Games and Scenarios for Real-Time System Validation

Shuhao Li

Aalborg University
Department of Computer Science

Games and Scenarios for Real-Time System Validation

Games and Scenarios for Real-Time System Validation

Shuhao Li

Ph.D. Dissertation

April 20, 2010

Aalborg University
Department of Computer Science

Abstract

This thesis presents research on the validation of real-time embedded software systems in the context of model-based development. The thesis proposes scenario-based and game-theoretic approaches to system analysis, verification, synthesis and testing to address the challenges that arise from the system characteristics of environment uncertainties, complex process interactions, quantitative timing constraints, partial observability and combinations thereof.

We make timed extensions to live sequence chart (LSC) such that the inter-process behaviors and scenario-based requirements of concurrent communicating real-time systems can be modeled and specified with LSC. By translating LSC to timed automata (TAs), we reduce scenario-based model consistency checking and property verification to CTL real-time model checking problems, and reduce scenario-based synthesis to a timed game solving problem. By linking our prototype translators with existing model checker UPPAAL and game solver UPPAAL-TIGA, we show that these methods contribute to the interaction correctness and timeliness of early system designs.

The thesis also shows that testing a real-time reactive system can be viewed as playing a timed game between the tester and the system under test (SUT). We propose methods of using winning strategies as test cases for black-box conformance testing. The methods are generalized to problems where only possibly winning game strategies can be obtained. In this case continued testing requires some early-stage "cooperations" from the SUT. Furthermore, we adapt the methods to the partial observability settings where only imperfect information about the SUT is available. All these methods contribute to the improved ability to test for reactivity correctness and timeliness of the systems in question. Experimental evaluations with case studies indicate that the proposed approaches are conceptually, algorithmically and computationally viable.

Keywords: Real-Time Systems, Embedded Systems, Validation, Scenarios, Live Sequence Charts (LSCs), Consistency, Synthesis, Timed Labeled Transition Systems (TLTS), Timed Automata (TAs), Timed Games, Timed Game Automata (TGAs), Controller Synthesis, Strategies, Model-Based Testing, Test Cases

Dansk sammenfatning

Denne afhandling omhandler forskning i validering af realtidsbaserede indlejrede software-systemer i forbindelse med modelbaseret udvikling. Afhandlingen præsenterer scenariebaserede og spilteoretiske tilgange til systemanalyse, verificering, syntese samt testning for derved at kunne håndtere de udfordringer, der opstår ved systemkarakteristika for ukontrollerbare miljøer, komplekse procesinteraktioner, kvantitative tidsrestriktioner, partiel observerbarhed og kombinationer heraf.

En udvidelse af live sequence chart (LSC) med tid præsenteres, således at indbyrdes procesadfærd og scenariebaserede krav til samtidige og kommunikerende realtidssystemer kan modelleres og specificeres i LSC. Ved at oversætte LSC til tidsautomater (TAs), reduceres verifikation af konsistens af scenariebaserede modeller samt verifikation af egenskaber for disse til verifikation af CTL-egenskaber for realtidssystemer. Ligeledes reduceres syntese af eksekverbare systemer fra scenariebaserede modeller til at løse tidsafhængige spil. Ved at anvende vores prototypeoversætter i samspil med modelverifikationsværktøjet Uppaal og værktøjet Uppaal - Tiga, som anvendes til at finde løsninger til spil, påvises det, at disse metoder kan bidrage til at sikre korrekt interaktion for tidlige systemdesign, samt at disse opfylder eventuelle tidskrav.

Afhandlingen viser endvidere, at systematisk gennemprøvning (testning) af reaktive realtidssystemer kan ses som et tidsafhængigt spil mellem testudføreren og systemet som afprøves. Derfor præsenteres metoder, hvori vindende strategier anvendes som test-cases for black-box conformance testing. Disse metoder generaliseres til at imødekomme problemstillinger, der kun giver anledning til muligt vindende strategier. I disse tilfælde er den fortsatte afprøvning baseret på kooperative vindende strategier, der kræver samarbejde fra systemet tidligt i spillet. Endvidere tilpasses metoderne til tilfælde, hvor kun partiel observerbarhed haves, dvs. hvor kun ufuldstændige oplysninger fås fra systemet. Alle disse metoder bidrager til i højere grad at kunne teste for korrekt reaktion ved stimuli og rettidige reaktioner fra de pågældende systemer. Eksperimentelle evalueringer gennem case-studies indikerer, at de foreslåede metoder er realistiske – både konceptuelt, algoritmisk og med hensyn til beregnelighed.

Nøgleord: Realtidssystemer, Indlejrede Systemer, Validering, Scenarier, Live

Sequence Charts (LSCs), Konsistens, Syntese, Tidsmærkede Transitionsystemer (TLTS), Tidsautomater (TAs), Tidsspil, Tidsspilsautomater (TGAs), Kontrolsyntese, Strategier, Modelbaseret Testning, Test-Cases.

Acknowledgements

My sincere gratitude goes to supervisor Kim G. Larsen. Thank Kim for his continuous guidance and support throughout the three years, for inspiring me with his great passion and enthusiasm, and for all his confidence, trust and patience on me.

I am equally grateful to co-supervisor Brian Nielsen. Thank Brian for advising, coaching and enlightening me in many respects, for the numerous thought-provoking discussions, and for the valuable comments and suggestions to each piece of my work.

I would like to thank Alexandre David for helping me understand the ins and outs of timed games, UPPAAL and UPPAAL-TIGA. Thank Saulius Pušinskas for the collaborations on live sequence charts. Also a thank you to Sandie Balaguer and Alexandre for joining our efforts in combining these interesting topics and materializing them.

During the PhD study I had a one-month external stay at MRTC of Mälardalen University. I would like to thank Prof. Paul Pettersson for hosting me and for the close collaboration on resource-constrained controller synthesis. Thank Aida Čaušević, Leo Hatvani, Cristina Seceleanu and Jagadish Suryadevara for receiving me and for the interesting discussions on components, services, resources and case studies thereof.

I wish to thank all people in the CISS/DES group at Aalborg for making it a pleasant, dynamic and stimulating research environment. Special thanks to Susanne Larsen and Rikke Uhrenholt for helping with a lot of practical issues over the years; to Anders Ravn for broadening my horizons with the many interesting conversations; and to István Knoll, Marius Mikučionis, Joseph Okika and Claus Thrane for the many technical as well as non-technical assistance.

Finally I would like to thank my wife Mianmian. Thank you so much for always believing in and understanding me, and for all your love, encouragement and support during this process.

This work has been funded jointly by the Faculty of Engineering, Science and Medicine of Aalborg University, the Center for Embedded Software Systems (CISS) at Aalborg University, and the Danish Network for Intelligent Embedded Systems (DaNES) project.

Contents

Chapter 1

Motivation

1.1 Real-time embedded systems

An *embedded system* is an engineering artifact involving computation that is subject to physical constraints [HS06]. These constraints arise from the computational processes' need to react to a physical environment and to execute on a physical platform. Due to the requirements on computation and reaction timeliness, many embedded systems have real-time computing constraints. Real-time embedded systems are widely deployed in our society. They exist in consumer electronics, in domestic appliances, in industrial product lines, and in avionics and aerospace systems.

Characteristics of the software parts of real-time embedded systems include:

- *Environment uncertainties.* Many embedded systems are *open* systems, i.e., they are designed to interact with their residing environments, or with the plants that they are supposed to control. For example, in a wireless sensor network application, the sensor nodes might be placed in environments with different temperature, humidity, illumination and radiation. For another example, an embedded controller may be deployed to control different plants that are constructed based on the same formal model that has output uncertainty and timing uncertainty of outputs;

- *Complex interactions.* For even small-scaled embedded systems, there could be a large number of possible interactions among the system processes (or agents, objects), or between the system processes and the surrounding environments. For instance in a cell phone application, there could be concurrent interactions among the user, the keypad, the chip, the memory stick and the radio-based station (environment);

- *Quantitative requirements.* Embedded systems are usually subject to some quantitative constraints such as energy consumptions, stochastic properties and timing constraints. In particular quantitative timing constraints are essential in real-time embedded systems – failing to satisfy them will lead to unacceptable systems. In this project we consider *continuous* real-time in the early-stage analysis and design models, which is suitable for characterizing asynchronous systems where time delays between events may be arbitrarily small; and

- *Imperfect information.* Due to limited-precision sensors, the measurements could be fluctuating and thus inaccurate. Due to aging and wearing-out of some instruments, the measurements could be biased and thus inaccurate. Consequently, chances are that we can have only *incomplete* information of a real-time embedded system, i.e., it is possible that we do not know in which exact state the plant is in, or what exact values the variables of the plant are assuming.

Real-time embedded systems need sound methodologies for their design, development and validation, because they are:

- *Safety-/life-/mission- critical.* Even a small software bug might result in the loss of lives, or severe damages to equipments, or environmental disasters. For example, the Ariane-5 launch failure [Lio96], the losses of the Mars Polar Lander [NAS00] and the Mars Climate Orbiter [NAS99]; and

- *Financial-critical.* Since real-time control software may be embedded (integrated) into those mass-volume manufactured items, the posterior detections of design flaws after the release of them may lead to massive recalls and thus incur substantial economic loss. For example, in 2003 Ford Motor Company recalled 43459 Lincoln Continental cars due to the airbag control problems [Aut].

1.2 Model-based development

A fundamental reason why software contain bugs is that there is a gap between software requirement and software implementation. The requirement (if a clear one ever exists) comes from the intended (end) users. The implementation is crafted by the programmers. Between users and programmers are the system analysts and designers. When evolving from the software requirements through software designs to software implementations, it is very likely that there will be semantic losses, or semantic deviations, or there will be introduced some inconsistencies. This is especially the case when the programmers build the systems in an entirely ad hoc manner.

To bridge the gap, the *model-based development* (MBD) paradigm has been proposed and advocated. The basic idea of MBD is that the developer creates a number of models for the system in question, gradually refines and validates these models in an iterative way, and finally obtains the implementation. If each step is well under control and does not introduce errors, then this can achieve the dream of "correctness-by-design".

Model-based development enjoys a lot of benefits, e.g.:

- Early-stage, tool-supported simulation, analysis and verification;
- Better traceability and diagnosability;
- Better adaptability and portability; and
- Automatic code generation.

Central to model-based development activities are the system models, which should be as faithful and analyzable as possible. To well model real-time embedded systems, there are some considerations to be made:

- *Environment uncertainties.* Since an embedded system might be deployed in different operational environments, the system (or controller) and its environment (or the plant) can be modeled separately as $\mathcal{S}ys$ and $\mathcal{E}nv$, respectively. With a clearly defined interface between $\mathcal{S}ys$ and $\mathcal{E}nv$, we can examine whether the system can properly function in different environments;
- *Complex interactions.* When building or before building a state/ transition-based behavioral model for each system process, we may wish to construct some scenario-based behavioral models to characterize how different processes communicate, collaborate and cooperate, or to specify scenario-based user requirements. We will use the very expressive graphical specification language Live Sequence Chart (LSC) [DH01];
- *Quantitative requirements.* Rather than modeling multiple quantitative requirements such as timing constraints, energy consumptions and stochastic properties at the same time, in this project we will focus only on the real-time aspect. We will model the dense-time constraints using Timed Automata (TA) [AD94]; and
- *Imperfect information.* To analyze system behaviors in a more realistic setting, we will include this characteristic in the model.

1.3 System validation

The high quality demands of real-time embedded systems call for sound and powerful validation techniques, which are both needed and (partially) enabled by the model-based development paradigm.

Various software validation techniques have been developed over the years. Roughly speaking, these include static verification techniques such as:

- *Equivalence and refinement checking*, which ask "are two given (design models of the) systems behavior-equivalent?", and "does one model refine another one?";
- *Theorem proving*, which asks "is some conjecture (the desired property) a logical consequence of a set of axioms and hypotheses (the system specification)?"; and
- *Model checking*, which asks "does a finite state model satisfy a given temporal logic property?"

and dynamic validation techniques such as:

- *Simulation*, which asks "does the software function as expected in an ideal (and often simplified, well-controlled) environment?"; and
- *Testing*, which asks "does the software or system function as expected?"

In our view, an ideal validation technique should have the following characteristics:

- *Early validation.* It is reported that the costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix in an early design phase [LRRA98]. Therefore bugs should be caught as early as possible;
- *Automated validation.* Manual validation techniques such as interactive theorem proving and manual testing are labor-intensive and error-prone. A survey by IDT (Innovative Defense Technologies) reveals that 30% - 70% of the overall software development schedule was spent on testing, and that 40% - 60% of the tests for most projects can and should be automated [DGG09]. In many areas, the "push-button validation" technologies are desirable, if they are at all possible. For a validation technique to be of practical use for large-scale, ever-changing and complex software systems, the potential for automation will be a major concern. This will also lead to reduced validation cost and time;
- *Rigorous and systematic validation.* Many static verification techniques are systematic approaches in the sense that they exhaustively explore the reachable state spaces. For ad hoc simulation and testing, the thoroughness of exercising the software largely depends on how the user steers the simulation or prepares the test cases. In contrast, simulation and testing based on algorithmic traversal of the state spaces of the system models may lead to systematic and well-steered test executions and thus contribute to improved test quality [UL06]. This latter kind of testing is usually referred to as *model-based testing* (MBT) [BJK$^+$05, UL06]; and

- *Debuggability.* Since we do not generally view validation simply as acceptance testing, it is desired that besides giving a verdict, a validation technique should also provide some diagnostics. It will be more convincing if it can provide witnesses in case it verifies a claim, and be more helpful if it can provide counterexamples in case it falsifies that claim.

Considering our validation objectives and the strengths/weaknesses of each validation technique and the existing algorithm and (in-house) tool support, we will focus on two of the above mentioned techniques: model checking and model-based testing.

System designers typically evaluate the quality of a developed software system against two kinds of requirements: functional requirements and extra-functional requirements [HS06].

Functional requirements specify the expected functionality, services and features. In other words, they characterize what the software should do and what it should not do. For a multi-object system, designers may inspect it from a global or local perspective. Accordingly, functional requirements include:

- *Interaction correctness*, which asks "do the different objects (or processes) in the system interact as intended?" This is a kind of *inter-object (inter-process) requirement* or *scenario-based requirement* [HM03]; and
- *Reactivity correctness*, which asks "does each individual object (or process) in the system accept external stimuli, then carry out certain computations, and then produce outputs as intended?" We call this *intra-object (intra-process) requirement.*

Extra-functional requirements specify: *performance*, which characterizes the efficient use of real time and of resources; *robustness*, which characterizes the ability to deliver some minimal functionality under circumstances that deviate from the nominal ones; and other properties, such as security, understandability and portability.

Among those extra-functional requirements, *timeliness* is of vital importance for real-time systems: merely producing logically correct outputs is insufficient; rather these outputs should be produced in the right time frame. Timeliness requirements are typically embodied as timing constraints on the systems.

In this project we will focus on functional correctness and timeliness.

1.4 Research objectives

Compared with ordinary reactive systems, real-time embedded systems have their special characteristics such as environment uncertainties, complex process interactions, quantitative timing constraints and partial observability. Accordingly, validating these systems is faced with new challenges.

The interaction correctness of a multi-object concurrent communicating system should be guaranteed in the very early stage of model-based development. By ensuring this "global correctness" the system architects and designers can rest assured that a next round refinement of the system models or the transition from models to implementations can start safely. This is especially true for real-time embedded systems where the quantitative timing constraints will usually interwind with and thus further complicate the complex process interactions.

The environment uncertainties should be well taken care of when designing real-time embedded systems, because these systems are supposed to interact with varying, ever-changing and behavior- and timing-unpredictable operating environments.

Compared with full observability, partial observability is a more realistic assumption for many embedded systems due to e.g., limited precision sensors, information hiding and encapsulation, and abstractions. However, the benefit of modeling with partial observability comes at the expense of adding extra difficulty to the validation of real-time embedded systems.

As argued above, in this PhD project we strive:

- To validate a given timed system against *scenario-based* requirements to ensure its *interaction correctness* and *timeliness*;
- To validate a given timed system against *environment uncertainties* to ensure its *reactivity correctness* and *timeliness*; and
- To validate a given timed system in the context of *partial observability*.

Chapter 2

System Modeling and Specification

This chapter introduces preliminaries on system modeling and specification for reactive systems in general and for real-time embedded systems in particular.

Software models are the major visible artifacts of model-based development activities. They also serve as the starting points of formal validation techniques such as model checking and model-based testing. As the outcomes of requirement analysis, models are mathematical abstractions of the systems in question. By establishing correctness of earlier stage models, developers can gain confidence on the progress so far, and can hopefully preserve the correctness by gradually refining the models until the final system implementations are obtained.

Software models could be *structural* (*static*) *models*, which describe how things are deployed, arranged and inter-connected, or *behavioral* (*dynamic*) *models*, which characterize the dynamics of the processes in the systems, i.e., how the software systems behave. Since in this project we are mainly interested in the functional and timing correctness of real-time embedded systems, we will focus on behavioral models.

