies in operation models.

Our translation ensures that all inconsistencies in the KAOS operation model correspond to deadlock in the derived LTS model. To illustrate this, consider the following specification of the startPump operation whose required trigger condition differs from the one in Figure 3 by not requiring Methane to be false.

Operation startPump

...

ReqTrig For {PumpOnWhenHighWater}

HighWater

ReqPre For {PumpOffWhenMethane}

¬ Methane

This specification fragment is inconsistent because the required trigger condition does not imply the required precondition. This is automatically detected by LTSA that generates the following deadlock trace shown here up to the last occurrence of tick:

tick, aboveLow, **tick**, aboveHigh, methaneAppears, **tick**.

This trace shows a sequence of events leading to a state in which HighWater and Methane are both true, so that the required precondition prevents the occurrence of startPump before the next tick and the required trigger condition prevents the occurrence of tick until startPump has occurred.

Conversely, all deadlocks in a LTS derived from a KAOS operation model correspond either to an inconsistency, or to an implicit required preconditions in the KAOS model. Implicit required conditions in KAOS operation models are due to interactions between requirements on different operations. For example, a trigger condition on one operation may implicitly prevent another operation from being applied even if all required precondition on this operation are true. An operation is said to have an implicit required precondition when the actual condition in which the operation is allowed to occur is stronger than the conjunction of all its stated domain and required preconditions.

As an example, suppose that the reactor of a nuclear power plant has 4 states: Off, Starting, On, and Stopping. Among the operations that change the reactor state, consider the following two:

Operation stabilizeReactor
DomPre Starting
DomPost On ∧ ¬ Starting

Operation stopReactor
DomPre On ∨ Starting
DomPost Stopping
ReqTrig SafetyInjection

In this specification fragment, the operation stabilizeReactor has for implicit required precondition the condition "¬ SafetyInjection" because in a state where Starting and SafetyInjection are true, the operation stopReactor must occur, and because stopReactor and stabilizeReactor cannot occur at the same time, stabilizeReactor may not occur in such a state.

Implicit required conditions cause deadlocks in our derived LTS model because our derivation procedure does not include explicit constraints for them. Because the derived LTSs do not prevent an event from occurring when one its implicit required precondition does not hold, if such an event is taken form such state, the required conditions on other operations that cause the implicit requirement can no longer be satisfied at the next occurrence of tick, thereby preventing further occurrence of tick and causing the deadlock.

The presence in our derived LTS of deadlocks caused by implicit required precondition has both a positive and negative side. On the negative side, the need to make implicit requirements explicit in order to generate deadlock free LTSs involves additional work for modellers. On the positive side, being able to detect these implicit requirements and providing guidance for making them explicit is useful when moving towards an implementation of these operations. This

issue is reminiscent of implicit preconditions in Z operation schemas caused by state invariants, and the benefits of eventually making these preconditions explicit, notably in order to obtain more robust specification [Woo96].

The derivation of the implicit requirements could be automated for some restricted simple cases (for example above when an implicit required precondition on an operation op is due to a required postcondition on another operation that corresponds exactly to the domain postcondition on op). The general case however is much more difficult to handle.

5.3 Goal-Based Animation

Our translation also allows one to animate KAOS operation models using the animation features of the LTSA toolset [Mag00]. The animator can be used to explore model behaviours interactively with stakeholders, or replay error traces generated during formal analysis. The animation can be visualized as textual sequences of events (as shown in the paper) or as a graphical animation of a domain scene such as the one for the mine pump control system depicted in Figure 7. This animation scene had first been developed to animate a FSP specification of the mine pump control system, and has been reused without modification to animate the LTSs model derived from the KAOS specification.

The animation of goal-derived LTS enjoys all benefits of goal-based animations [Hun04]:

- it is possible to animate partial operation models associated to specific goals; i.e. goals provide the scope of the animation,
- 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 the "property LTS" derived from goal definitions as described in Section 3.

