Requirements Modelling by Synthesis of Deontic Input-Output Automata

Emmanuel Letier and William Heaven
Department of Computer Science
University College London
London, United Kingdom
{e.letier, w.heaven}@cs.ucl.ac.uk

***Abstract*— Modelling requirements is an effective mean to undesrtand a system's required behaviour and explore alternative system designs. The resulting models are also valuable for subsequent activities of software design, implementation, and verification. However, elaborating these models requires significant expertise and effort. The paper aims at reducing this effort by automating an essential activity of requirements modelling which consists in deriving a machine specification satisfying a set of goals in a domain. It introduces deontic input-output automata —an extension of input-output automata with permissions and obligations— and an automated synthesis technique over this formalism to support such derivation. This technique helps modellers identifying early when a goal is not be realizable in a domain and can guide the exploration of alternative models to make goals realizable.**

I. INTRODUCTION

Our objective is to support the incremental elaboration of requirements models where a set *Goals* of required properties is to be satisfied by a machine specification *Spec* in the context of domain properties and assumptions *Dom* [1][2]:

Spec, *Dom* |= *Goals*.

The machine specification must be *realizable* [3][4]. Intuitively, this means it must correspond to a transition relation between the machine's monitored and controlled events [5]. The domain model *Dom* is typically composed of multiple agents (components) corresponding to human agents, hardware devices, and other automated systems. Systematic techniques have been developed to support the elaboration of goals, domain models, and realizable machine specifications satisfying the above condition [2][4][6][7][8]. However, elaborating formal models using these techniques requires significant effort and expertise.

Automatically deriving a realizable specification that satisfies a set of goals for a given domain model is an instance of a controller synthesis problem [9][10]. Such problems have been studied extensively in a wide variety of formal frameworks [11][12][13][14][15][16][17][18]. However, most of these frameworks rely on assumptions that are not valid in a requirements engineering context, such as that of a machine that can observe all events referenced in the goals or of a machine that can always act faster than its environment (even if it is composed of other automated systems). For example, recent techniques for deriving live behaviour models from goals and domain properties assume the machine can observe all domain events [19][20]. Our objective in this paper is to present a formal framework for automated specification synthesis that is suitable in a requirements engineering context where a clear distinction must be made between goals, machine specifications and domain properties, where machine specifications can be derived compositionally from goals and domain properties, and where assumptions about the relative speed of the machine are recorded explicitly as domain assumptions instead of being built into the formalism.

A first contribution is to present a novel event-based formal foundation for requirements modelling. This foundation is motivated by previous work showing that mismatches between the synchronous, state-based foundations for goal modelling [4][7] and the asynchronous, event-based foundations for scenarios and behaviour modelling [21] prevent an efficient derivation of event-based behaviour models from declarative goals [22]. In this paper, we introduce *deontic input-output LTS* to resolve that problem. A deontic input-output LTS is an extension of input-output labelled transition systems [23] that allows one to distinguish between the permission and obligation to perform an event – a distinction that is essential to requirements modelling [7] but absent in standard event-based behaviour models. We adapt previous definitions of agent responsibility [6] and goal realizability [4] to this context.

The second contribution is an automated specification synthesis technique defined over that formal foundation. The technique derives the weakest (i.e. least constraining) realizable machine specification that satisfies a goal for a particular domain model or signals that the goal is not realizable by the machine in the domain. The technique is compositional: partial specifications can be derived from different goals and domain specifications before being combined to form a specification that satisfies the combined set of goals. It can also be used symmetrically to derive weakest domain assumptions to satisfy goals with a given machine specification. Our approach is limited to discrete-time, finite-state models and to goals that correspond to safety properties (including bounded liveness properties [16]). Our algorithm is a version of a controller synthesis algorithm under *partial observability*, which means that the controller is not assumed to be able to observe all events in its environment [12][13][14] [17][18]. It differs from previous work in this area by its clear separation between machine specification and domain assumptions, its distinction between permission and obligation, and the possibility of relying on alternative synchrony assumptions (saying that the machine is infinitely fast compared to its environment) or on no such assumption at all. We have implemented our technique in

the LTSA toolset [21]. This allows our technique to take as input goals and domain models specified as Fluent Linear Temporal Logic (FLTL) assertions [24][25], Finite State Processes (FSP) [21], the subset of the KAOS modelling language that has a LTS semantics [22], and a combination of all the above within the same model. The tool and all models mentioned in this paper are available online[1].

A third contribution is a discussion of the role of specification synthesis during requirements modelling. In practice, goals and domain models are rarely specified completely and without error before deriving the machine specification. Elaborating goals, domain models, and machine specification is an intertwined process during which alternative models corresponding to alternative choices of agents and interfaces are explored. Most works on controller synthesis are presented without explicit attention to such a process. We illustrate through an example how realizability checking and specification synthesis can be used to incrementally elaborate and explore alternative requirements models.

II. EXAMPLE

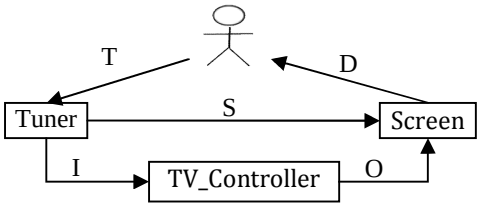
Modern television systems are built by assembling a large number of hardware and software components [26]. One of the simplest systems is a television consisting of a *Tuner*, a *Screen*, and a single *TV-Controller* software [24]. Fig. 1 shows a context diagram for this small system. In this diagram, a box denotes a system component, and a labelled arrow between two components denotes a set of events that are controlled by the source component and monitored by the target component. The *Tuner* is a component that accepts channel selections from a user and is responsible for transmitting the signal from the selected channel to the *Screen*. It also interacts with the *TV-Controller* to signal when it starts and ends tuning to a new channel. The *Screen* can be blanked and unblanked by the *TV_Controller* and when unblanked it will display whatever signal it receives from the *Tuner*. The system has two goals:

(G1) The screen should never display a static signal, a noisy signal sent by the *Tuner* while tuning between channels.

(G2) When a channel is selected, it should be displayed on the screen as soon as possible.

The *TV_Controller* must ensure these two goals by blanking the screen when the tuner starts tuning and unblanking it when the tuner has reached the selected channel. Although the system is relatively simple, we will see that elaborating a realizable specification that satisfies the goals is not trivial. For example, the specification of the *TV_Controller* in [24] requiring that the screen be blanked when the tuner is tuning, specified by the formal assertion []($Tuning \rightarrow Blanked$), is not realizable because it requires the *TV_Controller* to perform a blank event *before* a tune event occurs (this will be further explained in the following section). Specifying that the *TV_Controller* must perform a blank event *immediately after* receiving a tune event would solve the problem but raises the question of how to adequately model such a property, a problem we will solve by introducing the concept of obligation in deontic IO LTS.