A behavioral model may describe the *intra-object* (*intra-process*) behaviors of a software system, i.e., how an individual object (or process, agent) behaves under all possible circumstances, e.g., finite state machine (FSM), Statechart [Har87] and UML State Diagram [Org05]; it may also describe the *inter-object* (*inter-process*) behaviors, i.e., how different objects interact, e.g., Message Sequence Chart (MSC) [IT99], Live Sequence Chart (LSC) [DH01, HM03] and UML Sequence Diagram [Org05].

When looking for a suitable behavioral model, there are several points that we should take into consideration:

- *Expressive power.* For example, state-based and functional specifications focus on sequential behaviors, whereas history-based, state/ transition-based and operational specifications focus on concurrent behaviors [vL00]. For real-time embedded systems which usually consist of a number of concurrent processes, we tend to adopt the latter category of specification models;

- *Manageability.* Developers are supposed to construct the models in a piecewise, incremental rather than a monolithic manner. In this way the complexities of the models can be kept under a manageable level;
- *Usability.* The model should have a simple theoretical basis and thus be easy to comprehend and easy to construct. In this sense, visual formalisms such as FSM, TA, Statecharts, Petri nets, MSC and LSC are considered to be good candidates; and
- *Algorithm and tool support.* A model will be more preferable if there exist mature, powerful and versatile algorithm and tool support.

For intra-process behavior modeling of real-time embedded systems, we tend to use *state/transition-based* visual formalisms, more precisely timed automata, mainly because they can describe complex concurrent real-time systems intuitively, they support parallel composition of a number of automata, and we have in-house algorithm and tool support. For inter-process behavior modeling of real-time embedded systems, we prefer the visual formalisms, more precisely live sequence chart, mainly because it adds "liveness" to conventional scenario-based formalisms.

2.1 Labeled Transition Systems

A labeled transition system is an edge-labeled directed graph which consists of:

- a set of *states*, each of which represents a particular control location of a system (or a "process" in the terminology of process algebras);
- a set of *labels*, each of which represents an action that can be performed; and
- a *transition relation* $\rightarrow$, which describes the changes in process states.

Definition 1 (labeled transition system, LTS [Kel76]). *A* labeled transition system *is a tuple* $(S, L, \rightarrow, s_0)$*, where*

- *S is a countable, non-empty set of states (or processes);*
- *$s_0 \in S$ is the initial state;*
- *L is a set of labels (or actions); and*
- *$\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is a transition relation, where $\tau \notin L$ represents the internal actions.* □

For $s \in S$ and $a \in L$, we write $s \xrightarrow{a}$ iff $\exists s' \in S \,.\, s \xrightarrow{a} s'$. The τ-abstracted transition relation $\Rightarrow \subseteq S \times S$ is defined as: $s \overset{a}{\Rightarrow} s' \Leftrightarrow s \xrightarrow{\tau}^{*} \cdot \xrightarrow{a} \cdot \xrightarrow{\tau}^{*} s'$. The relation $\Rightarrow$ can be extended to consume a sequence of actions in the usual manner.

Fig. 2.1 is an example LTS of a coffee vending machine.

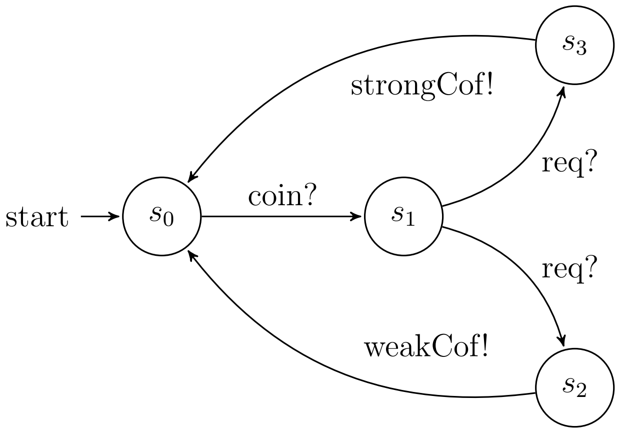


Figure 2.1: An example LTS of a coffee vending machine.

LTS is a very simple model for describing the behaviors of reactive systems. On one hand, LTS can describe non-deterministic behaviors. For instance in Fig. 2.1, it is possible that the user presses the “req” button and gets weak coffee, or she presses the “req” button and gets strong coffee. On the other hand, two or more LTSs can be parallelly composed. In this case, they can also describe the concurrency in the system in terms of the different interleaved executions.

One refinement of the LTS definition is to distinguish between input and output actions. To this end, the set L of actions can be partitioned into L_I and L_U, representing the input and output actions, respectively. To abstract away from unnecessary details of the system, a special action $\tau \notin L$ can be introduced, denoting an internal (i.e., user-undistinguishable and unobservable) action. Hence the notion of input/output labeled transition system.

Definition 2 (input/output labeled transition system, IOLTS [Tre96a])**.** *An* input/output labeled transition system *is a tuple* $(S, L_I, L_U, \rightarrow, s_0)$*, where*

- *S is a countable, non-empty set of states (or processes);*
- *$s_0 \in S$ is the initial state;*
- *L_I and L_U are countable sets of input and output actions, respectively, such that $L_I \cap L_U = \emptyset$; and*
- *$\rightarrow \subseteq S \times (L_I \cup L_U \cup \{\tau\}) \times S$ is a transition relation, such that every reachable state is weakly input-enabled: $\forall s \in S, a \in L_I \,.\, s \overset{a}{\Rightarrow}$.* □

To model the behavior of real-time systems, an LTS can be extended with the notion of time. In this way we obtain a *timed LTS*. A basic difference between a timed reactive system and an untimed reactive system is that in the former one, a state contains the control location information as well as the current timing information, and a transition could be either a discrete (and instantaneous) jump from one control location to another, or a time delay during which the control location remains unchanged.

Definition 3 (timed labeled transition system, TLTS)**.** *A* timed labeled transition system *is a tuple $(S, L, \rightarrow, s_0)$, where*

- *S is a set of states (or processes) with two dimensions: the control location and the timing information;*
- *$s_0 \in S$ is the initial state;*
- *$L = (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$ is a set of labels which consist of discrete actions $a \in Act$, internal action $\tau \notin Act$, and time delays $d \in \mathbb{R}_{\geq 0}$; and*
- *$\rightarrow \subseteq S \times L \times S$ is a transition relation which satisfies the following sanity rules for time passage:*
  - time additivity (a delay period can be arbitrarily divided)*: if $s \xrightarrow{d} s'$ and $0 \leq d' \leq d$, where $s, s' \in S$, and $d, d' \in \mathbb{R}_{\geq 0}$, then there exists $s'' \in S$ such that $s \xrightarrow{d'} s'' \xrightarrow{d-d'} s'$;*
  - zero delay (a delay period of 0 does not change the system state)*: $\forall s \in S \,.\, s \xrightarrow{0} s$; and*
  - time determinism (at any state, a given delay transition never leads to two or more different next states)*: $(s \xrightarrow{d} s') \wedge (s \xrightarrow{d} s'') \Rightarrow s' = s''$.* □

We may denote a TLTS as $(S, s_0, \rightarrow)$, since L has already appeared in $\rightarrow$.

A *timed I/O transition system* (*TIOTS*) [LMN04, BB04] is a TLTS with its set L of labels replaced by $L' = Act_I \cup Act_U \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$, which correspond to input, output, internal actions, and time delays, respectively. Also, the weak input-enabledness requirement should be satisfied.

Both LTS and TLTS are intuitive, precise and easy-to-learn formal models. In addition to being used to model the systems in question, they are also used as the underlying semantic models of other (usually richer) models such as FSM, Statechart and TA.

2.2 Timed Automata and Timed Game Automata

Timed automaton (TA) [AD94] is a popular visual formalism for specifying the intra-process behaviors of continuous real-time systems. According to Alur and

Henzinger [AH97], the underlying philosophy of TA is that a real-time system can be viewed as a discrete system with clock variables:

- The discrete system is represented as a finite directed graph. Each vertex of this graph represents a (control) *location*. Each edge represents an instantaneous *switch* (or discrete jump) which might be triggered by an enabled (observable or internal) event, or by a timeout; and

- The system has a finite set of *clock variables*. Each clock variable keeps the elapsed time value since last time this clock was reset. All clock variables increase at the same speed, reflecting the ideal situation where all the clocks in the system are perfectly synchronized. Time can elapse in a location. Each instantaneous switch may be associated with a clock constraint called *condition* (or guard), specifying the enabling condition of this discrete jump. Each location may be associated with a clock constraint called *invariant*, specifying the condition under which time can still elapse in this location.

Pragmatically, location invariants are used to allow the system to stay in a location for only a limited period of time, and then force it to leave that location.

Let C be a set of real-valued clocks, and $B(C)$ be the set of clock constraints, i.e., the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let Act be the alphabet of observable actions, and $\tau \notin Act$ be the internal (unobservable) action.

The following definition of TA is excerpted from [BDL04], which in spirit agrees with the classical definition by Alur and Dill [AD94].

Definition 4 (timed automaton, TA [BDL04]). *A* timed automaton *is a tuple* (L, l^0, C, Act, E, Inv), *where*

- L *is a set of locations;*
- $l^0 \in L$ *is the initial location;*
- C *is a set of clocks;*
- Act *is the alphabet of actions;*
- $E \subseteq L \times (Act \cup \{\tau\}) \times B(C) \times 2^C \times L$ *is a set of edges between locations. Each edge has an action, a guard and a set of clocks to be reset; and*
- $Inv : L \to B(C)$ *assigns invariants to locations.* □

Fig. 2.2 is an example timed automaton of a coffee vending machine.

A *clock valuation* is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set C of clocks to the non-negative real numbers. Let ${\mathbb{R}_{\geq 0}}^C$ be the set of all clock valuations. Let u^0 be the zero valuation, i.e., $\forall x \in C \,.\, u^0(x) = 0$. If a clock constraint $cc \in B(C)$

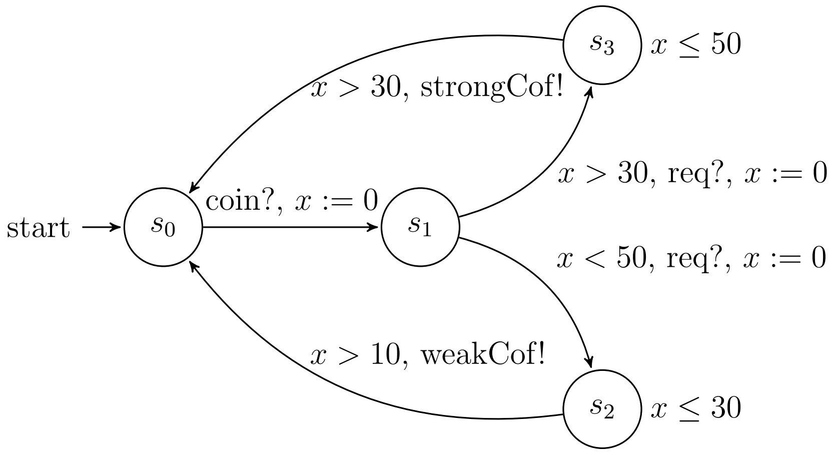


Figure 2.2: An example timed automaton of a coffee vending machine.

evaluates to true under a valuation $u \in {\mathbb{R}_{\geq 0}}^C$, then we say u satisfies cc, denoted $u \models cc$. For the sake of simplicity, we can view a clock constraint as the set of all clock valuations that satisfy this clock constraint, i.e., $\forall cc \in B(C) \,.\, cc = \{u \in {\mathbb{R}_{\geq 0}}^C \mid u \models cc\}$. For $d \in \mathbb{R}_{\geq 0}$, we use $u + d$ to denote the valuation that maps all $x \in C$ to $u(x) + d$. For $r \subseteq C$, we use $[r \mapsto 0]u$ to denote the valuation that maps all clocks in r to 0, and agrees with u over $C \backslash r$.

Definition 5 (semantics of TA [BDL04])**.** *Let $\mathcal{A} = (L, l^0, C, Act, E, Inv)$ be a timed automaton. The* semantics *of $\mathcal{A}$ is defined as a timed labeled transition system $\langle S, s^0, \rightarrow \rangle$, where $S \subseteq L \times {\mathbb{R}_{\geq 0}}^C$ is the set of states, $s^0 = (l^0, u^0)$ the initial state, and $\rightarrow \subseteq S \times (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \times S$ the transition relation such that:*

- (delay transition)*: $(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d' : 0 \leq d' \leq d \,.\, u + d' \in Inv(l)$; and*
- (discrete transition)*: $(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ such that $u \in g$, $u' = [r \rightarrow 0]u$, and $u' \in Inv(l')$.* □

Definition 6 (run of a TA)**.** *A* run *of a TA (L, l^0, C, Act, E, Inv) is a sequence of states $s^0 \cdot s^1 \cdot \ldots$ that are connected by transitions, i.e., $\forall i \geq 0 \,.\, \exists a \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \,.\, s^i \xrightarrow{a} s^{i+1}$.* □

The transition relation $\rightarrow$ as mentioned above each time consumes only a single letter $a \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. We extend it to $\rightarrow^*$ such that it consumes a (finite or infinite) word $w \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$. A word w that corresponds to a run of the TA is called a *timed trace* of the TA.

In a timed trace, the consecutive delays can be merged into a single delay, and two consecutive discrete transitions can be connected by a zero length delay transition. After these merging and insertions, the states in a timed trace will be connected by an alternating sequence of discrete and delay transitions. Such a trace is called a *normalized timed trace.*

To describe a system which consists of a number of concurrently running processes, a network of timed automata can be constructed, each for a process. The composition operator $||$ allows for interleaved executions of different processes (TAs). According to [AD94], these automata can synchronize on common actions. In UPPAAL, in addition to the CCS-style hand-shake binary synchronizations, the CBS (Calculus of Broadcasting Systems [Pra95])-style 1-to-many broadcast synchronizations[1] are also supported.

UPPAAL TA supports parallel composition by viewing *Act* as a set of synchronization channels. Hence for each element $a \in \mathit{Act}$, there are a message sending action $a!$ and a message receiving action $a?$ (i.e., the co-action). We denote $A = \{a!, a? \mid a \in \mathit{Act}\}$ as the set of all observable actions.

Assume that there are a network of n timed automata over a common set C of clocks and a common set A of actions, $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, \mathit{Inv}_i)$, $1 \le i \le n$. A *location vector* $\bar{l} = (l_1, \ldots, l_n)$ is a vector of locations of the member automata. We compose the invariants of the member automata into a common invariant over location vectors $\mathit{Inv}(\bar{l}) = \bigwedge_i \mathit{Inv}_i(l_i)$. We write $\bar{l}[l_i'/l_i]$ to denote the vector where the i-th element l_i of $\bar{l}$ is replaced by l_i'.

Definition 7 (semantics of a network of TAs [BDL04])**.** *Let* $\mathcal{A}_i = (L_i, l_i^0, C, A, E_i, \mathit{Inv}_i)$ *be a network of timed automata,* $1 \le i \le n$. *Let* $\bar{l}^0 = (l_1^0, \ldots, l_n^0)$ *be the initial location vector. The* semantics of $||_i\, \mathcal{A}_i$ *is defined as a timed labeled transition system* $\langle S, s^0, \rightarrow\rangle$, *where* $S \subseteq (L_1 \times \ldots \times L_n) \times \mathbb{R}_{\ge 0}{}^C$ *is the set of global states,* $s^0 = (\bar{l}^0, u^0)$ *the initial global state, and* $\rightarrow \subseteq S \times (A \cup \{\tau\} \cup \mathbb{R}_{\ge 0}) \times S$ *the transition relation defined by:*

- (delay transition)*:* $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ *if* $\forall d' : 0 \le d' \le d \,.\, u + d' \in \mathit{Inv}(\bar{l})$*;*
- (internal action)*:* $(\bar{l}, u) \xrightarrow{\tau} (\bar{l}[l_i'/l_i], u')$ *if there exists an* $l_i \xrightarrow{\tau, g, r} l_i'$ *such that* $u \in g$, $u' = [r \rightarrow 0]u$, *and* $u' \in \mathit{Inv}(\bar{l}[l_i'/l_i])$*;*
- (binary synchronization)*:* $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i, l_j'/l_j], u')$ *if a is a binary channel and there exist* $l_i \xrightarrow{a!, g_i, r_i} l_i'$ *and* $l_j \xrightarrow{a?, g_j, r_j} l_j'$ *such that* $u \in (g_i \wedge g_j)$, $u' = [r_i \cup r_j \rightarrow 0]u$ *and* $u' \in \mathit{Inv}(\bar{l}[l_i'/l_i, l_j'/l_j])$*; and*
- (broadcast synchronization)*:* $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l_i'/l_i, l_j'/l_j, l_k'/l_k, \ldots], u')$ *if a is a broadcast channel and there exist an* $l_i \xrightarrow{a!, g_i, r_i} l_i'$ *and a maximal set*[2]

[1] If the emitting edge transition is enabled, then that transition itself can fire. If the emitting edge transition is fired, then all enabled receiving edge transitions should synchronize.