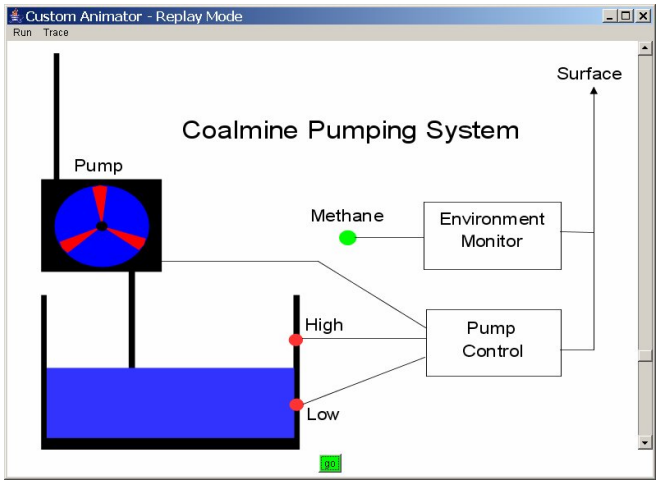

Figure 7. Graphical animation of the Mine Pump System

6 THE SAFETY INJECTION CASE STUDY

We now validate our techniques on the safety injection system of a nuclear power plant. 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 [Cou93, Hei96]. 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 block a safety injection signal 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'.

6.1 Elaborating the KAOS model

A goal-oriented elaboration of the goals and requirements for this system can be found in [Lam02]. A high-level goal of the system is to avoid explosion due to loss of coolant. The elaboration of requirements for the ESFAS software agent involves refining this high-level goal, managing conflicts between goals and identifying and resolving exceptional conditions, called obstacles, that may prevent goal satisfaction. Figure 8 shows the operation model for the ESFAS software component resulting from this process. In this model, the ReqPre/Trigger keyword is used to model conditions that are both required pre- and required trigger conditions for a given goal.