The purpose of our specification synthesis technique is to support a compositional requirements modelling process where one is able to derive distinct *TV_Controller* specifications *S1* and *S2* from each goal separately. We want these specifications to be such that (a) *S1* satisfies *G1* for some domain assumptions *Dom1* about the screen and tuner, (b) *S2* satisfies *G2* given some other domain assumptions *Dom2* (consistent with *Dom1*), and (c) the composition of *S1* and *S2* ensures the satisfaction of both goals in the domain described by the combination of *Dom1* and *Dom2*. Such compositionality is essential to support a goal-oriented requirements modelling process where explicit links are maintained about which individual statements of domain assumption and machine specifications are needed to satisfy which goal [2].



T = {select[Channel]}
D = {display.{[Channel], static, blank}}
S = {signal.{[Channel], static}}
I = {tune, end_tune}
O = {blank, unblank}

Figure 1. Context Diagram for a Simple Television System

III. BACKGROUND

We use labelled transition systems (LTS) as a *semantic domain* for requirements modelling and analysis. The formalism is well-studied, has effective tool support [21], and is used to express the semantics of a variety of modelling languages including the FSP process algebra [21], fluent temporal logic [24][23], scenario-based models [27][28][29], and an expressive subset of the KAOS language [22].

A. Labelled Transition Systems

Let *Act* be the universal set of observable events and τ denote a special unobservable event. A *labelled transition system M* is a tuple $<Q, \alpha M, \delta, q0>$ where Q is a finite set of states; $\alpha M \subseteq Act$ is a set of observable events called the alphabet of M; $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation; and $q0 \in Q$ is the initial state. An event ev is said to be enabled in a state q if there exists q' such that $(q, ev, q') \in \delta$. A LTS M is non-deterministic if it contains τ-transitions or if there exists (q, ev, q') and (q, ev, q'') in δ with $q' \neq q''$. Otherwise, M is deterministic. In this paper, we are concerned with LTS trace semantics. The traces of an LTS M, noted $Tr(M)$, are the sequences of observable events starting from the initial state and obeying the transition relation. The parallel composition of two LTS P and Q, noted $(P \parallel Q)$, denotes their joint behaviour which is the result of both LTSs executing asynchronously but synchronizing on all shared events, i.e. events (other than τ) in $\alpha P \cap \alpha Q$.

Figure 2 shows an example LTS describing the behaviour of a *TV_Controller*. By convention, the initial state is always labelled 0. The numbers attached to the other states have no meaning; they are identifiers generated during the mapping from modelling languages (e.g. FSP) to LTS. Other LTS, not shown in the paper, describe the behaviour of the *Screen* and *Tuner*. The behaviour of the whole system is given by the composition (*Tuner* $\parallel$ *TV_Controller* $\parallel$ *Screen*). In our model,

[1] http://www.cs.ucl.ac.uk/staff/e.letier/ltsa-ctrl-2012.zip

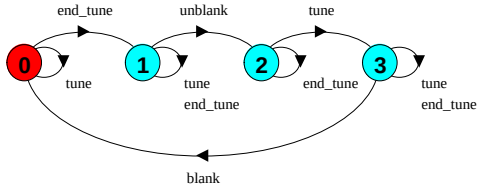


Figure 2. IO LTS for the *TV_Controller*

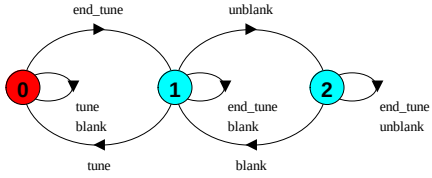


Figure 3. Interface LTS for the *TV_Controller*

the system starts in a state where the screen is blanked and the tuner is tuning to channel 1. In Fig. 2, when the *TV_Controller* receives *end_tune* in state 0, it moves to state 1 where it can perform *unblank* leading to state 2. In state 2, when it receives *tune*, it can perform *blank* leading back to state 0.

LTS models merely specify when events *may* occur; they do not specify when they *must* occur. For example, when the *TV_Controller* is in state 3, *blank* may occur (it is enabled) but nothing in the LTS itself specifies that it must occur. In this example, this is a problem because while the *TV_Controller* is in state 3, if it doesn't perform *blank* immediately, the *Tuner* could send a *signal.static* event which could be followed by the *Screen* performing *display.static* which would violate *G1*.

To force events to occur, behaviour models usually rely on additional fairness and "maximal progress" assumptions. For example, the fair choice assumption in [21] states that every transition that is enabled infinitely often must be executed infinitely often. Maximal progress assumptions consist in giving higher priority to machine-controlled events (in our example to the *TV_Controller* events *blank* and *unblank*) so that when enabled, these events are always performed before all other events. Maximal progress means that the machine always performs its enabled events faster than its environment. In our example, giving higher priority to *blank* and *unblank* events in (*Tuner* || *TV_Controller* || *Screen*) ensures that when the *TV_Controller* is in state 3 of Fig. 2, it will perform *blank* before the *Tuner* and *Screen* send and display a static signal. However, fairness and maximal progress are problematic for requirements modelling. They both are global system properties that prevent compositional reasoning needed for goal-oriented requirements modelling. Their validity is also questionable. In requirements models, fairness assumptions can't be justified as being abstractions of a global process scheduler as they are for models of concurrent programs. The validity of maximal progress assumptions is also hard to justify when the domain is composed of other automated components that may be as fast if not faster than the machine. In Section IV, we'll present a different way to specify progress using obligations without relying on fairness and maximal progress assumptions. In Section VI, we'll also show how alternative "synchrony assumptions" about the relative speed of the machine with respect to its environment can be included explicitly in the domain model.

B. Interface and Input-Output LTS

An *interface labelled transition system* (I-LTS) is an LTS M = <Q, αM, δ, q0> where the alphabet αM is partitioned into three sets *in*, *int*, and *out* denoting the input, internal, and output events of the LTS, respectively [30]. An *input-output labelled transition system* (IO-LTS) is an interface LTS whose transition relation δ is required to satisfy the **input-enabling rule** according to which every input event must be enabled in every state, i.e. for all $s \in Q$, $ev \in in$, there exists $s' \in Q$ such that $(s, ev, s') \in \delta$ [23].

The input-enabling rule is fundamental to requirements modelling: it corresponds to the real-world property that an agent cannot prevent occurrences of events controlled by its environment [1][4][23]. Fig. 2 satisfies this rule for the *TV_Controller*. Fig. 3 is an alternative LTS model, used as a specification for the *TV_Controller* in [24], that does not satisfy this rule because the input event *tune* is not enabled in state 2. As we will see below, this LTS defines the behaviour admitted by the assertion []($Tuning \rightarrow Blanked$) requiring the screen to be blanked while the tuner is tuning. It is impossible to implement a *TV_Controller* satisfying this model because in state 2, a state where the *Tuner* is tuned and *Screen* unblanked, it requires the *TV_Controller* to perform *blank* before *tune* occurs.