[2] We require $\{j, k, \ldots\}$ to be a *maximal* set because if we choose to synchronize on a broadcast channel, then all the enabled receiving edge transitions are obliged to engage in the broadcast synchronization. In a very extreme case, the set could be an empty set. This happens when only the emitting edge transition itself is fired (i.e., one "speaker", zero "audience").

$\{j, k, \ldots\}$: $l_j \xrightarrow{a?, g_j, r_j} l'_j$, $l_k \xrightarrow{a?, g_k, r_k} l'_k$, ..., *such that* $u \in (g_i \wedge g_j \wedge g_k \wedge \ldots)$, $u' = [r_i \cup r_j \cup r_k \cup \ldots \to 0]u$ *and* $u' \in Inv(\bar{l}[l'_i/l_i, l'_j/l_j, l'_k/l_k, \ldots])$. □

Runs and traces of a network of TAs are defined similarly as those for a single TA.

In addition to the binary and broadcast synchronizations, the Uppaal TA modeling language also extends the classical TA with the following features [BDL04]:

- *bounded-range variables*: Integer and boolean variables can be declared. They can be used in guards, invariants and assignments;
- *urgency*: A location is *urgent* if time is frozen in that location, i.e., once entered, the location must be exited within 0 time delay. A channel is *urgent* if whenever there is an enabled synchronization on this channel, there cannot be time delay before this synchronization is fired; and
- *commitment*: A location is *committed* if it is an urgent location, and all enabled outgoing transitions from this location have higher priority to be fired than those enabled outgoing transitions from non-committed locations.

2.2.1 Timed I/O automaton

In order to adapt the timed automata formalism to the testing context, an extension of it called *timed I/O automata* (TIOA) is proposed. In the general form, the actions of a TIOA are partitioned into input and output actions such that:

- *input-enabledness*: each input is enabled in (the "interior" [3] of the invariant of) each location. Depending on whether internal actions are considered, it could be weak input-enabledness or strong input-enabledness.
- *non-blocking*: for every state, there exists an infinite execution fragment that starts in this state and contains no input actions, and in this fragment the sum of the delays diverges.

In order to ensure the "testability" of the model, the following restrictions are further imposed [SVD01]:

- *determinism*: if two transitions have the same source location and the same action label, and their guards are both satisfied, then they must lead to the same destination location;
- *isolated output*: for each state, if an output is enabled then no other input or output transition is enabled.

[3]This means that inputs are enabled as long as time can progress.

2.2.2 Timed game automaton

To tackle the controller synthesis problems for timed systems, Maler and colleagues adapt timed automata to the game settings, giving rise to the notion of *timed game automata* (TGA) [MPS95]. In a TGA, a transition will be mastered either by the game player, or by the game opponent.

Fig. 2.3 is a (simplified) timed game automaton for the coffee vending machine. The solid and dashed lines represent the player- (user-) controllable and opponent- (machine-) controllable transitions, respectively. For example, in location s_2 there are two outgoing uncontrollable transitions. Intuitively, after the user inserts a coin (*coin*?) and makes a request (*req*?), the machine can choose either to deliver weak coffee (*weakCof*!) or to deliver strong coffee (*strongCof*!) within 50 time units. The exact coffee delivered and its exact delivery time are determined solely by the machine.

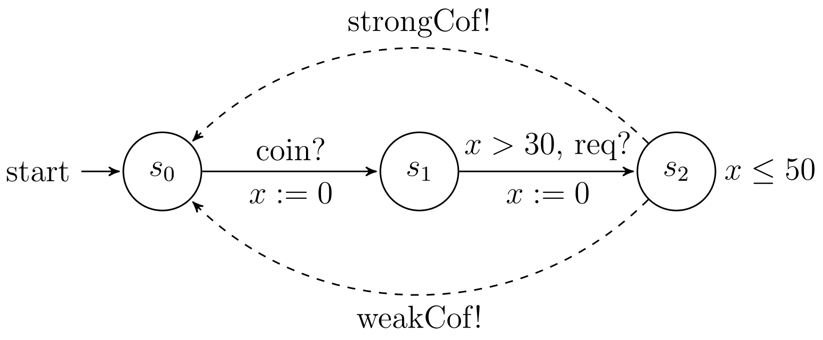


Figure 2.3: An example timed game automaton of a coffee vending machine.

In a game-theoretic context, the plant specification can be given as a TGA. If a controller can be synthesized for a particular control objective, then under its guidance (supervision), the plant TGA will lead to only "good" timed traces.

2.3 Live Sequence Charts

A *scenario* is a typical interaction among a number of processes (or agents, objects, components) within a communicating concurrent system. It describes one "story" for all relevant objects. Scenario-based models specify which scenarios are allowed, desired, or forbidden. *Inter-process behavior modeling* (a.k.a. *scenario modeling*) is an important method to characterize object interactions. Scenario modeling and validation usually take place in the early stage of a model-based development process, aiming to give the developers a system-wide and component-wise feel of how the system functions. The reason is that before diving into the very details about the required functionalities of the individual system processes, the developer wants to make sure that all these processes will collaborate, co-operate and coordinate in the desired manner and thus as a whole achieve the intended goals of the system.

One of the earliest and most widespread formalisms for scenario specification is Message Sequence Chart (MSC) [IT99], which offers an intuitive and visual way of describing possible interactions of concurrent and distributed systems. It focuses on message exchanges among communicating processes in the systems.

Fig. 2.4 is an example MSC chart which describes the scenario of opening the cover of a cell phone. Each participating process in the scenario is represented by a vertical line that is called the *lifeline* of that process. A directed m-labeled arrow from process p to q means that if the system progresses to a situation where p may send and q may receive this m-labeled message, then there could be such a message passing between p and q. The "SYNC" condition represents a rendezvous synchronization between the involved processes.

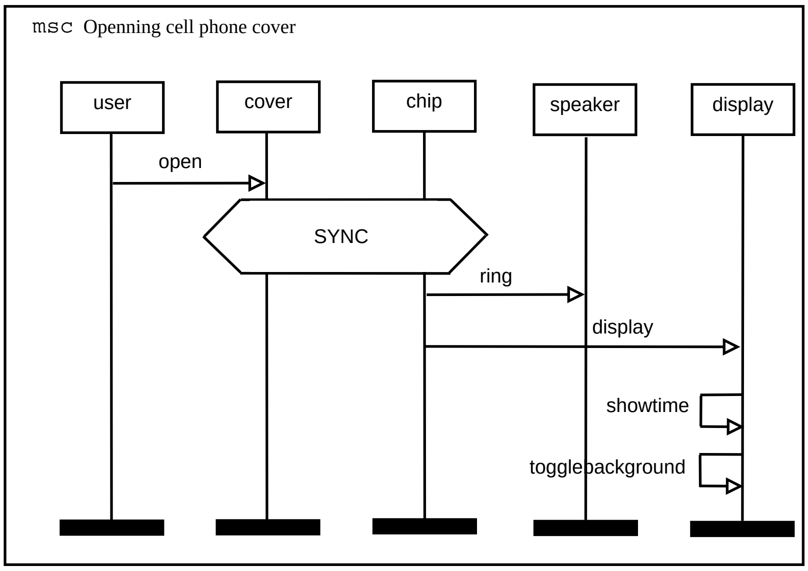


Figure 2.4: An MSC describing the scenario of opening the cover of a cell phone.

According to the MSC Specification [IT99], *basic MSC* (*bMSC*) is the building block of more complex higher-level scenario models. A bMSC has a finite set P of processes, a finite message alphabet M, and a finite set Act of actions (here an action represents a computational task). For any process $p \in P$, along its lifeline, p can perform some actions $a \in Act$, or send some message $m_1 \in M$ to other process $p_1 \in P$, or receive some message $m_2 \in M$ from other process $p_2 \in P$. These are all the actions that a process can execute along its lifeline.

To describe the timing constraints in real-time embedded systems, MSC and its variants have been extended with various syntactic constructs such as:

- *timers* [AHP96, IT96], which are used to express timing constraints within a single MSC chart and along a single lifeline. The timers can be (pre-)set to a value, or reset to 0, or observed for timeout. However, timers cannot be shared among different instances in an MSC; and

- *delay intervals* [AHP96, Ng93], which are also used to express timing constraints within a single MSC.

Basic MSC has limited expressive power. To increase the expressive power of bMSC and to improve its usability, a lot of extensions have been proposed, such as *High-level MSC* (*HMSC*) [IT99], *Dynamic MSC* [LMM02], *Triggered MSC* [SC02] and *Template MSC* [GMMP04].

MSC and its many variants have received much attention in academia as well as in industry (especially in the telecommunication industry). The main advantages of MSC include:

- *Simplicity*. There are not excessively many language constructs;
- *Intuitiveness*. As a graphical formalism, it has good usability and understandability, and it does not have a steep learning curve; and
- *Standardization* by ITU (International Telecommunications Union) [IT99].

Although MSC has been widely applied and has been proved to be an effective means for early stage modeling and validation, there are some inherent limitations to this specification language:

- *Expressive power*. The partial order semantics of basic MSC are rather weak. While MSC can specify *possible* (or *expected*) behaviors, it cannot specify *mandatory* behaviors such as "**if** process P sends message m to Q, **then** Q **must** pass on this message to R". Nor can we specify *forbidden* behaviors with MSC by means of the so-called "*anti-scenarios*";
- *Semantic gap*. MSC is an inter-process specification language which is basically considered to be declarative rather than imperative. Since MSC cannot "force" anything to happen, it lacks an executable semantics. Accordingly, MSC models do not have a perfect semantic mapping to the intra-process state/transition-based specification formalisms such as Statechart.

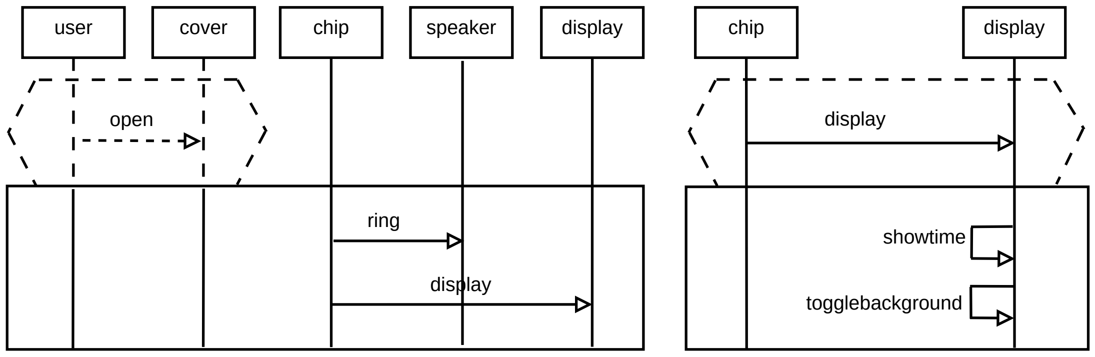
From the above limitations we can see that MSC lacks some necessary ingredients for being an ideal scenario-based system behavior modeling language. This explains why MSCs are typically used to specify expected scenarios of behaviors in the requirement stage, or used as test scenarios in the validation activities.

In view of these problems, Damm and Harel propose *Live Sequence Chart* (*LSC*) [DH99, DH01, HM03], a powerful scenario-based executable modeling (i.e., used as driving charts) and requirement specification (i.e., used as monitored charts) language. LSC makes fundamental and significant extensions to MSC by adding chart-level (universal, existential) and element-level (hot, cold) modalities and by distinguishing between possible, mandatory and forbidden behaviors.

LSC overcomes the limitations of MSC as follows:

- *Enhancing the expressive power.* LSC can represent liveness requirements. Such an expressive power of LSC is comparable to that of temporal logics and state/transition-based executable specifications such as Statechart [HT03].
- *Bridging the semantic gap.* Since LSC has executable semantics, it is possible to synthesize a state/transition-based system from a collection of LSC charts, provided that these charts are consistent [HK00, HK02, HKP05, KPP09].

Fig. 2.5 shows two example universal LSC charts. They together describe the scenario of opening the cover of a cell phone: "**if** the `user` opens the `cover`, **then** the `chip` **must** ask the `speaker` to ring, and ask the `display` to show something (in this order); and **if** the `chip` asks the `display` to show something, **then** the `display` **must** show the current time and toggle the background color".



(a) chart 1 (b) chart 2

Figure 2.5: Two universal LSC charts that describe the scenarios of opening the cover of a cell phone.

The most notable thing is that LSC extends MSC with the ability to describe possible (may) and mandatory (must) behaviors, both at the entire chart level and at the individual chart element level.

At the whole chart level, there are two types of charts, i.e., the *universal* charts (uLSCs) and the *existential* charts (eLSCs). A uLSC chart (see Fig. 2.5(a)) is used to specify requirements that *all* the possible runs of the system implementation must satisfy. It often consists of a prechart (Fig. 2.5(a), dashed hexagon) that specifies the "premise", and a main chart (Fig. 2.5(a), solid rectangle) that specifies the "conclusion" (or "obligation"). The prechart is followed by the main chart, and this captures the requirement that if along any system run the scenario depicted in the prechart occurs, then the run must match the scenario depicted in the main chart immediately afterwards. In comparison, an eLSC chart is used to specify sample interactions that *at least one* system run must satisfy. The typical usage of eLSC is to specify system tests.

At the chart element level, there are *cold* (denoted by dashed lines) and *hot* (denoted by solid lines) temperatures for the elements (i.e., condition, location, message, cut), denoting the may- and must- requirements, respectively. A hot message must be received after it is sent, whereas a cold one (Fig. 2.5(a), the "open"-labeled message) may be sent but not received, which can be viewed as a communication failure due to e.g. lossy channels. Cold and hot conditions are provisional and mandatory guards, respectively. If a cold condition is violated, then the chart context (i.e., the immediate enclosing scope of this condition, which is either a control structure like if-then-else or loop, or a subchart, or the whole chart itself) is exited, and execution may continue. If a hot condition evaluates to false, then it means that the "hard" requirements have been violated. In this case, execution may not continue and the system should abort. In modeling practice, cold conditions can be used to construct e.g. the while loops and repeat-until loops, whereas hot conditions can be used as assertions, or used to specify anti-scenarios.

A *basic universal LSC* over a finite set P of processes (or agents, objects), a finite message alphabet M, a finite set Act of chart-local actions that represent the computational steps, and a prechart Pch, is a structure $S = (E, \preccurlyeq, Pch, \lambda)$ [HT03] where:

- $(E, \preccurlyeq, \lambda)$ is a labeled partial order with $\lambda : E \to \Sigma \cup \{Pch\}$, where E is the set of events in the system, and Σ as defined w.r.t. (P, M, Act) is the set of actions of all the processes in the system;
- $Pch = (E_{Pch}, \preccurlyeq_{Pch}, \lambda_{Pch})$ is the prechart such that $E_{Pch} \cap E = \emptyset$;
- There is a unique event e_0 which is the least under $\preccurlyeq$, and $\lambda^{-1}(Pch) = e_0$; and
- $Ch = (E', \preccurlyeq', \lambda')$ is a chart called the *main chart*, where $E' = E - \{e_0\}$, $\preccurlyeq' = \preccurlyeq|_{E' \times E'}$, and $\lambda' = \lambda|_{E' \times \Sigma}$. □

In the above definition of universal LSC, e_0 represents the event that the prechart is matched in the system run. The definition requires that whenever the prechart Pch is executed, it must be followed by an execution of the main chart.

An executable (operational) semantics for LSC was defined by Harel and Marelly [HM03]. LSC can also be equipped with a trace-based semantics [KHP$^+$05]. Informally, by trace-based semantics, a chart is subject to linearization to get a trace language that it accepts. Verification problems such as asking whether a system satisfies an LSC specification can boil down to the problems of language inclusion or language emptiness, depending on whether the specification is given as a universal chart or an existential chart.

LSC has been extended with timing constructs [HM02] to model real-time systems. To follow the spirits of timed automata [AD94], a special single object `Clock` has been added, which has a single property *Time* and a method *Tick*. The object, its property and method can be referred to in the chart. By adding these constructs into LSC, it has been shown that a rich set of timing constraints (such as timers and message delays) and time-based behaviors (such as time events [HM02]) can be characterized.

It is worth noting that UML 2.0 Sequence Diagram [Org05] has adopted some important features from MSC-2000 [IT99] such as the *InteractionFragment*, the *CombinedFragment*, and the operators for choice, sequential, parallel and iterative compositions of either plain interactions or combined fragments. More noteworthy things in UML 2.0 Sequence Diagram are the neg operator that defines traces that **may not** occur, and the assert operator that defines traces that **must** occur at a given point in the scenario.