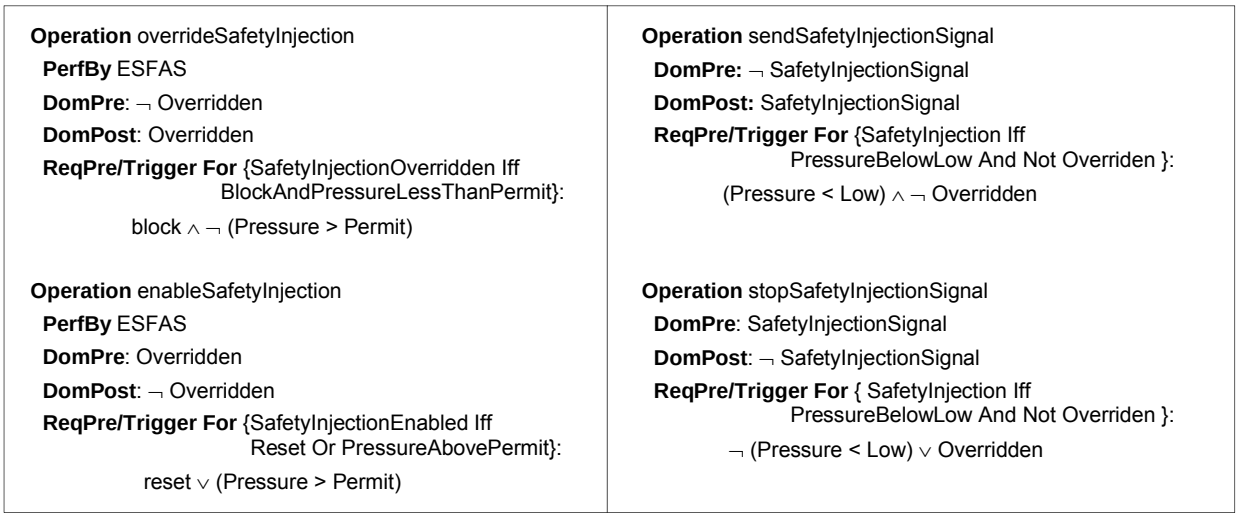

Figure 8. Operation model for the ESFAS software component

In order to derive initiating and terminating events for fluents whose values depend on the occurrence of operations performed in the environment, we have extended this model with the environment operations shown in Figure 9. (These operations are specified here in terms of the "fluent-based" KAOS operation model that would be obtained at the end of Step 1 in Section 4). The goal refinement graph and obstacle model provided guidance for identifying and specifying these operations. For example, the operations explode and leakAppears were identified from the high-level goal Avoid[Explosion] and obstacle LossOfCoolant, respectively. The model also includes operations describing how the nuclear reactor changes states, and how the steam pressure varies according to the state of the reactor and the presence or not of a leak in the cooling system. The DomPre/Trig keyword denotes necessary and sufficient conditions for the operation to be applied in the application domain. Domain trigger conditions are translated in LTS according to the same schema as required trigger conditions.

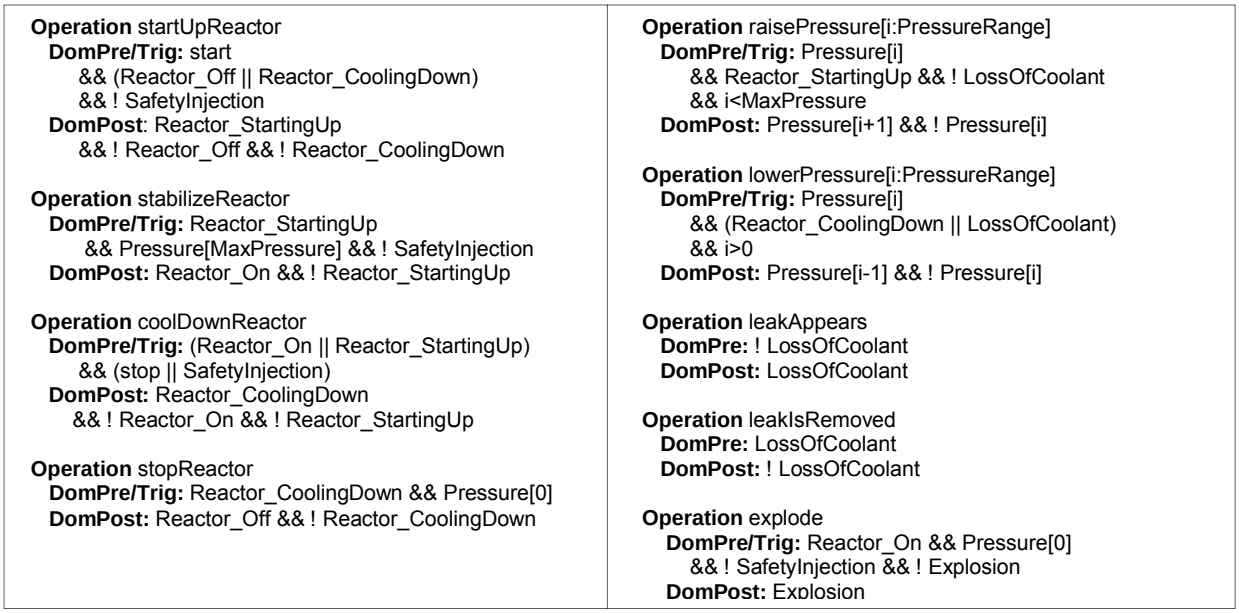

Figure 9. Operation model for the ESFAS environment

6.2 Formal Analysis using LTSA

Our translation technique was applied to derive an LTS model from the KAOS operation model. The 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 LTS to derive the event-based model for this component. The model of the environment has another 9 operations. With the set PressureRange taking values between 0 and 6, generating the full LTS model requires translating and composing 52 FLTL constraints. This takes less than 10 seconds on a standard desktop computer (3Ghz, 512 Mb). The generated model, composed of the software and its environment, has around 80 000 states, and can successfully be animated and formally analysed in LTSA. An animation scene was built to visualize system behaviours and error traces.