Instead of requiring every component's LTS to satisfy the input-enabling rule, the formalism of interface automata constrains the environment models that can be composed with an interface automaton: the environment model must be such that it never produces an input event when that event is not enabled in the interface LTS, and it should never block the occurrences of output events performed by the interface LTS. This approach is problematic for requirements modelling because it means that an interface LTS mixes the component's required behaviour with assumptions on its environment. For example, Fig. 3 models both constraints on the *TV_Controller* behaviour (not to perform *unblank* in state 0) and assumptions on the *Tuner* behaviour (not to perform *tune* in state 2).

In the paper, we will use the LTS 'interface' operator $M@\alpha$ to denote the projection of an LTS *M* on an alphabet $\alpha \subseteq Act$. This projection is obtained from *M* by replacing in the transition relation all events not in α by τ. At the trace level, the projection of a trace σ on $\alpha \subseteq Act$, noted $\sigma\uparrow\alpha$, is the trace obtained by removing from σ all occurrences of events not in α. We will also use the common approach of modelling passage of time by an explicit *tick* event to which all LTS synchronize [21].

C. Modelling System Properties

System properties are modelled as property LTS. A *property LTS* is a deterministic LTS *P* whose set of traces defines a set of admissible behaviour over the property's alphabet. A LTS *M* satisfies *P*, noted $M \models P$, if and only if $Tr(M)\uparrow\alpha P \subseteq Tr(P)$. A property *P* is weaker (less constraining) than or equivalent to a property *Q* if and only if for all M, if $M \models Q$ then $M \models P$. Every property LTS *P* has an associated *error LTS* P_{err} that characterizes all possible violations of *P*. Such LTS contains a special error state, labelled -1, denoting a state to be avoided. This error LTS is obtained from *P* by adding transitions to the error state for each missing transition in *P*. To verify whether $M \models P$, LTSA computes ($M \parallel P_{err}$) and

checks whether the error state is reachable in the resulting model. Any trace to the error state is a counter-example to $M \models P$. If the error state is not reachable, then $M \models P$ [31].

For example, Fig. 4 shows the error LTS for G1. To verify this goal, LTSA checks whether the error state is reachable in (*Screen* || *Tuner* || *TV_Controller* || $G1_{err}$).

At the syntactic level, properties can be described in FSP [21] or more declaratively in FLTL [24]. FLTL assertions are linear temporal logic assertions that can refer to events and to state propositions called fluents. A fluent is defined by a pair of set of events that make the fluent become true and false, respectively and by its initial condition. For example, in our TV system, the fluents *Tuning* and *Blanked* are defined as follows:

fluent *Tuning* = <*tune*, *end_tune*> **initially** *True*
fluent *Blanked* = <*blank*, *unblank*> **initially** *True*

FLTL assertions are build from fluents using the usual Boolean operators, linear temporal logic operators such as [] (always), <> (evenutally), *W* (waiting for) as well as bounded temporal operators such as $<>_{<d}$ (eventually within *d* time units) which can be used to model bounded liveness properties [25].

If an FLTL assertion *P* is a safety property, LTSA can generate an *error LTS* P_{err} characterizing all possible violations of *P*. For example, *G1* can be defined by the FLTL assertions []! *display.static* and *G2* by *forall* [c:*Channel*] [] (*request*[*c*] -> <> {<*d*} *display*[c]) where *d* is a constant (e.g. set to 2 time units). Both are safety properties. The error LTS corresponding to *G1* is shown in Fig. 4. *G2* generates another error LTS (of 8 states for 2 channels and d=2) characterizing sequences of events that violate this goal.

In any LTS *P*, transitions to error can be removed using the function *behaviour*(*P*). This can be used to transform an error LTS generated from an FLTL assertion into a property LTS. For example, as mentioned previously, the behaviour LTS for the FTL assertion [](*Tuning* → *Blanked*) is shown in Fig. 3.

IV. FOUNDATION

We now introduce our novel event-based foundation for requirements modelling. This foundation adapts the synchronous state-based model of KAOS agents [4] to the asynchronous event-based model of LTS. In our model, an agent is characterized by the following items:

- an *interface* declaring disjoint sets of events that are monitored (input) and controlled (internal and output) by the agent;
- a *behaviour model* given as an LTS that satisfies the conditions of a deontic IO LTS given below;
- a *responsibility relation* that maps the agent to the set of goals the agent is responsible for.

Multiple agents run concurrently and synchronize on shared events. A single event can be monitored by several agents, but we require each event to be controlled by at most one agent. A system context diagram, such as the one shown in Fig. 1, provides a graphical view of the agents' interfaces.

The key differences with the previous KAOS model of agents are that the agent interfaces are composed of events rather than state variables, the agents' behaviour models are defined as asynchronous deontic IO LTS instead of synchronous state-based transition systems, and the goals and agents' responsibilities are defined as asynchronous properties instead of synchronous ones (see [22] for an explanation of the difference between synchronous and asynchronous properties).

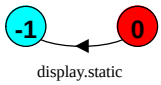


Figure 4. Error LTS for *G1*

A. Deontic IO LTS

Deontic IO LTS are extensions of IO LTS that enable the distinction between states in which controlled events are permitted to occur and states in which controlled events are obliged to occur. This distinction relies on the concepts of stable states and transient states: a *stable state* is a state in which time is allowed to pass, a *transient state* is one in which time is not allowed to progress. The formalism has no *explicit* constructs for obligation and permission. The notions of permission and obligation rather come from the (informal) interpretation we give to transitions in the LTS. The *permissions* of an agent in a given state are represented as usual by the controlled events that are enabled in that state. The *obligations* of an agent are represented by the sequences of controlled events leading to a stable state. If the agent is already in a stable state, it has no obligation. If it is in a transient state, then it has an obligation to perform one or more controlled events to reach a stable state.

Instead of extending LTS with new language constructs for stable and transient states, we have chosen to rely solely on *tick* events to mark this distinction. If *tick* is not in the LTS alphabet, then all its states are stable. If *tick* is in the LTS alphabet, then stables states are those where *tick* is enabled and transient states are those where *tick* is not enabled. This approach has the benefit of being consistent with the standard interpretation of LTS models with respect to time and permissions. By convention, in LTS figures, transient states are marked with a double ring.

For example, Fig. 5 shows a deontic IO LTS for the *TV_Controller* specifying that this agent *may* perform *unblank* in state 1 and it *must* perform *blank* in state 3. Fig. 2 is another deontic IO LTS for the *TV_Controller*. Here, because the model is untimed, all states are stable and the model includes no obligation. This model has the usual LTS meaning of specifying that *unblank* may occur only in state 1 and *blank* may occur only in state 3.