UML 2.0 Sequence Diagrams are able to express some kind of stronger (i.e., mandatory and forbidden) requirements. But since in UML 2.0 Sequence Diagrams, assert and neg are used as *operators* rather than as *modalities*, there are some confusions with the semantics of assert and neg [HM08]. This is also a reason why we prefer LSC in this project.

2.4 Requirement specification

Many formal verification techniques need two inputs: a system model and a property specification. The latter input describes which aspects we want to validate the system against. It will drive or steer the verification engine to explore the relevant state space of the system.

Property specification languages and system behavior modeling languages serve different purposes:

- A system modeling language cares about the system dynamics and implementation details, i.e., to describe the "how". During system modeling, one strives to describe the system by providing sufficient details, i.e., to yield a *complete* description of the system behaviors; and

- A property specification language just intends to query whether the system satisfies some specified properties, i.e., to describe the "what". In this activity, one typically makes a *partial* or *incomplete* description of a particular aspect of the system.

In this section we consider how to specify the user requirements on the system.

Intra-process user requirements

Intra-process user requirement considers whether a single process or several processes as a whole (i.e., the "product" process) satisfy some properties such as reachability, safety, liveness and responsiveness.

In formal verification (e.g., model checking), temporal logics are often used to specify the properties of state/transition-based models. We choose the CTL logic as the specification language. CTL has the following abstract syntax:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathsf{E}\Diamond\,\varphi \mid \mathsf{A}\Diamond\,\varphi,$$

where p is an atomic proposition.

After being added with $\wedge$, $\Rightarrow$, $\mathsf{E}\Box$ and $\mathsf{A}\Box$ in the usual manner, CTL can conveniently describe some very useful property patterns such as:

- (reachability) $\mathsf{E}\Diamond$: "eventually on some path ...";
- (safety) $\mathsf{A}\Box$: "always on all paths ...";
- (safety) $\mathsf{E}\Box$: "always on some path ...", or alternatively, "something will possibly never happen";
- (liveness) $\mathsf{A}\Diamond$: "eventually on all paths ..."; and
- (responsiveness) $\mathsf{A}\Box(\varphi \Rightarrow \mathsf{A}\Diamond\phi)$: "whenever φ, then eventually ϕ on all paths afterwards".

Inter-process user requirements

Inter-process user requirement (a.k.a. *scenario-based* requirement) considers whether a set of processes in the system interact properly.

In telecommunication areas, MSC-like formalisms are often used to specify scenario-based user requirements. In this project, we choose LSC as the specification language. LSC achieves the goal of property specification through its monitored charts. These charts act as observers: they just "listen" and never "speak". Monitored charts can also have the universal or existential types, representing the requirements over all possible system runs and over at least one satisfying system run, respectively. Furthermore, since monitored universal LSC charts and driving LSC charts share a lot of semantic interpretations, adopting LSC as requirement specification language will enable us to validate LSC-modeled system against LSC-specified requirements in a natural and seamless way.

Chapter 3

System Validation

We give an overview of a number of approaches to the validation of real-time embedded systems, namely model checking, model-based testing, and methods that combine model checking, model-based testing and controller synthesis. The necessary background knowledge and preliminaries for presenting our research questions for this thesis will be provided.

3.1 Model checking

This section introduces the principles of and tool support for model checking real-time systems against intra-process requirements. The work on checking systems against scenario-based requirements will be given in Sections 3.6.1 and 5.1.

3.1.1 Model checking real-time systems

Model checking [CGP99] is an automatic technique for verifying finite state concurrent systems. Given a behavioral model of the system such as a Kripke structure (or labeled state transition system) and a user requirement on that system such as a CTL or LTL temporal logic formula, model checking can decide whether the user requirement is satisfied by means of exhaustive state space exploration.

In the continuous real-time setting, such a behavioral model could be a timed automaton, and a user requirement could be a formula of (real-time) temporal logic, say (Timed) CTL. A basic type of user requirement is reachability. For example, $\mathsf{E}\Diamond\,\varphi$ asks whether some "good" system states as described by φ can be eventually reached. Model checking a TA against a reachability property consists of exploring the state space of the TA to see if the goal states can be reached after a sequence of action and time delay transitions.

As we know from Section 2.2, the state space $SS \subseteq L \times \mathbb{R}_{\geq 0}$ of a TA is of infinite size. This implies that we cannot solve the problem straightforwardly, because model checking algorithms operate only on finite state systems. In order to obtain a finite representation of the infinite state space of a TA, an equivalence class of TA semantic states can be aggregated as a *clock region* [AD94]. A TA has only finitely many regions. Existing state space exploration algorithms can be applied on these regions.

Model checking of timed automata is decidable, and the notion of region provides the means of concluding this, giving a finite-state abstraction with respect to time-abstracted bisimulation. Compared with the fine-grained state space partitioning using regions, an improved *symbolic* approach works on clock zones. A *clock zone* φ is an equivalence class of semantic states that correspond to the solution set of a conjunction of clock constraints each of which puts a lower or upper bound on a clock or on difference of two clocks. If a timed automaton has k clocks, then the set φ is a convex set in the k-dimensional Euclidean space. Zone partitioning is much coarser than region partitioning, therefore clock zones are more compact representation of equivalence classes of TA semantic states than clock regions. Zones can be represented as *difference bound matrices* (*DBM*) [Dil89]. The operations on zones such as intersection, clock reset and delaying can be efficiently encoded as operations on the respective DBMs.

The decidability and complexity of a number of basic problems of timed automata have been reported, e.g.:

- The *timed bisimulation* problem is decidable in EXPTIME [Cer92];
- The *untimed bisimulation* problem for timed automata is decidable in EXPTIME [LY97];
- The *time-abstracted simulation* and *bisimulation* problems are decidable for timed automata [Alu99];
- The *reachability* and *emptiness* problems for timed automata are PSPACE-complete, and they can be solved in time $O(m \cdot k! \cdot 4^k \cdot (c \cdot c' + 1)^k))$, where m is the number of edges in the TA, k is the number of clocks, c is the largest numerator in the constants in the clock constraints, and c' is the least-common-multiple of the denominators of all the constants in the clock constraints [AD94];
- The *cycle detection* problem, i.e., whether there is an infinite timed word accepted by the TA (or whether we can reach a cycle in the region graph of the TA from the initial state), is PSPACE-complete [AM04];
- The *universality* problem is undecidable [AD94]; and
- The *trace inclusion* and *trace equivalence* problems are undecidable [AM04].

In particular it has been shown that:

- The TCTL *model checking* problem is PSPACE-complete [ACD93].

3.1.2 Uppaal

Uppaal [LPY97, BDL04] is an integrated environment for modeling, simulation and verification of TA-modeled timed systems. A system is modeled as a parallelly composed network of timed automata, which have CCS-style hand-shake synchronization and CBS-style broadcast synchronization. Uppaal also supports shared-variable communication.

The Uppaal modeling language makes several extensions to the TA formalism as proposed by Alur and Dill [AD94], e.g., bounded integer and boolean variables, committed locations, and urgent channels.

Uppaal uses a fragment of the CTL logic as its property specification language. As mentioned in Section 2.4 there are five property patterns, namely, $\mathsf{E}\Diamond$, $\mathsf{E}\Box$, $\mathsf{A}\Diamond$, $\mathsf{A}\Box$ and $\varphi \leadsto \phi$. However, nested quantification is not supported in Uppaal, except in $\varphi \leadsto \phi$ which is a shorthand for $\mathsf{A}\Box(\varphi \Rightarrow \mathsf{A}\Diamond\phi)$. Furthermore, bounded liveness properties such as $\varphi \leadsto_{\leq t} \phi$ can be verified in Uppaal by using auxiliary clocks or auxiliary TA such as a test automaton [ABL98].

3.2 Model-based testing

This section will sketch out the general framework of model-based testing, with an emphasis on the principles of conformance testing with labeled transition systems [BAL$^+$90, Tre99, BT00] and their timed versions.

3.2.1 Overview

Software testing consists of the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" [AMB$^+$04].

Classical software testing are typically in the forms of:

- *manual testing*, where the tester manually derives and executes test cases based on the (informal) requirement documents; or
- *scripted testing*, where the tester achieves automated test execution by running pre-programmed test scripts using some test execution tools.

Manual testing and scripted testing have been widely used in practice. However, there are some problems inherent to them:

- *Late testing.* Constructing test cases in these two paradigms usually happens only at the very late stage of the software life cycle;
- *Manual design of test cases.* This is a labor-intensive and error-prone step, and thus constitutes a major bottleneck towards fully automating the testing process;

- *Poor adaptability.* In case the software requirement specification is changed, all test cases and test scripts need to be manually re-designed. This is a severe problem for regression testing;
- *Unguaranteed test coverage.* Since the functionality aspects of the SUT are not systematically explored, it is difficult to evaluate the thoroughness of the test activities and to gain full confidence on the system in question; and
- *Poor traceability.* Due to the limitations of test suite maintenance and management, it is difficult to always relate each test case to the system requirement specifications.

Model-based testing (*MBT*) can be defined as the process of generating test cases with oracles from a behavioral model [UL06]. A model is the starting point for testing. The oracle information enables us to decide whether a test has passed or failed.

MBT has the ingredients of an ideal validation method as mentioned in Section 1.3. Specifically, MBT features:

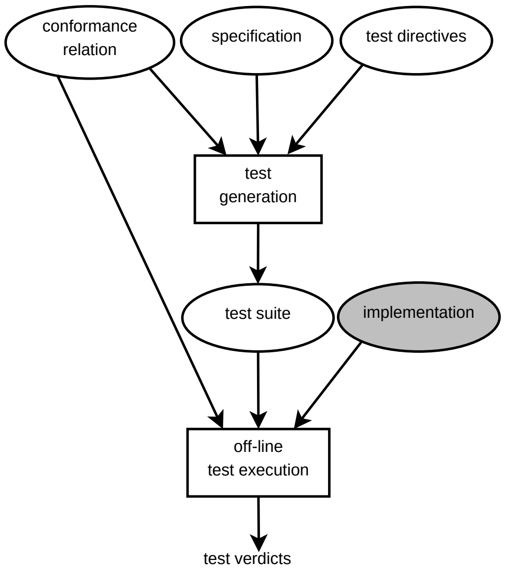
- *Early validation.* Executable software models can be validated against the requirement specifications. In this way, some design errors such as conflicting (contradictory) software requirements can be spotted in the prototyped artifacts in the very early stage of the software development life cycle;
- *Automated testing.* Automatic generation of test cases from models and automatic execution of them on the implementation under test (IUT) will significantly improve the efficiency of the test activities;
- *Better adaptability.* The requirement changes require only modifications of the models. Re-generation of tests and executing them is much less costly than those with manual or scripted testing;
- *Rigorous and systematic testing.* Many formal methods and techniques such as static analysis and formal verifications can be applied on models of software systems. Algorithmic explorations of system models can generate a large number of test cases and can guarantee required test coverage; and
- *Visualization and animation.* Some models have executable semantics. Tests may be visually animated and simulated, and counterexamples of model checking can be effectively debugged on these models.

As far as the different dimensions of software quality are concerned (see Section 1.3), model-based testing is mostly devoted to *functionality* testing [BGT04], i.e., we care whether the system does what it is supposed to do in terms of e.g. correct responses to given stimuli. Furthermore, model-based testing is mainly considered a kind of *black-box* testing, i.e., we do not use information about the internal structure of the implementation code.

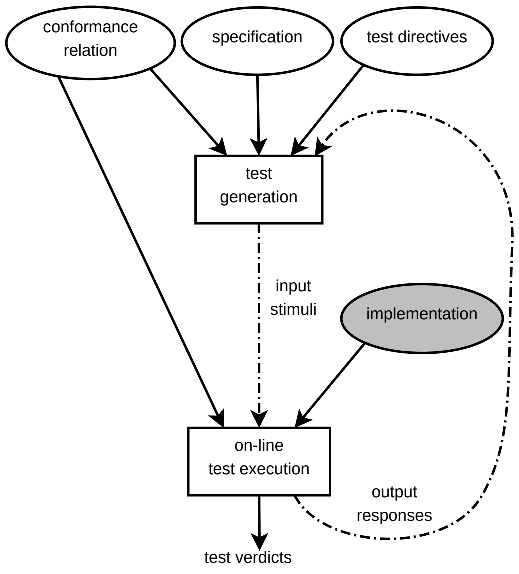
3.2.2 Conformance testing framework

In many cases, model-based testing aims to show that a system implementation behaves in compliance with its behavioral model. In this sense it is a kind of *conformance testing*. This section presents the principles of conformance testing, which mainly follow the theoretical framework by Tretmans [Tre08].

The setup of model-based conformance testing is shown in Fig. 3.1. Under the model-based testing framework, test generation and test execution could be two separate phases. In this case, it is called *off-line* testing (Fig. 3.1(a)). They can also be tightly combined and thus go hand in hand. In this case it is called *on-line* testing (Fig. 3.1(b)).



(a) off-line testing



(b) on-line testing

Figure 3.1: Schematic views of model-based conformance testing.

Let *SPEC* be the set of specification models. A specification model $s \in SPEC$ (Fig. 3.1(a)) is assumed to faithfully characterize the *expected* behaviors of a system in question. It specifies what the system should do and what it may not.

Let *IMP* be the set of implementation models. An implementation model $i \in IMP$ is assumed to accurately model the *actual* behaviors of a physical implementation.

The actual physical implementation under test (IUT) as the test subject is treated as a black box (Fig. 3.1(a)), whose internal structure is unknown to us. The only way a tester can control and observe an IUT is via its well-defined interfaces. By providing stimuli to and observing responses from IUT, the tester can conclude whether the IUT works as intended.

To enable formal reasoning of the specification and the implementation, a test assumption has to be made: any physical implementation IUT can be represented as some formal object $i_{IUT} \in IMP$.

A *conformance relation* (a.k.a. *implementation relation*) (Fig. 3.1(a)) specifies in what sense we can say that an implementation complies with its specification. It can be expressed as a binary relation between specification models and implementation models:

$$\mathsf{imp} \subseteq IMP \times SPEC$$

An implementation model $i \in IMP$ conforms to its specification model $s \in SPEC$ if $(i, s) \in \mathsf{imp}$. This is also denoted as $i\,\mathsf{imp}\,s$.

Let $TEST$ be the set of test cases. A *test case* $t \in TEST$ is an experiment that is carried out on the implementation. It includes the tester stimuli and the expected responses from the implementation. If all the observed actual responses coincide with the expected responses [4], then the implementation i is said to *pass* the test t, denoted as $i\,\mathsf{passes}\,t$; if some observed actual responses do not coincide with the expected responses, then i is said to *fail* t, denoted as $i\,\mathsf{fails}\,t$. Thus the successful (passes) and unsuccessful (fails) test execution procedures can be viewed as binary relations:

$$\mathsf{passes} \subseteq IMP \times TEST$$

$$\mathsf{fails} \subseteq IMP \times TEST$$

The aim of conformance testing is to establish a link between the conformance relation and the pass/fail test verdicts.

Given a specification model s and a conformance relation imp, a test suite T_s (Fig. 3.1(a)) is *sound* with respect to imp if, all conforming implementations pass all the test cases in T_s. In other words, if a certain implementation i fails T_s, then i indeed does not comply with the specification.

A test suite T_s is *exhaustive* (or *complete*) with respect to imp if, all implementations that pass T_s are indeed conforming implementations. In other words, if a certain implementation i does not comply with the specification, then there must exist a test case $t \in T_s$ such that $i\,\mathsf{fails}\,t$.

In Fig. 3.1 the user-specified test directives are used to focus testing efforts on a reduced set of all the possible behaviors. They could be in the forms of coverage criteria, test purposes, or fault models, among others.

3.2.3 Conformance relation

A widely used implementation relation is the *input-output conformance* (ioco) relation [Tre96b]. This relation is defined on IOLTS implementation models

[4] If the implementation behaves non-deterministically, then the condition needs to hold for all possible test runs. Similarly for fails, if the implementation behaves non-deterministically, then the corresponding condition needs to hold for at least one test run.

and LTS specification models. These models share the same input and output alphabets.

Informally, an implementation i is ioco-conforming to a specification s if, when any experiment that is derived from s is executed on i:

- if there is an output from i, this output should be allowed by s; and
- if there is no output from i, and the system does not progress unless a further tester input stimulus is provided, i.e., the system has a *quiescence* “action” at this state, then this quiescence should be allowed by s.

The essential idea of the ioco-conformance relation is that the implementation should do what it is supposed (or allowed) to do (w.r.t. the specification), and should never do what it is not allowed to do (also w.r.t. the specification).

For real-time system, a number of conformance relations have been defined on the timed labeled transition system (TLTS) implementation and specification models, such as tioco (timed ioco) [KT04], rtioco (relativized timed ioco) [LMN04] and tioco_M (timed ioco for *quiescent* real-time systems) [BB04]. The basic idea of them is to adapt the ioco-conformance that is interpreted on untimed traces of input/output actions to the timed settings.