During the elaboration of the operation model for the environment, the detection of deadlock in the LTS model derived from the operations proved to be very helpful to detect subtle inconsistencies in our specification of the reactor's operations. For example, our initial specification for the operation startUpReactor did not include the clause "&& ! SafetyInjection" in its domain pre- and trigger conditions. This resulted in an inconsistency with the specification of the stopReactor operation. The inconsistency was revealed by a deadlock trace showing that in a state where start, Pressure[0], and Reactor_CoolingDown are true, there is no successor state in the KAOS model because both operations must be taken but would lead to inconsistent states. A similar inconsistency was detected and corrected between the operations stabilizeReactor and coolDownReactor. These inconsistencies would have been extremely difficult to detect without tool support.

Once we had a consistent operation model of the software system and its environment, we have successfully checked that the ESFAS operation model form a complete operationalisation of its declarative requirements.

An interesting result was obtained when we checked whether the operation model of ESFAS component and its environment satisfied the higher-level goal Avoid[Explosion], as the tool generated the error trace shown in Figure 10. This trace shows that an explosion will occur if a leak occurs when the pressure is above 'Permit' (level 4) and the operator pushes the block button just when the pressure reaches 'Permit'. The tick events in a LTS model derived from a KAOS specification denote time points at which the system state is observed. Because the KAOS model of the safety injection system does not use real-time operators, the significance of tick events in this counter-example is to delineate sets of events that occur between two KAOS states, rather than expressing precise timing for the events.

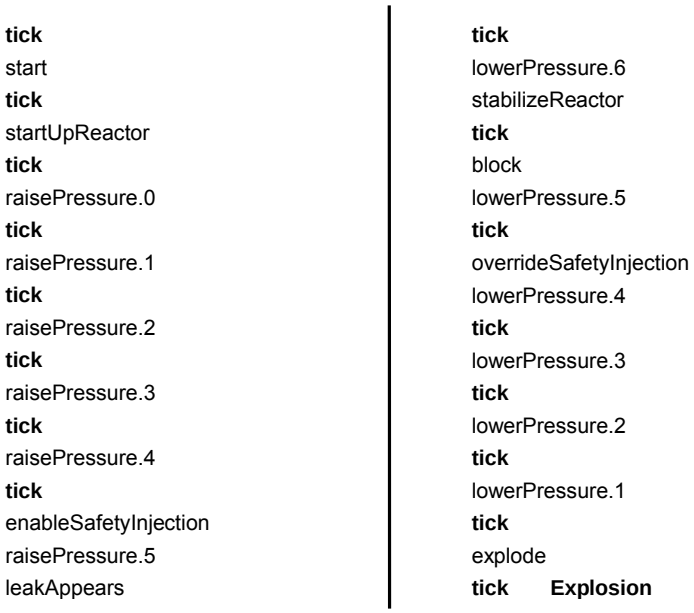

Figure 10. Error trace leading to Explosion

In the goal model of [Lam03], the behaviour depicted in this error trace is ruled out by an assumption on the operator agent. In our operation model however, we allowed all possible operator behaviours so as to obtain a more realistic animation in which the behaviour of the operator is played by a human during the animation rather than constrained by the model.

The importance of the detected problem needs to be checked with application domain experts and stakeholders. At the very least, it raises questions about whether assumptions on the operator behaviour in the goal model that prevents this problem form occurring are adequate and whether a safer system design that eliminate or reduce the risks of human errors could be proposed.

It is worth noting that the SCR specification of the ESFAS component [Hei96] has the same behaviour as the one described in our KAOS model and therefore contains the same potential problem. This problem however cannot be detected by analysing the SCR model, because this model only describes the behaviours of the ESFAS system at the interface with its environment, and the problem here is due to possibly inadequate assumptions about behaviours in the environment that are not at the interface with the automated system. This illustrates the well-known fact that many errors in requirements specifications are caused by inadequate assumptions about the environment in which the software will

operate [Lev95, Kni02], and reinforces the importance of modelling and reasoning about the environment further than at its immediate interface with the automated system [Jac95, Lam02].