For a deontic IO LTS to be well-formed, an agent should not have obligations that it can't fulfil because it doesn't have the necessary permissions to do so. This is captured by the *independent progress rule*. It states that for every transient state, the transition relation must include a sequence of controlled events that lead to a stable state. In other words, the agent model must be such that, *whatever the behaviour of other agents*, the agent is always able to independently reach a stable state where time can progress.

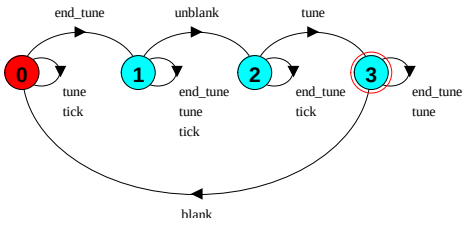


Figure 5. *unblank* may occur in state 1; *blank* must occur in state 3. A state with a double ring denotes a transient state.

Figure 5 satisfies independent progress because in state 3, the *TV_controller* can perform *blank* to reach the stable state 0. Figure 6 does not satisfy it because in states 1 and 2, the *TV_Controller* has no sequence of controlled events leading to a stable state.

Formally, deontic IO LTS are defined as follows:

Definition (deontic IO LTS). A deontic IO LTS is an IO LTS $M = <Q, \alpha M, \delta, q0>$ that must satisfy the ***input-enabling rule*** and an additional ***independent progress rule*** requiring that for every transient state $s0 \in Q$, there is a finite sequence of controlled events (i.e. output and internal events) $<c_1, ..., c_n>$ that leads to a stable state, i.e. for all $s_1, ..., s_n$ such that $<s_{i-1}, c_i, s_n> \in \delta$, s_n is a stable state.

The trace semantics of deontic IO LTS is the standard trace semantics of LTS. Note that stable and transient states should not be confused with accepting and non-accepting states in Buchi automata. Unlike accepting states in Buchi, stable states are not required to be visited infinitely often.

Because deontic IO LTS are LTS, they can be composed, determinized, minimized and projected on alphabets like any other LTS. The behaviour of a system composed of multiple agents is defined by the standard LTS parallel composition of the agents' deontic IO LTSs. Because in our framework every event is controlled by at most one agent, the input-enabling and independent progress rules ensure there are no interferences between the transition relations of different agents (no agent model can affect the permissions and obligations in another agent's model).

The behaviour model of a single agent can also be defined as the composition of several deontic IO LTSs for the agent. This composition is guaranteed to preserve the satisfaction of the input-enabling rule, but *not* of the independent progress rule. This will for example happen when one deontic IO LTS imposes an obligation to perform an event in a certain state and the other deontic IO LTS forbids this event to occur in that state. In practice, this means that when we define the behaviour of an agent as the composition of several deontic IO LTSs, as we will do in Section VI and VII, it is necessary to check that the composed model satisfies the independent progress rule. Violations of that rule signal an inconsistency between the agent's obligations and permissions.

Note that deontic IO LTS allow one to model agents that have multiple simultaneous obligations without introducing an inconsistency. This is a characteristic that caused difficulties in previous deontic formalisms where obligations was defined as the obligation for an event to occur next in the execution trace [32]. When traces are defined as sequences of events (i.e. they have an interleaving semantics), such semantics of obligation introduce inconsistencies as soon as an agent has more than one simultaneous obligation because only one of the obliged events can possibly occur next in the trace. Our approach avoids this problem by defining obligations as the obligation to perform events before the next occurrence of *tick*.

An important difference between deontic IO LTS and other models of concurrency is that it doesn't rely on fairness to ensure that some of the enabled events are eventually performed. This limits the expressive power of deontic IO LTS to safety properties (it's impossible to model a liveness property with a deontic IO LTS). This is a deliberate choice justified by the pointlessness in requirements models of assigning responsibility for such a property to an agent. A liveness property —classically defined as a property for which every finite trace can be extended into a trace satisfying it—is by definition not testable (no violation can be witnessed in the running system) and it imposes no real constraint on the agent behaviour (its satisfaction can always be postponed)[4]. For example, a specification stating that all debtors must repay their debts *eventually* –without time bound on the repayment– allow debtors to indefinitely postpone repayment without ever violating the specification. Liveness properties are therefore inadequate as statements of requirements. This does not mean that we exclude liveness entirely from requirements models. They are useful abstractions of bounded liveness properties when it's more convenient to leave the actual bound unsepecified [16], but as long as they remain pure liveness properties they are untestable and impose no real constraints on the running system. In our formal framework, goals and domain properties defined in FLTL can be livness properties, but the controller synthesis technique presented in this paper can't deal with them.

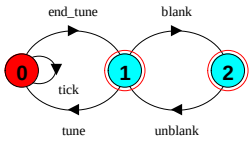


Figure 6. Violation of independent progress

B. *From LTS to deontic IO LTS*

When an agent LTS model does not satisfy the input-enabling and independent progress rules, it is possible to automatically transform it into a deontic IO LTS by adding additional transitions to the models. This is useful for transforming standard LTS defining the semantics of FSP, FLTL, and KAOS operation models into deontic IO LTS. It shows that it is possible to define the semantics of existing requirements modelling languages in deontic IO LTS.

Alternative extension of an LTS into a deontic IO LTS are possible. For our synthesis procedure in Section V, we will use the *weakest IO completion* that given an LTS P with alphabet αP and a set of controlled event $Ctrl \subseteq \alpha P$, generates the weakest deontic IO LTS (i.e. the one with the most traces) that satisfies P. This LTS is generated by adding to P a new "*sink*" state (i.e. a state with no outgoing transitions) with self-transitions for all events in αP and new transitions (s, e, *sink*) from every state s where some input event $e \subseteq \alpha P \backslash Ctrl$ is not enabled, and new transitions (s, *tick*, *sink*) from every transient state s for which there is no sequence of controlled events to a stable state in P. For example, Fig. 8 shows the weakest IO completion for the LTS in Fig 7. The new sink state is state 3. This model an agent that behaves as in P, but enters into a sink state where all behaviours are permitted as soon as some unexpected input or tick event occurs. Alternative functions to transform an LTS into a deontic IO LTS are the *strongest IO completion* (where the sink state admits input and tick events only) and the *robust IO completion (*where instead of adding transitions to a sink state, missing input and tick events generate new self-transitions, i.e. these event are just ignored by the agent). The choice of which completion function to use is left to the language designers or to modellers. Considerations about modelling language design are left out of this paper.

C. Responsibility and Realizability

In our model, goals define sets of admissible traces. The *responsibility* of an agent is a set of goals that the agent must satisfy regardless of the behaviour of other agents in its environment [2][4][6]. This is captured by the following constraint.