3.2.4 Test case

A test case is a description of how the tester carries out a finite experiment on the implementation. If the specification model has totally predictable behaviors (i.e., upon receiving an input stimulus it produces at most one output response), then a test case could be a preset linear sequence of stimuli and responses. Otherwise, it needs to be adaptive and thus might be in the form of a tree.

A test case typically needs to satisfy the following requirements:

- *finiteness*. A test case should reach a test verdict within finitely many steps;
- *test verdicts*. A test case in the end should draw a conclusion on whether the test run is successful, unsuccessful, or it has deviated from the given test purpose, and then to issue a **pass**, **fail** or **inconclusive** verdict, respectively;
- *determinism*. In response to an output from the implementation, the test case should offer no more than one test stimulus, i.e., from the perspective of the implementation, a test case is a deterministic process; and
- *input-enabledness*. A test case should be able to handle output responses from the implementation at any time, thus it should be input-enabled.

In the context of conformance testing with labeled transition systems, a test case could be an IOLTS. In the timed settings, test cases could be preset linear sequences of timed inputs [SVD01, HLN^{+}03], or adaptive sequences of timed inputs which depend on the observation history [KT04, BB04], or a timed labeled transition system.

3.2.5 Test execution

Intuitively, executing a test case t on an implementation i consists in parallelly composing t with i, and letting them interact with each other via the input and output actions (i.e., letting them synchronize on the pre-defined channels). Once a terminal (sink) state of t is reached, the test execution will terminate and a test verdict will be issued.

If on the parallel composition of a test case t and an implementation i there is a run r that ends in a terminal state of t, then r is called a *test run*. An implementation i is said to *pass* t if all possible test runs of the composition $t \,||\, i$ end up with a pass-verdict.

3.2.6 Test generation

Test generation consists in algorithmically and systematically deriving test cases from a given specification model w.r.t. a given conformance relation, and under the guidance of user-specified test directives such as coverage criteria and test purposes (Fig. 3.1(a)). For on-line testing, in order to determine which input stimulus will be generated in the next step, the test generation procedure may need one more input: the output response that is produced by the implementation in the previous test execution step (Fig. 3.1(b) [5]).

Let imp be an implementation (conformance) relation. In a most generic form, the test generation can be formulated as a function:

$$gen_{\mathsf{imp}} : \mathit{SPEC} \rightarrow 2^{\mathit{TEST}}$$

A test generation algorithm is *sound* w.r.t. imp if, for all specification models $s \in \mathit{SPEC}$, the generated test suites $gen_{\mathsf{imp}}(s) = T_s \in 2^{\mathit{TEST}}$ are sound. Likewise, the algorithm is *exhaustive* w.r.t. imp if, for all $s \in \mathit{SPEC}$, the generated T_s is exhaustive. An ideal test generation algorithm should be both sound and exhaustive.

[5] In Fig. 3.1(b), although the input stimuli and output responses logically constitute a loop, they do not necessarily constitute a strictly alternating sequence of inputs and outputs. To highlight this subtlety they are drawn in dash-dot lines.

3.2.7 Test selection

Generating sound test suites is a reasonable and practical requirement. However, in most cases, generating exhaustive test suites requires that *infinitely* many test cases be derived. In practice, this is not possible. Hence we are faced with the problem of test selection, i.e., to choose finitely many finite-length test cases to exercise the behaviors of the implementations.

An ideal test selection should help minimize the test costs and at the same time maximize the likelihood of fault detection.

Depending on whether the tester has in mind some general selection criteria or some prioritized test objectives, model-based testing could be:

- *randomized testing* [TB03];
- *fault model-based testing*;
- *coverage-based testing.* In this case, test selection may be based on some structural coverage criteria such as state coverage and transition coverage of an LTS, or based on equivalence class partitioning or boundary value analysis of data-intensive systems; or
- *targeted testing*, where the tester tries to focus testing efforts on some particular portions of the system or on some particular properties of the system [HN04, JJ05].

For on-line testing of real-time systems, the test selection usually lies in the non-deterministic test case construction procedure. As such, it is usually randomized, e.g. in tools TorX [BB05] and Uppaal-Tron [LMN04].

For off-line real-time testing, test selection can be based on various criteria such as:

- *fault model*, which are often used by methods that reduce timed testing to untimed FSM-based testing, e.g. [ENDKE98] and [SVD01];
- *structural coverage* or *behavioral coverage*, which could be e.g. location (state) coverage, edge (transition) coverage or symbolic state equivalence class coverage [NS03]; or
- *test purpose*, which could be given as temporal logic formulas [HLN+03], observer automata [BHJP04] or test views [CO00].

In this project we consider targeted testing. To steer the test selection we may specify a *test purpose* which could be in the forms of *temporal logic properties* [HLN+03], *test automata* (*observer automata*) [JJ05] or *interaction scenarios* such as MSCs [DEF+96, HN04] and LSCs.

3.3 Correct-by-construction via synthesis

Formal reasonings of software designs using techniques such as static analysis and model checking are generally considered a *posteriori* approach to quality assurance. If some faults are revealed by these techniques, the design models have to be modified accordingly. This process will be repeated until the system is gradually refined to the desired level of details.

The correct-by-construction approach to software development aims at *automated system design*, where the problem is: given the model of an open system and given our requirements on the system, how can we synthesize an executable system that is guaranteed to satisfy the requirements? Clearly, this is an *a priori* approach to quality assurance.

3.3.1 Control system model

From a control engineering perspective, an embedded application consists of:

- the *plant* ($\mathcal{S}ys$), which is an *open* system to be controlled; and
- the *controller* ($\mathcal{C}$), which is the control program.

The plant and the controller are modeled separately. The plant can be viewed as the "environment" of the controller. The plant and the controller communicate via synchronization channels (i.e., coupled message sending/receiving actions). The controller provides input stimuli to and accepts output responses from the plant. The inputs are controllable by the controller, but the outputs are not controllable by the controller.

Fig. 3.2 uses an elevator system as the illustrating example. The controller issues commands goUp, goDown and stop (in solid lines) to the plant via actuators, and it observes outputs ascending, descending, approaching and arrived (in dashed lines) from the plant via sensors.

3.3.2 Controller synthesis

A *control objective* specifies what properties the system in question is supposed to satisfy. For examples,

- *reachability property*, which requires to eventually enforce some good states, e.g. in ACTL[6] formula $\mathsf{A}\Diamond$ `goodState`; and
- *safety property*, which requires to constantly avoid some bad states, e.g. in ACTL formula $\mathsf{A}\Box\ \neg$`badState`.

[6]ACTL is the universal fragment of CTL where path quantifiers can only be A.

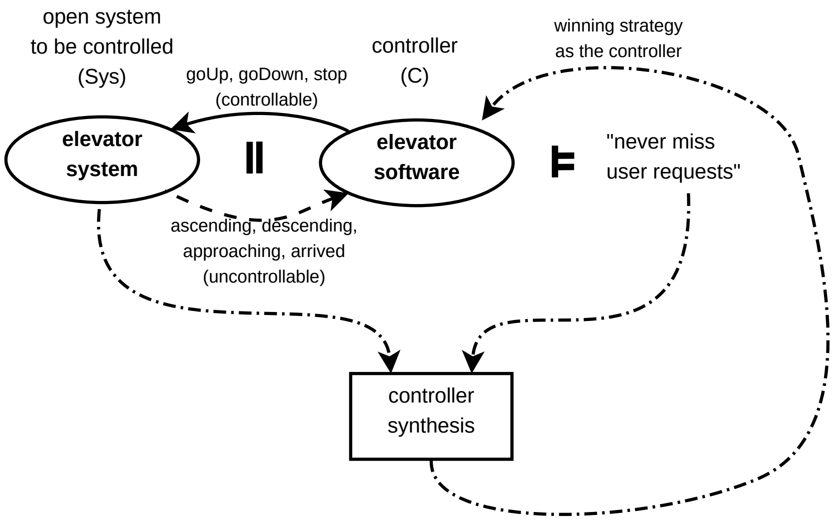


Figure 3.2: An example control problem of the elevator system.

Given a plant model $\mathcal{S}ys$ and a control objective φ, the *control problem* asks whether there exists a controller $\mathcal{C}$ such that $\mathcal{S}ys$ supervised by $\mathcal{C}$ (denoted $\mathcal{S}ys\,||\,\mathcal{C}$) will satisfy φ, no matter how $\mathcal{S}ys$ behaves. See Fig. 3.2.

The interactions between $\mathcal{C}$ and $\mathcal{S}ys$ can be viewed as game activities, where $\mathcal{C}$ is a game player and $\mathcal{S}ys$ is the game opponent. The control objective φ may be viewed as a winning condition of the game. Since only a subset of the actions are controllable by $\mathcal{C}$, the plant $\mathcal{S}ys$ could be "hostile" in the sense that it may spoil the game by not cooperating with $\mathcal{C}$.

With this game-theoretic interpretation, the control problem is equivalent to the game solving problem: there exists a controller if and only if the game is solvable.

If there exists a controller, the *controller synthesis problem* is to come up with such a controller (Fig. 3.2, dash-dotted lines). This is equivalent to finding a winning strategy for the game. Such a strategy is used as the controller.

3.3.3 Controller synthesis for discrete event systems

Controller synthesis for discrete event systems was introduced by Ramadge and Wonham [RW87] some two decades ago. Since then the problems in this field have been studied extensively. In a simple case, a plant $\mathcal{S}ys$ is modeled as a finite automaton where the actions (or the edges that they are attached to) Act are classified as the controllable ones (Act_c) that are controlled by the controller, and the uncontrollable ones (Act_u) that are only controlled by the plant (i.e., the "hostile" environment). A control objective may be a predicate over the state space specifying some "good" states to be enforced or some "bad" states to be avoided.

A strategy will give instructions on which controllable action to take. If the strategy gives instructions based on the information of a trace that has been made so far, it is said to be a *history-based* strategy; if it is based only on the information of the current state, it is a *state-based* (or *memoryless*) strategy.

A strategy is *winning* from a state s if, all strategy-supervised system runs that start from s win (or do not lose) the game. Specifically, if a strategy is winning from the initial state s_0, it is called a *winning strategy.*

It has been proven that for a finite state game and an ω-regular winning condition, the control problem and the controller synthesis problem are decidable.

Two-player games include:

- *turn-based* games, where the player and the opponent take turns firing their choices;

- *competing* game, where in each step the player can be preempted by the opponent; and

- *concurrent* games, where each player chooses a move, and the next state is the result of the combination of the two choices.

3.3.4 Controller synthesis for timed systems

A *timed* control problem asks: given a *real-time* system model $\mathcal{S}ys$ and a property (control objective) φ which is to be enforced on $\mathcal{S}ys$, whether there exists a *timed* control program (or controller) $\mathcal{C}$ such that $\mathcal{S}ys$ supervised by $\mathcal{C}$ satisfies φ, i.e., $(\mathcal{S}ys \,||\, \mathcal{C}) \models \varphi$? A *timed* controller synthesis problem consists in finding such a timed controller if it ever exists.

Maler and colleagues [MPS95] propose to characterize the control problem for timed systems using game theories. An open system is specified using a *timed game automaton* (TGA).

In order to enforce the property φ, a strategy can guide the system to take appropriate controllable actions at appropriate moment in time. It could be:

- *history-based* strategy, which is a partial function from the set of runs of the system (i.e., sequences of alternating discrete and time steps) to the set $Act_c \cup \{\lambda\}$, where λ stands for a special move which means "do nothing at this moment in time"; and

- *state-based* ("memoryless") strategy, which is a partial function from the set of states of the system to the set $Act_c \cup \{\lambda\}$.

If all possible runs of a strategy-supervised system win the game, the strategy is said to be a *winning* strategy.

Since the pioneering work on controller synthesis for dense-timed systems by Maler and colleagues [MPS95], there have been a number of improvements [AMPS98, TA99, AT02].

Initially, controller synthesis for timed systems is based on backward fix-point computations of the set of winning states [MPS95, AMPS98]. To improve the efficiency, a partially on-the-fly method for timed game solving is proposed [TA99, AT02]. However, the method involves an expensive preprocessing step in which the quotient graph of the dense time transition system w.r.t. time-abstracted bisimulation needs to be built.

More recently, a truly on-the-fly algorithm for efficient timed game solving has been proposed [CDF+05] and implemented in the tool Uppaal-Tiga [BCD+07]. This algorithm combines forward symbolic explorations and backward propagations of information of winning states and losing states, and yields very encouraging performance results [CDF+05].

Uppaal-Tiga accepts a network of timed game automata and a reachability or safety winning objective (in ACTL formulas) as inputs. It can check whether the timed game is solvable, and if yes, it can generate a winning strategy for the controller.

A *winning* state is a state from which there is a winning strategy. If the initial state is not a winning state, but a certain winning state can be reached from the initial state if the environment is willing to cooperate in some way (i.e., it is not "hostile" enough), then for the game there is a *cooperative winning* strategy from the initial state. Uppaal-Tiga is able to generate such a cooperative winning strategy if it ever exists [BCD+08].

3.4 Dealing with partial observability

In an ideal situation of the game problem, one can make the "perfect information" assumption, i.e., at any state, the controller (or strategy) $\mathcal{C}$ knows precisely what state the plant $\mathcal{S}ys$ is in, or $\mathcal{C}$ is able to deduce what state $\mathcal{S}ys$ is in by analyzing the outputs that are uncontrollable but all observable.

In a more practical setting, $\mathcal{C}$ will only have *imperfect* (or partial) information of $\mathcal{S}ys$ due to e.g.:

- limited-precision sensors, which means that the occurrences of some outputs from $\mathcal{S}ys$ may escape from our observation;
- limited-precision measurements and noises, which means that by reading a data variable we can get only an interval of possible values rather than an exact value; and
- imperfect clock synchronizations, which means that different clocks do not always progress at the same speed, and therefore we cannot read the exact values of the clocks in $\mathcal{S}ys$.

In the above cases, either not all uncontrollable actions are observable, or the $\mathcal{S}ys$ state information cannot be exactly read. Such systems are said to be *partially observable.*

Fig. 3.3 gives an example of a turn-based (untimed) game structure under partial observation. In each turn, Player 1 chooses a letter (a or b), and Player 2 resolves non-determinism by choosing the successor state. However, Player 1 cannot distinguish between states l_2 and l'_2, or between l_3 and l'_3. She can only make four possible observations (obs_1 - obs_4) on the system states. If the winning objective of Player 1 is to have observation obs_4, i.e., to reach state l_4, then for this partially observable game there is no surely winning strategy for Player 1.

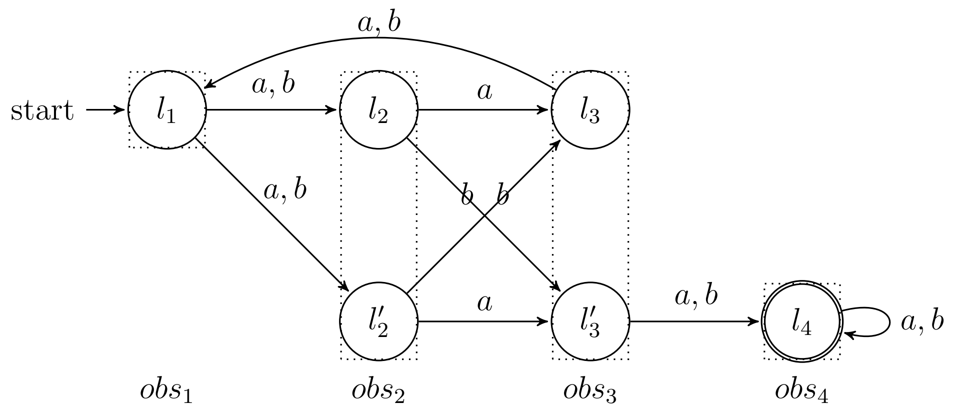


Figure 3.3: A turn-based game with imperfect information [CDHR06].

The partial observability of timed systems can be characterized in different ways:

- *controllable-observable partitioning approach* [BDMP03], i.e., to partition the alphabet Σ of actions into a set Σ_C of controllable actions and a set Σ_E of uncontrollable actions. Here Σ_E is further partitioned into a set Σ_E^o of observable actions and a set Σ_E^u of unobservable actions. Similarly, the set X of clocks of $\mathcal{S}ys$ is partitioned into a set X^o of observable (or readable) clocks and a set X^u of unobservable (or unreadable) clocks; and
- *observation-based approach* [CDL⁺07], i.e., to give a finite number of possible observations to be made on the system configurations. These observations provide the sole basis for the strategy of the controller.

Partial observability adds extra difficulty to the control problem and the controller synthesis problem. For discrete event systems, these problems are by now well studied and well understood, e.g., by means of nonemptiness test of alternating tree automaton [KV97] or subset construction [CDHR06].