7 DISCUSSION

7.1 Implementation and Scalability

Except for the bounding step requiring modellers' intervention, our procedure for transforming KAOS operation models into LTSs can be fully automated. We have provided an implementation for Steps 4 and 5 that are the most computationally expensive steps. Steps 1 to 3, whose automation requires integration with a KAOS model editor such as Objectiver [Obj07], are currently performed manually.

The resource-consuming steps in our derivation are the translation of FLTL assertions to Buchi automaton and the parallel composition of all derived automata. The complexity of translating each assertion is exponential in the size of the FLTL formula. Since each formula corresponds to a single required condition on an operation, it remains small enough to be handled by the tool. Through experimentation, we identified that this translation fails to complete when a required condition is composed of more than 6 fluents, a case we have not yet encountered in real KAOS operation models. Another source of scalability problems comes from the composition of all LTSs derived from the required conditions. Even if the final state space of the complete system would be manageable for LTSA, the intermediate state space generated during parallel composition might exceed the capacity of the tool. This is only a problem if one wishes to obtain an explicit representation of the parallel composition. Such explicit representation is not needed for the animation and model-checking features of the tool. Furthermore, as it has been illustrated above, 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.

7.2 State-space of the Timed Asynchronous Model

An inconvenient of the approach described in this paper is that in order to be semantically equivalent to the synchronous KAOS models, the derived event-based models need to refer explicitly to timing events. This results in models whose state spaces are much larger than untimed LTS models for the same systems. For example, the LTS model for the mine pump controller derived from the KAOS operation model in Figure 3 has 65 states, whereas the LTS model derived from an untimed FSP model that had been previously elaborated for the same system has only 15 states. This problem is inherent to the difference between the synchronous modelling approach used in goal-oriented requirements engineering and the asynchronous approach used in event-based modelling of software architectures.

If the only objective of our work was to provide formal support for analysing KAOS models, this inconvenient could be avoided by mapping KAOS models to synchronous state-based formalisms such as the one used in NuSMV [Cim02]. This is indeed the direction that has been followed in [Pon04]. However, our main objective in this paper is to combine well-established techniques for goal-oriented requirements engineering and for software architecture modelling and analysis. In this paper shows, we show that it is possible to map goal-oriented synchronous specifications of operations to event-based transition systems such as those used for architecture modelling, but only at the cost of obtaining timed transition models with larger state spaces than those that would be obtained from untimed asynchronous models.

7.3 Deriving LTS from Goals

During this work, we have also studied the possibility of deriving LTS directly from goals instead of deriving them from operations. 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 operationalizing the goals. Each KAOS goal would then be translated into a LTS using the technique described in Section 3.

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.

For example, consider again the synchronous assertion [] (HighWater -> X PumpOn) whose LTS is shown in Figure 5. In this LTS, the domain events startPump, stopPump, aboveHigh, and belowHigh are allowed in every states. The tick event is not allowed to occur in states 4 and 6, which are states for which PumpOn is currently false and HighWater was true at the last occurrence of tick (in the initial state, HighWater and PumpOn are both false). An occurrence of tick from these states would therefore violate the assertion.

Because goals have to be satisfied in the context of domain properties, it make sense to combine goal-derived LTSs with the LTSs derived from the operations domain precondition and the KAOS semantic constraint described in Step 3 of Section 4. Unfortunately, the resulting LTS contains deadlocks even if the KAOS goal from which it is derived is consistent with the domain preconditions. These deadlocks are due to the fact that the goal-derived LTS prevents the occurrences of a tick if the system is in a state that would violate the goal, but does not prevent the occurrences of domain events that lead to such states. Domain preconditions and the semantic constraint that an operation may occur at most once between two ticks prevent the system from performing events that would put the system back into a state where tick is allowed. To illustrate this, consider an execution <startPump, aboveHigh, tick> in the LTS of Figure 5 leading to state 3 where PumpOn and HighWater are both true. From that state, the LTS does not prevent stopPump from occurring despite the fact that the goal requires PumpOn to be true at the next occurrence of tick. An occurrence of stopPump leads the system in state 4 in which tick cannot occur because it would violate the goal. When this LTS is combined with the LTSs obtained from KAOS domain precondition and semantic constraints, the system is in a deadlock situation because the operation startPump that would put the system back in a state where PumpOn is true and allow time to progress cannot be applied because its domain precondition was not satisfied at the last occurrence of tick.