Definition (Agent responsibility constraint). If an agent is assigned responsibility for a goal *G*, then its behaviour model *M* must satisfy *G* for all valid environment *E* for this agent, i.e. $M \| E \models G$.

In this definition, a valid environment is one whose behaviour model is a deontic IO LTS for the environment: the environment's input, i.e. the agent's output events, must be enabled in every state, and the model must ensure independent progress through environment-controlled events. When these conditions are satisfied, we can show that the responsibility constraint is equivalent to the simpler constraint $M \models G$ requiring that *M* satisfies *G* without any environment model (which means that all environment events can occur at all time). This property is important to allow us to check agents' responsibilities independently of the environment models.

The notion of realizability has its usual meaning, adapted to agents characterized by deontic IO LTS behaviour models.

Definition (Realizability). A goal G is *realizable by an agent* if there exists a deontic IO LTS *M* for that agent such that $M \models G$. A goal *G* is *realizable by an agent* for a domain model *DOM* if there exists a deontic IO LTS *M* for that agent such that $M \| DOM \models G$.

As usual, two goals may be realizable individually, but not together because the behaviour models that realize each goal individually may be inconsistent with one another (their composition admits no trace or it does not satisfy the independent progress property).

V. Specification Synthesis

We now describe a technique for checking whether a goal *G* is realizable by an agent *Ag* in a given domain model *DOM*, and if the goal is realizable to synthesise the weakest realizable specification for *Ag* that satisfies *G* in *DOM*. Our technique takes as input a goal *G* given as a property LTS G_{err} (derived from an FLTL assertion or FSP property), a domain model *DOM* in the form of a deontic IO LTS (again, derived from FSP, FLTL, or KAOS descriptions), and the agent's interface alphabet αAg partitioned into its monitored (its input) and controlled events (its output and internal events). The agent's controlled events are noted *Ctrl*(*Ag*).

A. Specification Synthesis Algorithm

The algorithm proceeds in two steps. The first step relies on an existing technique [33] to generate the weakest *interface* LTS that satisfies *G* in *DOM* (our presentation and context of use are however different from [33]). The second step adds the constraint that the agent LTS must satisfy the input-enabling and independent progress rules of deontic IO LTS.

Step 1. Deriving the interface controller. The derivation of the weakest interface LTS that satisfies *G* in *DOM* has three substeps.

The first substep computes $C_O = (DOM \| G_{err})$ which gives an LTS whose error traces are all traces in *DOM* that violate *G*.

Example: To check the realizability of G1 by the *TV_Controller*, we computer $C_O = (Screen \| Tuner \| G1_{err})$. This generates an LTS (with 33 states) characterizing all behaviours of a system composed of a *Screen* and a *Tuner* (without *TV_Controller* yet) and where all behaviours that violate *G1* lead to the error state. □

If C_O has no error trace, the domain model satisfies the goal without the need for any constraint on the agent. In this case, the synthesis procedure terminates and the weakest inteface and deontic IO LTS for the agent consists of a single state that accepts all events in the agent's alphabet. If C_O contains error traces, the domain model in itself does not satisfy the goal, and the synthesis procedure must continue.

If the agent for which we want to derive a specification was omnipotent, i.e. if it could observe and control all events, then it would be able to avoid error traces in C_O (the subscript in C_O stands for omnipotent). But the agent is not omnipotent, and C_O may contain traces σ and σ' that are indistinguishable to the agent, i.e. such that $\sigma{\uparrow}\alpha Ag = \sigma'{\uparrow}\alpha Ag$. For example, the traces σ = <*select.2*, *tune*, *signal.2*> and σ' = <*select.2*, *tune*, *signal.static*> are indistinguishable by the *TV_Controller* because their projections on its alphabet are both equal to <*tune*>. These two traces lead to different states in C_O; the first leads to a state where the tuner has reached the selected channel and therefore it would be safe for the *TV_Controller* to unblank the screen, the second leads to a state where the tuner is still sending a static signal and therefore it wouldn't be safe to unblank the screen. However, when one of these two traces is executed, the *TV_Controller* will only observe the occurrence of *tune* and will therefore not know which of these two states (and possibly others) the system is in.

The second substep takes the agent's partial observability into account by merging together states in C_O that are indistinguishable to the agent. This is achieved by projecting C_O on the agent's alphabet and *tick* event (using the LTSA 'interface' operator @) and determinizing the resulting model:

$$C_I = deterministic(C_O @ \alpha Ag \cup \{tick\}).$$

The determinization function uses the classic powerset construction in which sets of states in the non-deterministic LTS are mapped to states in the deterministic LTS. In this construction, if a set of states of the non-deterministic LTS includes the error state, then the related state in the deterministic LTS is labelled as error. Intuitively, if a transition can non-deterministically lead to the error state or not, then it should be avoided and therefore in the deterministic LTS it should lead to error.

Example: In our running example, $C_I = deterministic(C_O @ \{tune, end\_tune, blank, unblank, tick\})$. This generates the LTS shown in Fig. 7. □

In step 2, the derivation of the weakest deontic IO LTS realizing *G* will continue from C_I. The third substep of step 1 is needed only if one wants to derive the property LTS $Spec_I$ characterizing the weakest interface LTS satisfying *G* in *DOM*. This LTS is obtained by generating the weakest IO completion of C_I then removing its transitions to error.

Example: Applying the weakest IO completion to the LTS in Fig. 7 generates the LTS in Fig. 8 (see Section IV-C). To obtain the weakest interface property LTS, we then need to

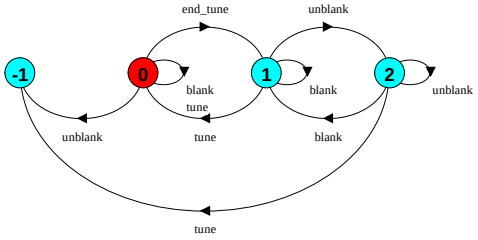


Figure7. LTS C_I derived from *G1* and *DOM*

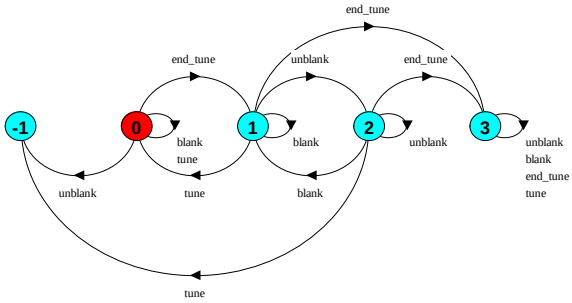


Figure 8. Error LTS for the property $Spec_I$ derived from G1

remove the transitions from state 0 and 2 to error. It is interesting to observe that the LTS in Fig. 3 corresponding to the property [](*Tuning* → *Blanked*) is a refinement of the synthesised model (all traces of Fig.3 are traces of $Spec_I$). □

The correctness of this last substep relies on the fact that transitions to the sink state added during the weakest IO completion correspond to transitions that are not enabled in *DOM* and therefore will never be exercised when $Spec_I$ is combined with *DOM*. A proof of correctness is given in [33].