The timed control problem under partial observability is in general undecidable [BDMP03]. By fixing the resources of the controller (i.e., a maximum number of clocks and maximum allowed constants in clock guards), the decidability can be regained [BDMP03]. In a recent work [CDL$^+$07], by means of knowledge-based subset construction, it is possible to transform a class of timed control problems under imperfect information into those under perfect information. An efficient algorithm to solve these games and to synthesize state observation-based stuttering-invariant (OBSI) winning strategies has been proposed [CDL$^+$07] and implemented in Uppaal-Tiga [BCD$^+$08]. These existing algorithm and in-house tool support motivate us to adopt the state observation-based approach to partial observability in this project.

Some partial observability information can be quantified such as the tolerance metrics of clock deviations in the robust semantics of *robust timed automata* [GHJ97, AM04]. While previous work on robust timed automata investigates the problems of emptiness [GHJ97] and safety [DWDMR08] checking, in this project we will study the partial observability in the context of (timed) controller synthesis and its applications in system validations.

3.5 The inter-process perspective

The development so far in this chapter (Chapter 3) concerns with only the state/transition-based models and the intra-process requirements. This section considers some *scenario-based* counterparts of those problems.

Scenario-based testing and verification

Given a system model and given a scenario-based requirement which is specified as e.g. a set of sequence diagrams, the goals of scenario-based testing and verification are to check whether the system operates in the desired manner as specified by those scenarios.

For requirements that are specified using LSCs, the verification must make sure that the LSC-required scenarios will indeed happen (within certain time frames) and that the LSC-disallowed scenarios (the *anti-scenarios*) will indeed never happen.

The LSC Play-Engine [HM03] implements the play-in/play-out methodology. It checks whether a set of *monitored* universal as well as existential charts can be satisfied by the executions that are induced by a set of *driving* (universal) charts. Monitored universal charts are used for the purpose of run-time verification, whereas monitored existential charts are used to specify tests. This kind of testing has been applied to industrial case studies [KSH07].

MSCs as test purposes can be used to specify scenario-based requirements on state/transition-based system models such as SDL, and then translated to TTCN for test execution [GH02].

Scenario-based synthesis

Synthesis from state/transition-based models has been long studied. If we use scenario-based formalisms such as LSC to model the inter-process behaviors of a system, then we are faced with the problem of synthesizing executable state/transition-based models from scenario-based descriptions. A major concern of this synthesis is how to bridge the gap between these two paradigms and their semantics.

A prerequisite for scenario-based synthesis is that the scenario descriptions are *consistent*, i.e., there are no conflicting requirements on different scenario fragments. Consistency is also called realizability, implementability, i.e., whether there exists any executable system model that satisfies the requirements.

Since being consistent is equivalent to the existence of a satisfying system model, a constructive proof of consistency can be used to synthesize such an executable system model.

3.6 Cross-fertilization

Model checking techniques have reached an encouraging level of academic and industrial maturity. In its initial form model checking operates on state transition systems (e.g. Kripke structures) and temporal logic specifications (e.g. CTL and LTL properties) [CGP99]. By means of model transformations, model checking can accommodate a wider class of modeling and specification paradigms, e.g., inter-process behavioral models and scenario-based requirements.

Although model checking is invented as a formal verification method, its many algorithms, techniques and tools could aid and support analysis problems beyond formal verification. For example, a number of model checking algorithms such as state space traversal and on-the-fly exploration have been adopted by model-based test generation; and quite a few model checkers such as SMV, SPIN and UPPAAL have been employed to generate counterexamples/witnesses as test cases.

The original purpose of controller synthesis is for automated system design. A synthesized controller can guarantee some nice properties, no matter how the environment behaves. This lends itself to the testing of *embedded* software systems which are characterized by environment uncertainties. In this case testing can be viewed as playing a game towards the test purpose (winning objective).

3.6.1 Scenario-based analysis via model checking

Scenario-based models such as LSCs characterize the system-wide behaviors naturally. Given a set of driving universal LSC charts modeling the behaviors of the system in question, and given a monitored universal or existential chart specifying the requirements on the system, we identify the following problems of scenario-based analysis:

- *consistency checking*, i.e., whether there is any internal contradiction among the set of driving charts;
- *reachability* (*satisfiability*) *checking*, i.e., whether an existential chart can be satisfied by a set of driving charts; and
- *property checking*, i.e., whether a universal chart can be satisfied by a set of driving charts.

It has been shown that by constructing a transition system which has one process for each actual object, consistency check can be encoded as a model checking problem [HKP05].

The reachability checking [HKMP02] and property verification [BS07] problems can both be reduced to model checking problems, and they are both PSPACE-complete.

3.6.2 Test generation via model checking

The capability of state space exploration and the features of counterexample/witness generation of model checkers can be utilized in model-based test generation.

It is possible to directly employing model checkers for model-based test generation. The principle is that test purposes [CSE96, EFM97, HLN$^+$03, BHJP04] or test coverage criteria (such as control-flow coverage [GH99, RH01, HLSU02, HLN$^+$03, GRR03, BHJP04], data-flow coverage [HLSU02, HCL$^+$03, HLN$^+$03, BHJP04] or mutation test coverage [ABM98, AB99]) can be encoded as temporal logic (such as CTL or LTL) formulas, which together with the system models will be fed into the model checkers (such as SMV [GH99, RH01, HCL$^+$03], SPIN [EFM97, GH99, RH01, GRR03], UPPAAL [HLN$^+$03, BHJP04]). A witness/counterexample can be generated by the model checkers as a test case for a reachability/safety property. For example, off-line generation of test cases in this way has been implemented in the tool UPPAAL-COVER [Hes07], which proves to be useful for some industrial case studies [HP06].

Another line of research is not to employ a model checker directly. Rather they adapt the model checking algorithms for test generation. This idea has been embodied in tools like:

- TROJKA [dVT00], which adapts the state space exploration algorithms of the SPIN model checker for on-the-fly testing;
- TGV [JJ05], which can handle non-deterministic specifications. A test purpose can be provided to guide test selections; and
- UPPAAL-TRON [LMN04], which exploits the UPPAAL symbolic state space manipulation and exploration algorithms to do on-line test generation and execution.

3.6.3 Testing as playing games

In software testing, the tester and the implementation under test (IUT) can be viewed as being engaged in binary synchronizations on input and output channels.

From a game theory point of view, the testing activity can be viewed as a two-player game between the tester and the IUT (Fig. 3.4). The tester as a game player masters a set of controllable actions (i.e., the input stimuli), and the IUT as the game opponent masters a set of tester-uncontrollable actions (i.e., the output responses). The tester tries to uncover faults by offering certain controllable inputs, whereas the IUT may try to "hide" them by producing some outputs that are not desired by the tester.

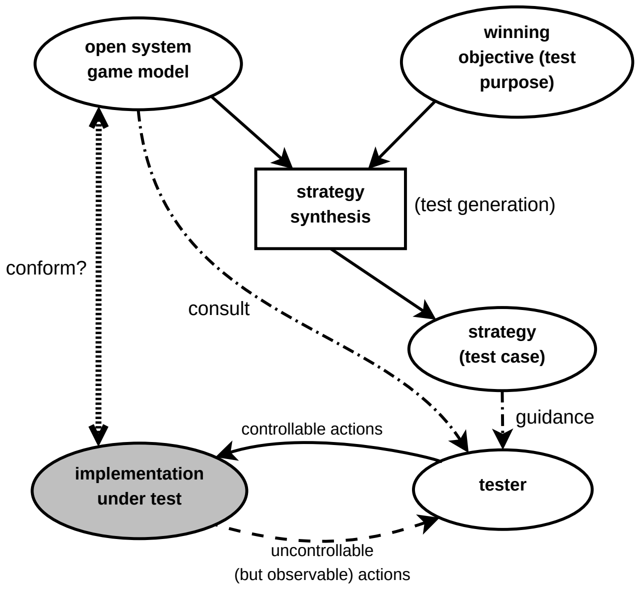


Figure 3.4: The schematic view of testing as playing games.

The tester may have a goal (test purpose), e.g., to eventually arrive at some good states (reachability objective), or to constantly avoid some bad states (safety objective). These test purposes can serve as the winning objective of a *reachability game* and *safety game*, respectively.

Given a game model and a winning objective, it is possible to synthesize a strategy (resp. an adaptive test) for the player (resp. the tester). A winning strategy will constantly guide the player what to do (i.e., which input stimulus to offer) such that the winning objective will be met. See Fig. 3.4.

If the winning objective is a reachability objective, the synthesized winning strategy can be used as a test case for *reachability testing*, i.e., the tester takes initiatives to offer test inputs and makes observations, and she is guaranteed to arrive at the goal state if the IUT is correct. If the winning objective is a safety objective, the synthesized winning strategy can be used to validate the implementation against the safety requirement, i.e., the tester takes initiatives to offer test inputs and makes observations, and she is guaranteed to constantly avoid the bad states if the IUT is indeed correct.

Chapter 4

The Thesis

4.1 Research questions

In this PhD project, we will be interested in the validation of *real-time* and *embedded* systems in the context of model-based development, where we are challenged by (combinations of) environment uncertainties, complex inter-process interactions, quantitative timing constraints and partial observability.

We would like to know *whether scenario-based and game-theoretic approaches are (conceptually as well as algorithmically) well-suited to address these challenges.*

Specifically, is it possible to adapt some of the existing methods and techniques on scenario-based analysis, model checking, model-based testing and controller synthesis to the above-mentioned settings?

In particular, we pose the following research questions:

Question 1: Can we verify a state/transition-based real-time system against scenario-based user requirements that are specified using Live Sequence Charts (LSCs)?

Question 2: Can we model and specify a real-time system entirely using LSCs, and carry out automated scenario-based analysis, verification and synthesis?

Question 3: Can we view real-time embedded system testing as playing a timed game, and synthesize winning strategies for the given test purposes, and use the strategies as test cases for conformance testing? Specifically,

(3a) What if there exist only possibly rather than surely winning strategies? and

(3b) What if we have only imperfect information of the system in question?

4.2 Thesis summary

We present a global picture of the working framework for this PhD project (Fig. 4.1). The shaded portions represent external inputs. The dash-dot rectangles sketch out the pieces of work in this PhD project, answering research questions #1 (paper A), #2 (paper B), #3 (paper C), #3a (paper D) and #3b (paper E), respectively.

As can be seen from Fig. 4.1, the work in this thesis constitutes two parts:

- *Scenario-based* approaches to system analysis, verification and synthesis, with the aims of revealing *system-wide* faults which arise from incorrectly designed *inter-process* interactions, and synthesizing executable object systems directly from the (checked-to-be-consistent) scenario-based models; and

- *Game-theoretic* approaches to reachability testing and safety validation, with the aims of checking whether the implementation of a real-time embedded system complies with its *intra-process* specification model (more precisely the timed automaton model) w.r.t. given test purposes in different settings.

These two parts are connected in the sense that:

- Game-theoretic testing can be carried out on the translated and synthesized real-time system models of timed game automata in Paper B (Fig. 4.1, dashed line connection); and

- We can possibly substitute a CTL test purpose (Fig. 4.1) with a monitored LSC chart. For example, if we replace it with an existential chart, then we can conduct game-theoretic reachability testing against scenario-based requirements.

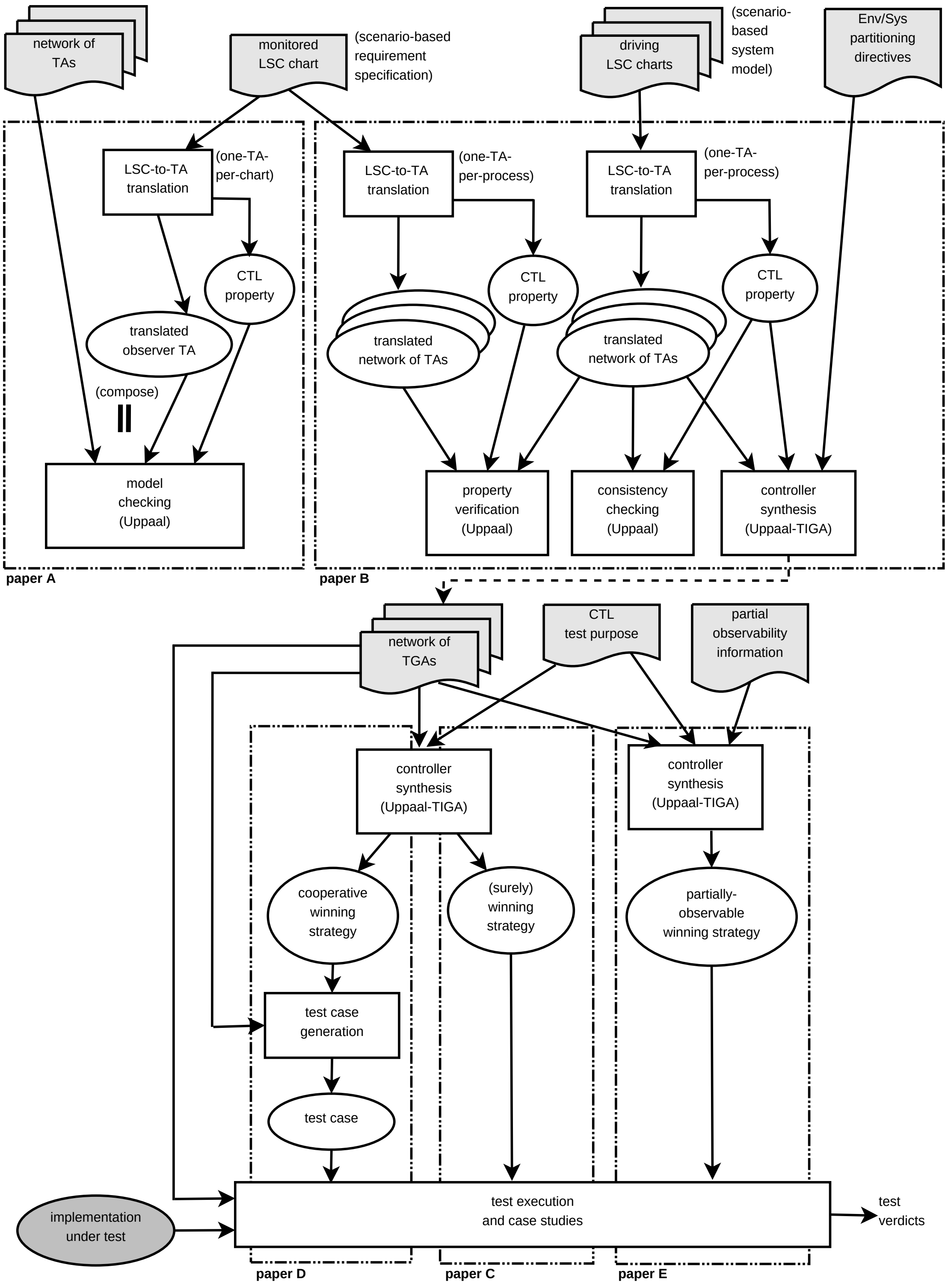

network of TAs
monitored LSC chart
(scenario-based requirement specification)
driving LSC charts
(scenario-based system model)
Env/Sys partitioning directives
LSC-to-TA translation
(one-TA-per-chart)
CTL property
translated observer TA
(compose)
model checking (Uppaal)
paper A
LSC-to-TA translation
(one-TA-per-process)
CTL property
translated network of TAs
LSC-to-TA translation
(one-TA-per-process)
CTL property
translated network of TAs
property verification (Uppaal)
consistency checking (Uppaal)
controller synthesis (Uppaal-TIGA)
paper B
network of TGAs
CTL test purpose
partial observability information
controller synthesis (Uppaal-TIGA)
cooperative winning strategy
(surely) winning strategy
test case generation
test case
controller synthesis (Uppaal-TIGA)
partially-observable winning strategy
implementation under test
test execution and case studies
test verdicts
paper D
paper C
paper E

Figure 4.1: A global picture of real-time embedded system validation.

Paper A: Verifying Real-Time Systems against Scenario-Based Requirements

Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, Saulius Pusinskas

In: *Proc. 16th International Symposium on Formal Methods* (*FM'09*), Eindhoven, the Netherlands, November 2009, LNCS 5850, Springer.

We propose an approach to automatic verification of real-time systems against scenario-based requirements. A real-time system is modeled as a network of timed automata (TA), and a scenario-based requirement is specified as a monitored Live Sequence Chart (LSC). We make timed extensions to a kernel subset of the LSC language, and define a trace-based semantics. By equivalently transforming an LSC chart into an observer TA and then non-intrusively composing this observer automaton with the original system model, the problem of verifying a real-time system against a scenario-based requirement reduces to a CTL real-time model checking problem. We show how this is accomplished in the context of the UPPAAL model checker.

Contributions

- We define a kernel subset of the LSC language that is suitable for capturing scenario-based requirements of real-time systems, and define a trace-based semantics;
- We propose to translate an LSC chart into an observer timed automaton based on the concepts of LSC cuts and advancement steps, and prove the behavior-equivalence of this translation;
- We present a method of embedding the translated observer timed automaton in the UPPAAL verification framework, and prove the behavior-equivalence of this embedding; and
- We describe a prototype tool implementation of this approach, and show how it works on an example.