In order to avoid such deadlocks, the goal-derived LTSs should prevent a domain operation to occur 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 the occurrence of operations that the synchronous goal does not allow to occur between the current KAOS state (i.e. at the last occurrence of tick) and the next KAOS state (the next occurrence of tick). In KAOS terms, these operations are those for which the required preconditions for satisfying the goal are not satisfied. The LTS derived from the required pre-, trigger- and post-conditions according to the technique described in Section 4 do not exhibit these deadlocks, because the FLTL constraints encoding the semantics of domain pre- and required pre-conditions prevent the occurrences of the domain event that would lead to such deadlocks.

In conclusion, the synchronous temporal logic used in KAOS and other goal-oriented requirements engineering approaches for defining goals is a severe obstacle to the derivation of deadlock-free LTS models directly from goals. This hinders to some extent a smooth transition from modelling approaches used in requirements engineering to those used for software architecture analysis and suggests that adoption of asynchronous logics may be more suitable for describing systems goals.

8 RELATED WORK

Significant amount of research has been devoted to goal-oriented requirements engineering approaches. In addition to the KAOS approach, a notable body of work has been developed around the Tropos framework [Cas02, Bre04]. The formalism that underpins formal modelling of goals in Tropos is also based on synchronous state-based temporal logic [Fux04]. Consequently the results and insights described in this paper are likely to be directly applicable to the Tropos framework.

In [Del04], we have investigated the possibility of translating KAOS operation models into event-based tabular specifications analysable by the SCR* toolset [Hei96, Hei98]. However, a semantic incompatibility between the KAOS and SCR 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.

In terms of support for animation and model-checking the completeness of KAOS operation models, another prototype tool has been developed [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 is performed by translating KAOS models into the symbolic model-checker NuSMV [Cim02]. In this paper, we follow a different approach by translating KAOS models into event-based transition systems with the aim of combining techniques used for goal-oriented requirements engineering and for event-based analysis of software architecture models.

Relating system requirements and software architecture is a critical activity of software engineering for which little systematic support is currently available [Lam00b, Nus01, Cas01, Ber03]. Many efforts for bridging the gap between the requirements and architecture have considered a goal-oriented approach (e.g. [Gro01, Liu01, Bra01, Cas02]). Among them, [Lam03] and [Jan05] describe two different processes for deriving architectural designs form goal-oriented requirements models expressed in KAOS. These processes are mainly concerned with providing guidance for identifying appropriate software structures to meet the various functional and non-functional requirements of the system. The precise specification of components behaviours and issues concerning the integration of the formal languages used at the requirements and architecture levels are not considered. In contrast, in this paper, we have studied the relationships between the formal languages used to describe behaviours at the requirements and architecture levels, while currently ignoring the problem of guiding the structuring of the software system into components.

9 CONCLUSION

The development of techniques supporting an intertwined elaboration of system requirements and software architectures requires a close examination of the formalisms used to support both activities and the definition of mappings that relate these formalisms at the semantic level.

In this paper, we have presented a technique for relating goal-oriented operational requirements to event-based transition systems that are typical of many architecture description languages. This allows requirements engineers to use the formal analysis capabilities of LTSA to analyse and animate KAOS operation models. These capabilities have been illustrated on a safety injection system for a nuclear power plant, revealing a potential safety-critical problem in previously published specifications. Our mapping also allows software architects to translate goal-oriented operational requirements into a black-box event-based model of the software behaviours expressed in a formalism appropriate to reason about behaviours at the architecture level.