Step 2. Deriving the deontic IO LTS controller. If the agent could control its input events and all occurrences of *tick*, then it would be possible to implement an agent that prevents error traces in C_I. But in our framework, an agent can't prevent occurrences of its input events and it can't prevent *tick* transitions to error unless it can perform a sequence of controlled events to a stable state.

For example, in Fig. 7, the *tune* transition to error cannot be prevented by the *TV_Controller* because it corresponds to an input event controlled by the *Tuner*. As another example, in Fig. 9, the *TV_Controller* can't prevent the *tick* transitions to error because it doesn't have a sequence of controlled events to a stable state. To avoid the error state, the agent must avoid reaching states that have such uncontrollable transitions to error. For example, the *TV_Controller* should avoid state 2 in Fig. 7 and it should avoid states 1 and 2 in Fig. 9. The second step of the synthesis procedure consists in transforming such states into an error state by *backward propagation* of the error state through such uncontrollable transitions. We have extended LTSA with a new "*Controllable*" function that performs this step:

$$C_D = Controllable(C_I)@\{Ctrl(ag)\}.$$

Our syntax overloads the @ symbol in LTSA; here it is used to declare the set of controlled events for the backward error propagation (rather than the alphabet on which to project an LTS). Our implementation assumes that the LTS model on which this function is performed is deterministic, which is always the case when performed after step 1.

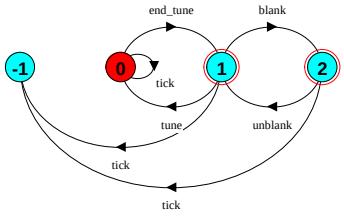


Figure 9. Uncontrollable tick transitions to error

Examples: (1) Applying this operation to Fig 7 with *Ctrl*(*Tv_Controller*) = {*blank*, *unblank*} turns state 2 into an error state resulting in the LTS in Fig. 10. This LTS specifies that to achieve *G1* the *TV_Controller* must never *unblank* the screen. This will make *G1* realizable but not in a way that can also satisfy *G2*.

(2) Applying this function to Fig. 9 first transforms states 1 and 2 into error, then it also transforms state 0 into error because its *end_tune* transition to state 1 (which is now assimilated to error) can't be prevented.

(3) Not every tick transition to error results in merging its source state with error. For example, state 3 of the *TV_Controller* model in Fig. 12 is not merged with error because an occurrence of *blank* leads the system to a stable state. In this model, when the *TV_Controller* is in state 3, it can avoid an error by blanking the screen before the next *tick*. □

If, as a result of this step, the initial state becomes the error state, there is no deontic IO LTS for the agent that can satisfy *G* in *DOM*, and *G* is therefore not realizable by the agent in *DOM*. Otherwise, the agent's weakest realizable specification $Spec_D$ can be constructed by generating the weakest IO completion of C_D and removing the remaining transitions to the error state in the same way as it was done to transform C_I into $Spec_I$ at the end of step 1.

The following proposition establishes that $Spec_D$ is the weakest deontic IO LTS for the agent that satisfies *G* in *DOM*.

Proposition. For all deontic IO LTS model *M* for *Ag*, $M \models Spec_D$ if and only if $(DOM \parallel M) \models G$.

Its formal proof is omitted due to space limitation. It follows the same structure as the proof of correctness for step 1 in [33].

B. Compositionality

Because we're dealing with safety properties only, our approach is compositional. We can show that if *S1* is the weakest specification satisfying *G1* in *DOM* and *S2* the weakest specification satisfying *G2* in *DOM*, then (*S1* || *S2*) is the weakest specification satisfying *G1* /\ *G2* in *DOM*, provided that (*S1* || *S2*) satisfies the independent progress rule. If it doesn't satisfy this rule, then *G1* /\ *G2* is not realizable by the agent in *DOM*.

This argument assumes that the same domain model *DOM* has been used to derive *S1* and *S2* from *G1* and *G2*. In practice, we will often use one set of domain assumptions *Dom1* for *G1* and another set of domain assumptions *Dom2* for *G2*. When this is the case, the model (*S1* || *S2*) will satisfy *G1* /\ *G2* in

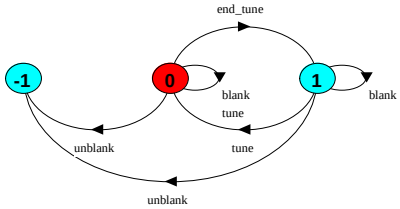


Figure 10. Controllable LTS C_D derived from Fig. 7

(*Dom1* || *Dom2*) but it is not guaranteed to be the *weakest* such specification. This has no serious consequences in practice. It only means that if (*S1* || *S2*) doesn't satisfy the independent progress rule (and is therefore not a valid specification for the agent), by rederiving *S1* and *S2* from *DOM* = (*Dom1* || *Dom2*), it might be possible to find a weakest specification that satisfies independent progress. If not, then *G1* and *G2* are not realizable together in *DOM*.

Note that we've been able to maintain compositionality even though our technique deals with agent's progress (through obligations). This is a consequence of modelling obligations as safety properties instead of relying on fairness to force events to happen eventually.

VI. A Realizability-Driven Modelling Process

We now consider how checking goal realizability and synthesising specifications from goals support the requirements modelling process. Finding that a goal is not realizable by an agent or that the weakest specification that realizes the goal is not adequate (as was the case for Fig. 10 in our running example) prompt modellers to modify the goal definition, domain assumptions, or add new agents and agent interfaces to fix the problem. In previous work [4], we have defined a library of model transformations to resolve realizability problems. In this section, we illustrate on our running example how identifying and resolving realizability problems help modellers exploring alternative requirements models relying on different synchrony assumptions about the relative speed of the machine with respect to its environment.

A. Alternative Models Satisfying G1

We have seen in the previous section that the weakest deontic IO LTS satisfying G1 is one where the *TV_Controller* never unblanks the screen. This is not a good model as it prevents G2 from being satisfied.

To understand the cause of a realizability problem, it is useful to look at the error traces in the intermediate models C_O and C_I characterizing the error trances for an omnipotent and interface automata, respectively. In this example, looking at Fig. 7 shows that the problem is caused by the occurrence of *tune* in state 2. To prevent such tune transition to error, we would like the *TV_Controller* to perform *blank* before *tune* occurs, thereby leading the system to the safe state 1. We explore four alternative model transformations to achieve this.

A first option is to add a domain assumption on the *Tuner* stating that after a *tune* event, it will not send *signal.static* before the next tick, modelled by the following FLTL assertion:

StaticTimeLag = [](*tune* -> (! *signal.static* W *tick*)).