Paper B: Scenario-Based Analysis and Synthesis of Real-Time Systems using Uppaal

Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, Saulius Pusinskas

In: *Proc. 13th Conference on Design, Automation and Test in Europe* (*DATE'10*), Dresden, Germany, March 2010.

We propose an approach to scenario-based analysis and synthesis of real-time embedded systems. The inter-process behaviors of a system are modeled as a set of driving universal Live Sequence Charts (LSCs), and the scenario-based user requirement is specified as a separate monitored universal or existential LSC. By translating the set of LSCs into a behavior-equivalent network of timed automata (TA), we reduce the problems of model consistency checking and property verification to CTL real-time model checking problems. Similarly, we reduce the problem of centralized synthesis for open systems to a timed game solving problem. We implement a prototype LSC-to-TA translator, which can be linked to our LSC editor and the existing real-time model checker UPPAAL and timed game solver UPPAAL-TIGA. Preliminary experiments on a number of examples and a case study show the applicability and effectiveness of this approach.

Contributions

- We propose timed extensions to a subset of the LSC language for modeling and property specification of real-time systems, and define a trace-based semantics;
- We present a method to translate an LSC system into a behavior-equivalent network of timed automata, where each process (instance line) in each chart corresponds to a timed automaton. We analyze the complexities of the translated timed automata, and prove the behavior equivalence of the translation;
- We show how to reduce the problems of scenario-based consistency checking and property verification to CTL real-time model checking problems in UPPAAL, and reduce the problem of centralized synthesis for open systems to a timed game solving problem in UPPAAL-TIGA; and
- We implement a prototype "one-TA-per-instance line" translator, and report preliminary experimental results.

Paper C: A Game-Theoretic Approach to Real-Time System Testing

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen

In: *Proc. 11th Conference on Design, Automation and Test in Europe* (*DATE'08*), Munich, Germany, March 2008.

We present a game-theoretic approach to the testing of real-time embedded systems whose models may have output uncertainty and timing uncertainty of outputs. By modeling a system in question using timed game automata (TGA) and specifying the test purpose as an ACTL formula, we employ a recently developed timed game solver UPPAAL-TIGA to synthesize winning strategies, and then use these strategies to conduct black-box conformance testing of the system. The testing process is proved to be sound and complete with respect to the given test purpose. Case study and preliminary experimental results indicate that this is a viable approach to real-time embedded system testing.

Contributions

- We show how to formulate timed testing as a timed game problem, and how to use tool to synthesize winning strategies as test cases;
- We show how to execute winning strategies as test cases in the context of real-time conformance testing;
- We prove the soundness and completeness of the proposed test methods; and
- We conduct experimental evaluation of test generation with case studies.

Paper D: Cooperative Testing of Timed Systems

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen

In: *Proc. 4th Workshop on Model-Based Testing* (*MBT'08*), Budapest, Hungary, March 2008, ENTCS 200(1): 79-92, Elsevier.

This paper deals with targeted testing of real-time embedded systems. The testing activity is viewed as a game between the tester and the system under test (SUT) towards a given test purpose (winning objective). The SUT is modeled using timed game automata (TGA) and the test purpose is specified as an ACTL formula. We employ a timed game solver Uppaal-Tiga to check if the timed game is solvable, and if yes, to generate a winning strategy and use it for black-box conformance testing of the SUT.

Specifically, we show that in case the game solving yields a negative result, we can still possibly test the SUT against the test purpose. In this case, we use Uppaal-Tiga to generate a *cooperative* winning strategy. The testing process will continue as long as the SUT reacts to the tester stimuli in a cooperative manner. In this way we can hopefully arrive at a certain state in the "surely winning" zone of the game state space, from which cooperation from SUT is no longer needed. We present an operational framework of cooperative winning strategy generation, test case derivation and test execution. The test method is proved to be sound and complete. Preliminary experimental results indicate that this approach is applicable to non-trivial timed systems.

Contributions

- We show how to test real-time embedded systems when the timed game between the tester and the SUT is not solvable with respect to the given test purposes (winning objectives);
- We present algorithms for the generation and execution of test cases in the context of conformance testing;
- We prove the soundness and exhaustiveness of the proposed test methods; and
- We conduct experimental evaluation of the methods, and report the performance results on a case study.

Paper E: Timed Testing under Partial Observability

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen

In: *Proc. 2nd International Conference on Software Testing, Verification, and Validation* (*ICST'09*), Denver, Colorado, April 2009.

This paper studies the problem of model-based conformance testing of partially observable timed systems. We model the system under test (SUT) using timed game automaton (TGA) that has internal actions, output uncertainty and timing uncertainty of outputs. We define the partial observability of SUT using a set of observable predicates over the TGA semantic state space, and specify the test purposes as ACTL logic formulas. A partially observable timed game solver UPPAAL-TIGA is used to generate winning strategies, which are then used as test cases. We propose a conformance testing framework for this particular setting, define a partial observation-based conformance relation, present the test execution algorithms, and prove the soundness and completeness of this test method. Experiments on some non-trivial examples show that this method yields encouraging results.

Contributions

- We propose a framework of conformance testing of uncontrollable timed systems which are only partially observable;
- We define an observation-based conformance relation between the specification and the implementation;
- We propose test execution algorithms based on the conformance relation, and prove their soundness and completeness; and
- We conduct case studies of test generation and report the experimental results. In particular, we show how the method scales and how it performs with different levels of controllability and observability; we also conduct comparative studies of test generations from fully observable and partially observable system models.

Chapter 5

Related Work

5.1 Scenario-based analysis and synthesis

This section surveys scenario-based analysis and synthesis methods, techniques and their tool support, both untimed and timed. Among the various visual formalisms for scenario-based modeling and specification, our emphasis will be laid on LSC.

5.1.1 Scenario-based consistency checking

A scenario-based model that consists of a set of driving universal LSC charts is *consistent* if and only if these charts are not internally contradictory, i.e., they can be satisfied by a certain state-based object system [HK00]. Consistency checking helps uncover conflicting requirements in the early stage of system design.

The smart play-out mechanism [HKMP02] implements a lightweight consistency checking for open systems. The idea is that given an environment stimulus, the Play-Engine will employ model checking techniques to compute a path (i.e., a sequence of reaction steps) for the system processes, such that along this path there will be no internal contradiction among all the involved processes. In this method, each instance line (process) will be represented by an automaton, and the consistency checking will be encoded as a model checking problem. Follow-up work [HKP04] supports smart play-out (and thus consistency checking) of *timed* systems.

The consistency checking as implemented in smart play-out is a *local* approach in the sense that the Play-Engine looks only one super-step ahead in the LSC state space. This implies that if the environment is too “hostile”, the play-out could be spoiled after interacting with the environment for a certain number of super-steps. To overcome this, Kugler and colleagues [KPP09] present a *complete* approach to consistency checking, i.e., they can thoroughly check whether the different system processes are consistent, no matter how the environment behaves.

An alternative approach [SD05a] to consistency checking of LSC charts is to transform LSCs into CSP processes, also one process per LSC instance line. Existing model checker FDR is employed to check the consistency.

The method of [SD05a] handles untimed LSC charts only. The method of [KPP09] supports time-enriched LSC which is defined in [HM03]. They use a special `Clock` object. This object has a property *Time* which is an integer variable, and a method *Tick* which each time increases *Time* by 1. Timing constraints in the charts take the form of only "*Time* **op** (time variable + delay expression)", where **op** is a relational operator.

Similar in spirit to [HKMP02, HKP05] and [SD05a], our LSC consistency checking (Paper B) also needs to translate each instance line to a process (automaton). A resemblance between [KPP09] and our method is that complete consistency checking is supported; a difference is that in our time-enriched version of LSC, there could be TA-like real-valued clock variables and clock constraints.

5.1.2 Scenario-based verification

By "scenario-based verification", we mean that either the system in question is modeled using a scenario-based formalism (more precisely LSC in this section), or we have scenario-based requirements in the forms of e.g. observer automata or LSC.

Table 5.1 summarizes the different approaches to scenario-based verification of reactive and real-time systems. Among them the combination of "inter-process model (LSC)" and "observer automata" will not be discussed here, because as far as we know there is very little research on this topic.

Table 5.1: Classification of scenario-based verification techniques.

		Requirement specification	
		observer automata	scenario-based requirement (LSC)
System	intra-process model (TA)	[ABL98, HLS99, Lah08]	[LK01, DK01, KW01, STMW04, BGS05, KTWW06]
	inter-process model (LSC)	/	[HM03, Bon05, CHK08, SD05a, WRYC04]

Intra-process model, observer automata requirement

Observer automata (a.k.a. *test automata* [ABL98]) is an approach to characterization of complex properties or scenario-based requirements. The basic idea of this verification is to construct a number of auxiliary automata to capture the scenario-based requirements, and then use these automata to "observe" the original system models. This approach should have the following characteristics:

- *compatibility*, i.e., the observer should be able to communicate with the original system through synchronization channels or shared variables;

- *non-intrusiveness*, i.e., the observer should only monitor the progress of the system model passively and thus be side-effect free. For instance, an observer timed automaton does not need to emit messages to the original system model, and it does not need to have location invariants to force the system to progress; and

- *efficiency*, i.e., the observation should incur as little communication and computation overheads as possible.

An observer timed automata approach to real-time system verification is suggested in [ABL98], which aims to reduce safety and bounded liveness property (i.e. property stating that some desired state will be reached within a given time) checking to reachability checking of the product automaton of the original system model and the observer automaton. An application of this technique for property verification has been described in [HLS99].

An observer timed automaton can describe scenario-based requirements such as "**if** process A sends message m_1 to B, and C sends m_2 to D (in any order), **then** E will send m_3 to F within $3 - 5$ time units".

The observer automata approach has been used to model check practically relevant systems such as the B&O power controller [HLS99] and some timed safety instrumented systems [Lah08]. Case studies indicate that the approach is effective.

However, there are some limitations with the observer automaton approach, especially when it comes to real-time system verifications:

- Manual constructions of observer automata could be labor-intensive and error-prone, and this is especially the case when the observer automaton grows large;

- To synchronize with the observer automata, the original system model may need to be modified and annotated. During this modification process, some new errors might be introduced. Things become more complicated if new timing errors are introduced; and

- Since the observer automata and the original system engage in normal channel synchronizations, they specify process interactions only *liberally* (i.e., no particular sending and receiving process is specified for a synchronization on a certain channel (message label)). To capture non-trivial scenario-based requirements, the synchronizations between the observer automata and the original system should be carefully designed by using e.g. auxiliary variables, semaphores or locking mechanisms.

Our method of verifying TA-modeled real-time systems against scenario-based requirements (Paper A) relies on a reduction to the observer automata approach. Compared with existing observer automata approaches, our observer automaton is constructed *automatically*, and it is guaranteed to observe the original system in a *non-intrusive* way. Furthermore, the automatically created auxiliary variables in our observer automata will enable our method to communicate with the original systems to *faithfully* reflect the LSC requirements.

Intra-process model, scenario-based requirement

LSCs can be used to capture scenario-based requirements on state/transition-based system models. Lettrari and Klose [LK01] introduce a number of LSC features into UML Sequence Diagrams, and develop a tool to monitor and test the executable UML models. Damm and Klose [DK01] propose to use LSCs to specify scenario-based requirements on STATEMATE models, and then carry out model checking. This methodology has been concretized and implemented in [KW01], where an LSC chart is transformed into a timed büchi automaton (TBA), which is further transformed into a temporal logic formula. Further descriptions of how LSC as a specification language can be used in a UML verification environment for the RHAPSODY tool are presented in [STMW04].

In [BGS05], LSC is applied in hardware verification, where the system models are given in Verilog and the user requirements are specified as LSCs. These LSCs are translated to LTL formulas and then fed into the verification environment FORMALCHECK.

Verification techniques that are based on LSC-to-temporal logic translation suffer from scalability problems. Industrial case studies [Klo03] show that the LTL formulas grow large even for LSCs of moderate size, and thus formal verification becomes expensive. To overcome this limitation, Klose and colleagues [KTWW06] investigate *efficient* model checking of Kripke structures against LSC requirements. The authors identify two sub-classes of LSCs that are easier to verify, although they are less powerful than full LTL model checking.

In our method of verifying TA-modeled real-time system against LSC-specified user requirements (Paper A), we translate a monitored LSC chart to an observer automaton and then compose it with the original system in a non-intrusive way. Since our observer automaton is tightly coupled with the original system, a very simple CTL property[7] can be extracted from the observer automaton to capture the LSC requirements. In this way we avoid translating LSCs to complex temporal logic formulas.

Inter-process model, scenario-based requirement

[7]For example, for a monitored universal chart $\mathcal{L}$, if the minimal cut of the main chart of $\mathcal{L}$ corresponds to location l_{min} in the observer automaton, and the maximal cut corresponds to l_{max}, then we can extract from the observer automaton the CTL formula $\mathsf{A}\Box(l_{min} \Rightarrow \mathsf{A}\Diamond\, l_{max})$.

In this case, the system is modeled as a set of driving universal LSC charts and the requirement is specified as a set of monitored universal or existential charts. Monitored universal charts should not be hot-violated, and monitored existential charts should be matched at least once [HM03, Bon05].

In the execution (or play-out) [HM03] of scenario-based models, the Play-Engine checks whether the monitored charts are respected. This is enhanced in the *smart* play-out mechanism, where planned state space exploration via model checking is added to the Play-Engine to bypass some avoidable hot violation situations that are caused by some “blind” interactions among the system processes. In a case study of a telecommunication application, Combes and colleagues [CHK08] check whether a set of monitored existential charts can be satisfied by a set of driving charts without violating any of them. Their verification method is based on the play-out and smart play-out mechanisms in the Play-Engine. The method supports timing constraints.

As mentioned earlier, LSC can be encoded as CSP processes. The CSP verification tool FDR can be employed to check whether a set of monitored existential charts can be satisfied by a set of driving universal charts [SD05a]. This work considers untimed charts only.

Wang and colleagues [WRYC04] employ constraint logic programming (CLP) techniques to enable the symbolic execution of LSC models. Their implementation supports both universal and existential charts, and supports timing constraints.

Compared with [SD05a, CHK08], our method (Paper B) of property verification (i.e., to check whether an LSC-modeled system satisfies an LSC-specified requirement) allows the requirements to be specified as universal charts. Furthermore, our method uses TA-like clock variables and clock constraints.

5.1.3 Scenario-based synthesis

Automated synthesis of state/transition-based executable object systems from scenario-based model is the key objective of a long-known dream of “automated system design”.

According to [BS07], the synthesis problems can be classified as

- *centralized* synthesis, where a single strategy (which can be represented as a single automaton) will be used to supervise all the system processes; and

- *distributed* synthesis, where the strategy will be distributed among all the system processes such that each process will be individually supervised by a “local” strategy.

Bontemps and colleagues [BS07] present a number of theoretical results on the decidability and complexity of synthesis from LSC specification models. For an open system, the problem of centralized synthesis is complete for EXPTIME, and the problem of distributed synthesis is undecidable.

Algorithmic synthesis for LSC is first studied by Harel and colleagues [HK00] and later detailed in [HK02]. They define the notion of *consistency* of LSC models, relate it to the problem of realizability or implementability, and then generate a state-based object system (e.g., a collection of finite state machines or statecharts) as a witness of proving this realizability. Since this approach needs to construct a global system automaton, it suffers from the state explosion problem. Moreover, the approach is mainly a theoretical contribution. The authors continue with that research by applying new verification-based techniques [HKP05]. They present a sound but not yet complete algorithm for statechart synthesis, and make a prototype implementation. To tackle the problem of incomplete synthesis of [HKP05], Kugler and colleagues [KPP09] apply the results from controller synthesis to the LSC synthesis problem. In this way, they can guarantee that if an LSC requirement model is indeed realizable, then they can synthesize an executable state-based object system that satisfies the model, no matter how the environment processes behave. This approach has been implemented in the Play-Engine as an extension to the smart play-out mechanisms. The method can support timing constraints.

A game-theoretic approach to synthesis from LSCs is presented in [BSL04]. By equipping LSC with a game-based semantics, the synthesis problem is reduced to a parity game problem. Initially, only a small subset of LSC features are considered, e.g., there is no LSC element of conditions [BSL04]. More features of LSC have been added later [Bon05]. The incomplete distributed synthesis has been implemented in the tool REMoRDS [Bon05].

Other attempts on synthesizing distributed processes from LSC by using CSP as the carrier have been reported in [SD05b, WQSD07]. CSP algebraic laws are used to group the behaviors of each object. This overcomes the problem of constructing a global system automaton [HK02]. However, there is no real-time support.

More recently, a *compositional* approach to synthesis has been proposed [KS09], where small parts of the scenario-based specification can be synthesized separately and then composed together.