The translation of fluent temporal logic assertion to LTS and the proposition relating logical conjunction of assertions to the parallel composition of the derived processes proved in Section 3 are key results for combining declarative and event-based state machine models whose applicability goes beyond the translation process presented in this paper.

In addition, the paper provides insights into crucial differences between modelling approaches used in requirements engineering and software architecture analysis, in particular concerning the use of synchronous and asynchronous temporal logics. In Section 7, we have shown that the synchronous nature of KAOS goals is an obstacle to deriving adequate event-based models directly from goals, and is responsible for generating event-based models whose state-spaces may be larger than necessary because of the inclusion of timing events.

Based on the insights gained during this work, we believe that considering specifying goals in asynchronous rather than synchronous temporal logic is a promising approach for future work aiming at combining methods for requirements engineering and software architectural design. A significant challenge will be to preserve the main benefits of the synchronous framework, in particular its ability to support an incremental elaboration process where partial operation model can be gradually derived from, and traced to declarative goals [Let02b]. In [Let05], we have also shown that important classes of requirements such as "immediate response" properties and some state invariants can be specified easily in synchronous temporal logic, but are extremely difficult to specify correctly in an asynchronous temporal logic. Patterns of asynchronous temporal logic properties for specifying such properties can and have been envisaged, but the study of such patterns requires further work.

Acknowledgement. The work reported herein was partially supported by the Leverhulme Trust and PICT 2005-11-32440.

10 REFERENCES

[Abr96] J.-R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge: Cambridge University Press, 1996.

[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.

[Bra01] Brandozzi, M., Perry, D.E.: Transforming goal oriented requirement specifications into architectural prescriptions. In Castro, Kramer, eds.: STRAW 2001 - From Software Requirements to Architectures. (2001) 54–60

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

[Ber03] D. M. Berry, R. Kazman, and R. Wieringa, editors, *The Second International Software Requirements to Architectures Workshop* (STRAW'03). At ICSE'03.

[Bor95] Borgida, A., Mylopoulos, J. and Reiter R., "On the frame problem in procedure specifications", IEEE Transactions on Software Engineering, October 1995.

[Bre04] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. 2004. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems* 8, 3 (May. 2004), 203-236.

[Cas02] Jaelson Castro, Manuel Kolp and John Mylopoulos, Towards requirements-driven information systems engineering: the Tropos project, Information Systems, Volume 27, Issue 6, September 2002, Pages 365-389.

[Cas01] J. Castro and J. Kramer, editors, *The First International Workshop on From Software Requirements to Architectures (STRAW'01).* At ICSE'01.

[Che99] S-C. Cheung, Jeff Kramer: Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transaction on Software Engineering and Methodology* 8(1): 49-78 (1999)

[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, Int. Conf. on Computer-Aided Verification (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.

[Dam06] C. Damas, B. Lambeau, A. van Lamsweerde: Scenarios, goals, and state machines: a win-win partnership for model synthesis. *SIGSOFT FSE 2006*: 197-207

[Dar96] 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.

[Del04] R. De Landtsheer, E. Letier and A. van Lamsweerde, Deriving Tabular Event-Based Specifications from Goal-Oriented Requirements Models, *Requirements Engineering Journal*, Vol. 9, n° 2, Springer-Verlag, May 2004, pp. 104-120.

[Fux04] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Roveri, Paolo Traverso: Specifying and analyzing early requirements in Tropos. Requir. Eng. 9(2): 132-150 (2004)

[Gia99a] D. Giannakopoulou, Jeff Magee, Jeff Kramer: Checking Progress with Action Priority: Is it Fair? ESEC / SIGSOFT FSE 1999: 511-527

[Gia99b] D. Giannakopoulou, Jeff Kramer, Shing-Chi Cheung: Behaviour Analysis of Distributed Systems Using the Tracta Approach. Automated Software Engineering 6(1): 7-35 (1999)

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

[Gia04] D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh: Assume-Guarantee Verification of Source Code with Design-Level Assumptions, *ICSE 2004*: 211-220

[Gia05] D. Giannakopoulou, C. S. Pasareanu, H. Barringer: Component Verification with Automatically Generated Assumptions. *Automated Software Engineering* 12(3): 297-320 (2005)

[Gro01] Daniel Gross, Eric S. K. Yu: From Non-Functional Requirements to Design through Patterns. Requirements Engineering Journal 6(1): 18-36 (2001)

[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 - 10th Annual Conference on Computer-Aided Verificatio*n, Vancouver, 1998, 526-531.