This gives the *TV_Controller* time to blank the screen before *signal.static* occurs. The controllable error LTS C_D generated with the modified domain model is shown in Fig. 11. It shows that the *TV_Controller* cannot perform *unblank* in state 0 and it must perform *blank* when it is in state 3.

Note that even though *G0* is an untimed safety property, its satisfaction imposes an obligation on the *TV_Controller* to blank the screen after a *tune* event. Such immediate obligation can't be modelled with untimed interface or input-output automata. In our deontic IO LTS, it is represented by the *blank* transition from transient state 3.

A second set of options is to use generic *synchrony assumptions* about the relative speed of the machine with

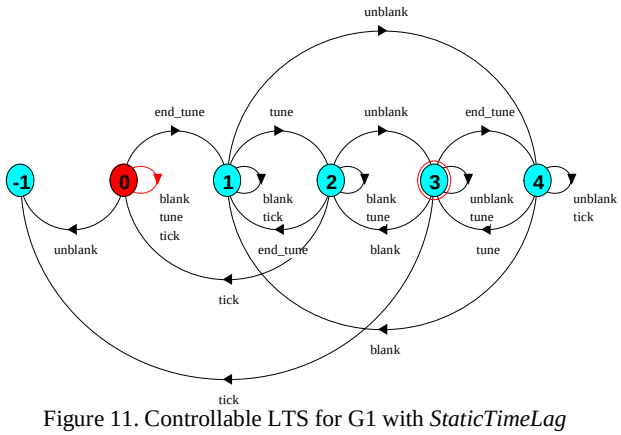


Figure 11. Controllable LTS for G1 with *StaticTimeLag*

respect to its environment. Different modelling frameworks rely on different built-in synchrony assumptions. For example, SCR requirements models rely on the *one-input and synchrony assumption* stating that the machine will receive one input at a time and will always be able to process it before the arrival of its next input [5]. In our framework, such assumption can be included explicitly in *DOM* by instantiating the generic LTS in Fig. 12. Checking *G1* realizability on the revised model reveals that the only way to satisfy the goal under this assumption is again to keep the screen always blanked. Kupferman and Vardi's controller synthesis technique under partial observability [12] relies on a stronger built-in synchrony assumption stating that the environment can perform at most one unobservable event followed by at most one input event before giving turn to the machine to perform its controlled events. In our approach, this can again be modelled explicitly in *DOM* rather than being built into the formalism. Checking *G1* realizability under this alternative assumption gives an acceptable specification in which the screen may be unblanked after an *end_tune* event and must be blanked again after a *tune* event. The generated weakest deontic IO LTS is weaker (and more complex in terms of number of states and transitions) than the one in Fig. 11 because it exploits the synchrony assumption to identify all possible states in which the *TV_Controller* could unblank the screen and still have time to blank it to prevent a goal violation.

The fourth option does not rely on any timing assumption but instead consists in adding to the context diagran of Fig. 1 a new event *tune.return* controlled by the *TV_Controller* and monitored by the *Tuner*, together with a domain assumption stating that after *tune*, the *Tuner* will not start tuning before it receives *tune.return* from the *TV_Controller*:

No_Static_Before_Tune_Return
= [](*tune* -> (!*signal.static* W *tune.return*)).

The synthesized controllable LTS is equivalent to the one in Fig. 11 except that all occurrences of *tick* are now replaced by *tune.return*. It states that when the screen is blanked, the *TV_Controller* may not perform *unblank* before receiving *end_tune*, and when the screen is unblanked, it may not perform *tune.return* before *blank*.

B. Synthesis for the bounded Achieve goal G2

Synthesis for goal G2 also revealed interesting realizability problems that led to transforming the initial model. To satisfy *G2*, the domain model was first extended with assumptions

specifying obligations on the *Tuner* and *Screen*, such as the assumption that when a user selects a new channel, the tuner will immediately send *tune* which will be followed by *end_tune* within at most one tick, and the assumption that when the screen is unblanked, it always displays the signals it receives immediately (before the next *tick*). Checking *G2* realizability with these assumptions reveals the goal to be unrealizable. The cause of the problem was that after selecting a channel, the user could select another channel before the first one had been displayed making the goal impossible to satisfy in the domain. The resolution in this case was to weaken the goal by requiring that the selected channel should be displayed within d time units (d=2), except if some other channel has been selected in the meantime.

VII. Discussion

We have applied our technique to a number systems including the classic turnstile [34] and mine pump [35] examples as well as a more complex, real-world proton therapy control system [36]. We make the following main observations from our experience:

- In all cases, the specification synthesis technique provided valuable early feedback about goal realizability and allowed us to synthesise detailed machine specifications in the form of deontic IO LTS that would be very hard to construct manually. The use of deontic IO LTS instead of untimed interface or IO LTS was for most problems essential to derive adequate machine specifications as these involved obligations that can't be modelled in other formalisms.
- Checking the domain model for satisfaction of the input-enabled and independent progress rules was also an efficient way to detect critical errors early in our models. There is a high risk that without these checks, one would synthesise a machine specification that satisfies the goal but for an invalid domain model. In practice, this is one of the main causes of requirements errors [2][8].
- Understanding the cause of unrealizability is essential to identify how to fix the problem. However, identifying theses causes by examining the behaviours in our intermediate models C_O and C_I is extremely difficult and would benefit from automated support.
- Even on these small examples, we sometimes struggled to find how to modify a domain model to make the goals realizable. We sometimes believed we knew upfront what the machine specification should be and that it would be easier to write this specification first and apply the synthesis technique to derive the weakest domain assumption needed by this specification to satisfy the goal. This rarely worked. The synthesised domain assumptions were generally either invalid or too complex to check their validity.

From this, we believe the two most important problems to improve the practicality of specification synthesis for requirements modelling are (1) to provide explanations about why a goal is unrealizable and how to fix it, and (2) to facilitate the modeller's understanding of the synthesised models, for example by inferring declarative required pre- and trigger conditions from deontic IO LTS.

The performance of our technique on all our examples was always almost instantaneous (a couple of seconds at most). The most computationally expensive steps in our approach are the generation of LTS from FLTL assertions and the determinization step. In practice, this becomes an issue only if an FLTL assertion involves a large number of fluents (6 or more) or the deterministic LTS becomes too large. Most goal-oriented requirements models will avoid these cases because they describe partial views related to specific goals. If needed, performance might be improved with symbolic techniques [15].

VIII. Related Work

Our formal foundation is based the KAOS semantics of agents that defined realizability in a requirements context but without automating realizability checking [4]. It also solves previous limitations for relating declarative requirements and even-based behaviour models [25]. Unlike interactive techniques for synthesising behaviour models from goals and scenarios [37][38], our approach does not rely on modellers to guide the synthesis. Furthermore, these works do not consider the question of realizability of the synthesised models.