[Hen98] T. A. Henzinger. It's about time: Real-time logics reviewed. *Proc. 9th International Conference on Concurrency Theory (CONCUR)*, LNCS 1466, Springer, 1998, pp. 439-454.

[Kni02] J.C. Knight, "Safety-Critical Systems: Challenges and Directions", Invited Mini-Tutorial, Proc. ICSE'2002: 24th International Conference on Software Engineering, ACM Press, 2002, 547-550.

[Jac00] D. Jackson, Automating first-order relational logic, ACM SIGSOFT in Proc. Conf. Foundations of Software Engineering, November 2000.

[Jan05] Divya Jani, Damien Vanderveken, Dewayne E. Perry: Deriving Architecture Specifications from KAOS Specifications: A Research Case Study, in *Proc. EWSA 2005, 2nd European Workshop on Software Architecture*, LNCS 3527, Springer 2005, pp. 185-202.

[Jon86] C. B. Jones, *Systematic Software Development Using VDM*. Prentice-Hall International series in computer science. Englewood Cliffs, N.J.: Prentice Hall International, 1986.

[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.

[Lam98a] 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.

[Lam98b] A. van Lamsweerde, L. Willemet: Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering* 24(12): 1089-1114 (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.

[Lam02] 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, 2002, pp. 325-340.

[Lam03] van Lamsweerde, A.: From system goals to software architecture. In Bernardo, M., Inverardi, P., eds.: Formal Methods for Software Architectures. Volume 2804 of Lecture Notes in Computer Science. Springer-Verlag (2003) 25–43.

[Lam04] A. van Lamsweerde, Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice, *Proceedings of RE'04, 12th IEEE Joint International Requirements Engineering Conference*, Kyoto, Sept. 2004, 4-8 (Invited Keynote Paper)

[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.

[Let04] E. Letier and A. van Lamsweerde, Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering, *Proceedings FSE 2004 - 12th International Symposium on the Foundation of Software Engineering*, ACM Press, Newport Beach, CA, November 2004, pp. 53-62.

[Let05a] E. Letier, J. Kramer, J. Magee and S. Uchitel, Fluent Temporal Logic for Discrete-Time Event-Based Models, *Proc. ESEC/FSE 2005*, Lisbon, September 2005.

[Lev95] N. Leveson, *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[Liu01] Lin Liu. Eric Yu. From Requirements to Architectural Design: Using Goals and Scenarios. In the First International Workshop From Software Requirements to Architectures (STRAW 01) at ICSE 2001. Toronto, Canada, 14 May 2001

[Mag95] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, Specifying Distributed Software Architectures, *5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, 989, pp. 137-153, September 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)*. June 2000, Limerick, Ireland.

[Man92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems:Specification* Springer-Verlag, 1992.

[Myl06] John Mylopoulos: Goal-Oriented Requirements Engineering, Part II, *Proceedings of RE'06, 14th IEEE Joint International Requirements Engineering Conference*, Minneapolis, Sept. 2006, p 4 (Invited Keynote Paper).

[Ng96] Keng Ng, Jeff Kramer, Jeff Magee: A CASE Tool for Software Architecture Design. *Automated Software Engineering* 3(3/4): 261-284 (1996)

[Nus01] B. Nuseibeh, Weaving Together Requirements and Architectures. *IEEE Computer* 34(2), pp. 115-117, 2001

[Obj07] http://www.objectiver.com.

[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.

[Ros98] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall series in computer science. London: Prentice Hall, 1998.

[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.

[Woo96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall international series in computer science. London: Prentice Hall, 1999