Modal Transition System (MTS) is an alternative formalism that extends LTS with a distinction between *may* and *must* transitions [39]. May and must transitions in MTS are *design-time* concepts used to model uncertainties about which events should be enabled in which state. In MTS, a may transition is one that *may be enabled*, a must transition is one that *must be enabled*. In contrast, permissions and obligations in deontic IO LTS are *run-time* concepts. An agent's obligation denotes transitions that *must be performed* before the next tick. Such obligations can't be modelled in MTS. Both formalisms are intended to support incremental model elaboration. Whether and how the two approaches should be combined is a topic for further study.

Our approach addresses a problem of controller synthesis under partial observability for discrete-time models. Previous solutions to this problem rely on assumptions about the interactions between the controller (the machine) and its environment that are too restrictive for requirements modelling. Notably, Lin and Wonham's technique [13] relies on a distinction between prohibitive and forcible events that is irrelevant for requirements modelling. Kupferman and Vardi's technique [12] assumes that the environment and machine act in turn. Our approach is more general and allows such assumption to be modelled explicitly as domain assumptions. Our work also differs from those in its ability to take as input goals and domain models specified in a variety of formalisms such as FSP, FLTL, and KAOS. Our algorithm extends standard synthesis algorithms using determinisation and backward error propagation (notably [33]) by its handling of *tick* events in the context of deontic IO LTS.

Recent work in the area of controller synthesis have focussed on increasing expressive power (e.g. to restricted forms of liveness properties but not under partial observability [16][19][20]), on improving performance by using symbolic techniques [15] and on developing sound theories for compositional synthesis under partial observability for dense time models [17][18]. The independence progress constraint of deontic IO LTS is the discrete-time equivalent of the corresponding constraint recently introduced for dense timed I/O automata [18]. Our work differs from those by its focus on requirements modelling. Our primary objective is to define agent's permission and obligation, not to model timed-systems;

our use of *tick* to distinguish stable from transient states is a convenient way to model obligations with minimal extensions to standard LTS. This has the advantage of facilitating integration with existing requirements modelling techniques using LTS for analysis and as semantic domain.

IX. Conclusion

Automated specification synthesis techniques can provide great benefits during requirements modelling. However, such techniques needs to be grounded on an adequate formalism; the machine cannot necessarily observe all phenomena referenced in goals and a clear separation must be made between machine specifications and domain assumptions. It also needs to be compositional to be compatible with goal-oriented requirements modelling. We have introduced deontic IO LTS and a novel specification synthesis to serve these purposes. We have shown on an example how this technique helps constructing and analyzing alternative requirements models relying on different synchrony assumptions or on no synchrony assumption at all. In the future, we intend to apply this technique to larger case studies and develop techniques to explain the causes of unrealizability and to translate deontic IO LTS back to requirements modelling languages.

References

[1] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM TOSEM*, vol. 6, no. 1, pp. 1–30, 1997.

[2] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons, 2009.

[3] M. Abadi, L. Lamport, and P. Wolper, "Realizable and unrealizable specifications of reactive systems," *Systems Research*, 1989.

[4] E. Letier and A. van Lamsweerde, "Agent-based tactics for goal-oriented requirements elaboration," in *ICSE'02*, 2002.

[5] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM TOSEM*, vol. 5, no. 3, 1996.

[6] M. S. Feather, "Language support for the specification and development of composite systems," *ACM TOPLAS*, 1987.

[7] E. Letier and A. van Lamsweerde, "Deriving operational software specifications from system goals," in *FSE'02*, 2002.

[8] M. Jackson, *Problem Frames and Methods: Analysing and Structuring Software Development Problems*. Addison Wesley, 2000.

[9] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. of the IEEE*, vol. 77, no. 1, 1989.

[10] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL'89*, 1989, pp. 179–190.

[11] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," in STACS'95, 1995.

[12] O. Kupferman and M. Y. Vardi, "Synthesis with Incomplete Informatio," in *ICTL'97*, 1997.

[13] F. Lin and W. M. Wonham, "Supervisory control of timed discrete-event systems under partial observation," *IEEE TAC*, vol. 40, no. 3, Mar. 1995.

[14] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit, "Timed Control with Partial Observability," in *CAV'03*, 2003.

[15] F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime, "Efficient On-the-Fly Algorithms for the Analysis of Timed Games," in *CONCUR'05*, 2005.

[16] O. Maler, D. Nickovic, and A. Pnueli, "On Synthesizing Controllers from Bounded-Response Properties," in *CAV'07*, 2007.

[17] W. Kuijper and J. van de Pol, "Compositional Control Synthesis for Partially Observable Systems," in *CONCUR'09*, 2009.

[18] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *HSCC'10*, 2010.

[19] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models," in *FSE '10*, 2010.

[20] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models for fallible domains," in *ICSE'11*, 2011.

[21] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd Edition. John Wiley & Sons, 2006.

[22] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Deriving event-based transition systems from goal-oriented requirements models," *ASE Journal*, vol. 15, no. 2, May 2008.

[23] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI QUARTERLY*, vol. 2, 1989.

[24] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *ESEC/FSE 2003*, 2003.

[25] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Fluent temporal logic for discrete-time event-based models," in *FSE'05*, 2005.

[26] R. van Ommering, "Horizontal communication: a style to compose control software," *Software: Practice and Experience*, vol. 33, no. 12, 2003.

[27] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE TSE*, 2003.

[28] C. Damas, B. Lambeau, P. Dupont, and A. Van Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE TSE*, 2005.

[29] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Monitoring and control in scenario-based requirements analysis," in *ICSE'05*, 2005.

[30] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC/FSE 2001*, 2001.

[31] S. C. Cheung and J. Kramer, "Checking safety properties using compositional reachability analysis," *ACM TOSEM*, 1999.

[32] J. Fiadeiro and T. Maibaum, "Temporal Reasoning over Deontic Specifications," *Journal of Logic and Computation*, May 1991.

[33] D. Giannakopoulou, C. S. S. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *ASE 2002*, 2002.

[34] M. Jackson and P. Zave, "Deriving specifications from requirements: an example," in *ICSE'95*, 1995.

[35] J. Kramer, J. Magee, M. Sloman, and A. Lister, "CONIC: an integrated approach to distributed computer control systems," *Computers and Digital Techniques, IEE Proceedings-E*, 1983.

[36] R. Seater, D. Jackson, and R. Gheyi, "Requirement progression in problem frames: deriving specifications from requirements," *RE Journal*, 2007.

[37] C. Damas, B. Lambeau, and A. Van Lamsweerde, "Scenarios, goals, and state machines: a win-win partnership for model synthesis," in *FSE 2006*, 2006.

[38] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *ICSE'09*, 2009.

[39] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE TSE*, 2009.