Our method of scenario-based synthesis (Paper B) is a kind of centralized synthesis for open systems. Compared with [KPP09], our complete synthesis supports TA-like clock variables and clock constraints.

5.2 Model-based testing of real-time systems

This section surveys some existing work on conformance testing of real-time systems based on the timed automaton models.

5.2.1 Models

Different testing methods may need different variants of the timed automaton models. These models may characterize the systems in question at different granularities or at different levels of abstraction.

Since testing is an interaction activity between the tester (or environment) and the system under test (SUT), it makes sense to model the inputs (stimuli) from tester to SUT and the outputs (reactions) from SUT to tester separately. Hence the refined versions of timed automaton where inputs are distinguished from outputs. There are a large body of testing work based on the TIOA (timed input/output automaton) models [SVD01, ENDKE98, HNTC99, LMN04], the TAIO (timed automaton with inputs and outputs) model [KT04], the TIOSM (timed input/output state machine) model [CKL98] or the more fundamental timed input/output transition system (TIOTS) models [BB04].

When modeling reactive and real-time systems, an important decision is:

- Should we assume perfect knowledge of the system, and thus model it as a *deterministic* system? or
- Should we abstract away some low-level details of the system, and thus model it as a *non-deterministic* system?

Depending on how much non-determinism are allowed, we classify the timed automaton models as:

- *(Fully) deterministic* timed automata [SVD01, HLN$^+$03]. In order to ensure "testability", the timed automaton models should be *controllable* in the sense that it should be possible for an environment (tester) to drive a timed automaton through all of its transitions [SVD01]. Controllability [8] requires a timed automaton (more precisely a TIOA) to have the following features:
  - *determinism*: the same source location and the same action will lead to the same target location;
  - *isolated output*: for each state, if an output is enabled, then no other input or output transition is enabled at that state; and
  - *output urgency* (or "input-enabledness only in the *interior* of the invariant of each location" [SVD01]): for each state, if there is an output, this output must be produced immediately and cannot be delayed.

  The full determinism assumptions lead to a number of nice properties of the system models. However, they come with some disadvantages: firstly,

[8]In this thesis, if a timed automaton model does not satisfy the "isolated output" or "output urgency" requirements (i.e., in some state of the timed automaton there are output uncertainty and/or timing uncertainty of outputs), then it is said to be an *uncontrollable* timed automaton.

it requires more efforts to ensure that the models are indeed fully deterministic; furthermore, systems specified in this way sometimes appear to be *over-specified*;

- *Restricted non-deterministic* timed automata [NS01a, Kho02, NS03]. In order to model the systems in a natural and faithful manner, and at the same time to regain the nice properties of fully deterministic timed automata, some determinizable subclass of timed automata are used for modeling the systems and for subsequent test generation from those determinized models. For example, the Event Recording Automaton (ERA) [AFH94] in [NS01a, NS03], the Determinizable Timed Automaton (DTA) [KJM03] and the set-exp-Finite State Automaton (se-FSA) [Kho02] that does not require output urgency; and

- *Non-deterministic* timed automata [CO00, KT04, LMN04]. In many cases of system modeling, non-determinism is desired in order: (1) to naturally describe the unforeseeable interleaved executions of different components of a concurrent system; (2) to allow implementation freedom (i.e., not to over-specify the systems in question); and (3) to focus only on the interesting behaviors of the systems. Non-deterministic timed automata can suit these needs.

The timed I/O game automata (TIOGA) model that we use in this thesis has output uncertainty and timing uncertainty of outputs. Despite some restrictions and assumptions such as deterministic transition (i.e., same source and same action lead to same destination) and input-enabledness, our model can be viewed as non-deterministic in the sense that after the tester offers an input stimulus, she cannot predict the exact out response.

5.2.2 Conformance relations

Various conformance relations for real-time system testing have been proposed over the years [ST08]. A number of them can be viewed as timed extensions to the classical ioco relation [Tre99] that is used for untimed systems:

- tioco (timed ioco) [KT04], which is defined by including time delays in the set of observable outputs;

- rtioco (relativized timed ioco) [LMN04], which takes the explicitly modeled environment of the SUT into account.

For both conformance relations, the incorporation of time makes it possible to check whether an output from the SUT is produced too early or two late. Only timely outputs are considered valid behaviors. Otherwise the SUT is considered non-conforming to the specification.

Another timed extension to ioco is tioco_M [BB04], which is developed for *quiescent* real-time systems. The basic idea is that to conclude that there is a quiescence in a certain state, we do not need to wait *forever* without observing an output. Rather we can make that conclusion as soon as a given period of time M has elapsed, during which there is no output.

Comparisons of several conformance relations for real-time systems have been made in [KT06, ST08], where it has been shown that tioco and rtioco are essentially equivalent.

The conformance relation that we use in Papers C and D is tioco. In Paper E, we propose a new conformance relation poco, which requires state observations to be made on partially observable timed systems.

5.2.3 Test case representation

A real-time test case t is derived from the real-time system model, and will be applied on the system under test. Basic requirements are that: t should be an experiment which specifies the tester inputs and desired delay, and should have a test verdict of pass or fail in the end.

Depending on whether the system under test has a deterministic model or non-deterministic model, test cases could be:

- (for deterministic systems) *preset* linear sequences of timed inputs [SVD01, HLN$^+$03]; or
- (for non-deterministic systems) *adaptive* tests which depend on the observation history [KT04, BB04]. They can be defined as a function which maps a timed trace observed so far to a certain suggested input action, or the instruction of “delay at this moment”, or a test verdict [KT04]. They can also be defined as a timed labeled transition system (TLTS) [BB04]. Essentially, tests for non-deterministic systems are tree-structure descriptions of the tester's strategy against the SUT.

In this thesis, a winning strategy will be used as an adaptive test case. The next move suggested by the test case depends on the current observation (Papers C and D) of the system, or on the observation history made so far (Paper E).

5.2.4 Test generation and execution

To generate preset test sequences, classical FSM-based test generation algorithms such as the W-method [Cho78] can be applied on the discretized model of a deterministic timed system [SVD01]. Real-time model checkers such as UPPAAL can also be used to generate a witness/counterexample as a test sequence [HLN$^+$03].

The non-deterministic test case construction procedure for untimed labeled transition systems [Tre96b] can be adapted to the timed settings [BB04, KT04]. To generate adaptive test trees, the procedure involves the non-deterministic and recursive application of the three steps, i.e., to offer an input, to wait for an output, or to terminate the test case and announce pass. Compared with [Tre96b], some timing constraints are added to the former two steps in [BB04, KT04]. The soundness [BB04, KT04] and exhaustiveness [BB04] properties are proved of the test generation procedures.

Test generation from and execution on real-time systems can be two separate steps of the software testing process (Fig. 3.1(a)). In this case, test cases are first derived by the test generators, and then fed into the test execution engines. This is usually referred to as *off-line* testing [HLN$^+$03]. With off-line testing, one can do a priori test selection and posterior test coverage evaluation based on certain criteria. However, compared with untimed systems, off-line testing of real-time systems is more likely to suffer from the scalability problems: on one hand, for large systems, off-line testing can easily encounter the state space explosion problem when it tries to systematically explore the whole state space; on the other hand, off-line generated timed test cases (trees) may be very large.

An alternative approach is to combine test generation and execution and let them progress hand in hand (Fig. 3.1(b)). In this case, one test input is generated at a time, then it is offered to the SUT, and then the output from the SUT together with its timing is observed and checked against the specification model to see if it is allowed. Only if it is allowed, a next test input can be generated. The process repeats in this way. This is known as *on-line* testing [LMN04, BB05]. On-line testing suffers less from the state space explosion problem. Furthermore, it can be used on non-deterministic models. The disadvantages are that the implementation of this method is likely to introduce extra communication and computation delays that may need to be compensated, and the fault diagnosis is in general more difficult.

Our methods can be viewed as off-line testing, because the test case (strategy) generation does not need to interact with the SUT. Since our methods allow output uncertainty and timing uncertainty of outputs, they can be applied on a wider class of models than those of conventional off-line testing methods such as [SVD01, HLN$^+$03]. Furthermore, in our targeted testing methods, for each given test purpose only one test case will be generated. Using particular test purposes instead of the more generic test selection criteria such as structural coverage or fault models ensures that our off-line methods generate only a limited number of test cases.

5.2.5 Tools and experiences

To implement the automatic test case generation technique which features symbolic reachability analysis and equivalence class partitioning [NS01a], a prototype test generation tool RTCAT [NS01b] has been developed. Application of the technique and tool to a realistic case study, the Philips Audio Protocol, shows that encouragingly small test suites can be generated.

The on-line on-the-fly testing technique is an important feature of the (untimed) TorX test methodology and tool. Researchers have successfully adapted this technique to the timed settings. Tools and experiences include:

- Uppaal-Tron (**t**esting **r**eal-time systems **on**line) [MLN04], which has been applied to an industrial electronic refrigeration controller case [LMNS05]. The general finding is that real-time online testing is an effective means of detecting discrepancies between the model and the implementation in practice; and

- (timed) TorX [BB05], which conducts on-the-fly testing of real-time systems where quiescence is allowed.

In addition there are on-the-fly test generation tool and experience for real-time systems such as:

- TTG (**t**imed **t**est **g**eneration) [KT04, KT09], which implements the on-the-fly generation of (digital-clock) tests. The tool has been applied to Bounded Retransmission Protocol (BRP), a protocol for transmitting files over an unreliable (lossy) medium. Experiments with BRP show that only a few tests suffice to cover thousands of reachable symbolic states in the specification.

Uppaal-Cover [Hes07] is a tool for off-line testing of real-time systems. A number of guidance for test selection are supported by Uppaal-Cover, such as test purposes in the forms of temporal logic formulas, coverage criteria and observer automata. An industrial case study with a WAP protocol [HP06] using Uppaal-Cover has successfully revealed several discrepancies between the model and the actual implementation.

5.3 Games, controller synthesis and their applications

5.3.1 Games for untimed testing

The idea of using games for (untimed) testing is initially proposed by Alur and colleagues [ACY95], where a major aim is to generate *adaptive* test cases for *non-deterministic* systems. A more detailed description of using games in testing is given by Yannakakis [Yan04], where the systems are modeled as a (deterministic or non-deterministic) FSM. Quiescence at the IUT side is modeled as a timeout or null output. It has been shown that the complexity of the existence of almost surely (a.s.) winning adaptive strategies is EXPTIME-complete [ACY95].

Test generation by means of turn-based reachability games has been studied in [NVS+04, BGNV05]. Abstract State Machine (ASM) is used as the system model. In order to generate optimal strategies, reachability games are formulated, analyzed and solved by means of linear programming and value iteration methods. This approach is supported by the tool SpecExplorer. There are extensions of this work to on-the-fly online testing [VCST05] and optimized test design by reinforcement learning [VRC06].

The work of synthesizing reactive planning tester for non-deterministic systems [VRKE07] is similar to a game-theoretic approach. The difference is that it makes a trade-off between random selection of test stimuli and perfect planning of test stimuli (e.g., in terms of winning strategies) such that models with abundant data variables (e.g., EFSM) can be handled.

Compared with existing work, our methods have a number of distinguishing features: firstly, they can be applied to real-time systems; secondly, they can handle both reachability and safety games; thirdly, they can be adapted to both the cooperative and the partial observability cases.

5.3.2 Timed controller synthesis and its applications

The timed game solver Uppaal-Tiga accepts a network of timed game automata and an ACTL control objective as inputs. For partially observable systems, the inputs also include a set of observable predicates on the system state space. Uppaal-Tiga has the following features [BCD+08]:

- synthesizability checking;
- winning strategy generation. A strategy could be stored in the forms of federations of clock zones or clock difference diagrams (CDDs) [LPWY99]. It can also be output as pseudo code;
- cooperative winning strategy generation. In this case, the first part of the strategy claims only “possibly winning”, and the second part guarantees “surely winning”;
- strategy generation for partially observable timed systems; and
- strategy generation for büchi winning conditions.

In combination with Simulink and Real-Time Workshop, Uppaal-Tiga has been used to construct a tool chain for synthesis, simulation and automatic generation of production code for a climate control system [JRLD07]. Uppaal-Tiga has also been used for autonomous robot control [AAG+07].

More recently it has been shown that UPPAAL-TIGA, when combined with PHAVER and SIMULINK, can be used to achieve the tasks of automatic synthesis of provably correct, robust and near-optimal controllers for a real industrial case study — an Oil Pump Control problem [CJL+09].

In addition to being used to synthesize strategies for timed control problems, timed games and UPPAAL-TIGA can be used to study the relationships between timed games themselves. In [CDL09], the problem of checking whether one timed game automaton simulates another one can be reduced to the problem of solving a reachability game, which can be tackled in UPPAAL-TIGA. The tricky encoding of the simulation checking problem as a (competing) timed game has been improved in [BCDL09], where the problem is instead encoded as a turn-based game.

Furthermore, the problems of consistency checking and composability checking in the fields of model-based development based on timed interface theories can also be encoded as timed games and thus solved using UPPAAL-TIGA [DLL+10].

Compared with the afore-mentioned applications, our methods put timed games and controller synthesis in a model-based testing context. This enables us to test real-time embedded systems that are characterized by environment uncertainties, quantitative timing constraints and partial observability.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

As mentioned in Section 4.1, the scope of this PhD project is validation of real-time embedded software systems in the context of model-based development. The challenges are that while validating these systems, we have to consider (combinations of) the environment uncertainties, the complex inter-process interactions, the quantitative timing constraints and the partial observability. The overall objective is to come up with methods, techniques and prototype tools for early-stage verification and testing of the behaviors of such systems.

By answering the research questions in Section 4.1, in short, we conclude that *game-theoretic and scenario-based approaches are well-suited to address these challenges conceptually, algorithmically and computationally.*

As depicted in Fig. 4.1, the work in this thesis constitutes two related parts: *scenario-based* approaches to system analysis, verification and synthesis; and *game-theoretic* approaches to reachability testing and safety validation.

In the first part of work (Research questions #1, #2), we make timed extensions to LSC such that TA-like real-valued clock variables, clock constraints and clock resets can be used in the LSC models. We define trace-based semantics for the time-enriched LSC. Based on these semantics, we propose the "one-TA-per-LSC chart" (Paper A) and "one-TA-per-LSC instance line" (Paper B) methods for translating LSC charts into behavior-equivalent timed automata.

We find out that the existing semantics of TA, our proposed time-enriched LSC and its semantics, and the existence of the powerful real-time model checker Uppaal make it conceptually, algorithmically and computationally feasible for us to reduce the problems of: (1) verifying a state/transition-based real-time system against LSC requirement, (2) consistency checking of a set of driving LSC charts, and (3) verifying a set of driving LSC charts against a monitored universal/existential LSC chart to CTL real-time model checking problems. Similarly, we find out that the semantics of TGA and the existence of the timed game solver UPPAAL-TIGA enable us to reduce the problem of centralized synthesis for the system processes of a set of driving LSC charts into a timed game solving problem.

The first LSC-to-TA translation method was initially implemented in a standalone prototype translator, which can handle more LSC features than what we have described in Paper A, such as control constructs and scoping [Pus10]. A better tailored translator for this method (with chart elements of instance lines, messages, clock constraints, clock resets and prechart) was later implemented and integrated into the UPPAAL GUI and verification server by Sandie Balaguer [Bal09]. The methods and tools have been tried out on a number of examples such as the Train-Gate problem. Results indicate that the approach is viable and effective.

The second LSC-to-TA translation method has been implemented in a prototype translator. The timed automata models generated by the translator can be fed into UPPAAL and UPPAAL-TIGA directly. A number of illustrating examples such as the Vending Machine, the ATM machine and the DHCP protocol have been tried on the prototype implementations. Experimental results show the feasibility and applicability of the proposed methodologies.

In both of the above-mentioned methods, compared with the subsequent verification efforts, the LSC-to-TA translations themselves consume negligible CPU time and memory.

Furthermore, our experience indicates that if we use LSC only as a requirement specification language (Paper A), then it is relatively easy and straightforward to capture the scenario-based requirements; if we use it both as a system modeling and as a requirement specification language (Paper B), then system modeling and property specification will require us to have a deeper understanding of the desired system behaviors and the chart activation modes. It is no surprise that there could be some unintended or unconscious design errors in the initial versions of the user-created driving and monitored LSC charts. Fortunately, many of them can be caught by using our consistency checking and property verification methods on the translated network of TAs.

We have found that our LSC charts could benefit from more expressivity after being enriched with LSC constructs such as subchart, if-then-else structure, loop, forbidden and ignored event, co-region and symbolic instances. However, even with the current simple form of LSC, we can conclude that our scenario-based approaches to real-time system analysis, verification and synthesis can solve problems that are not previously easily specified (Paper A), or can effectively handle TA-like clock variables and clock constraints (Paper B).

In the second part of work (Research questions #3,#3a,#3b), we propose a framework for applying timed games and winning strategies to model-based conformance testing of real-time embedded systems. The framework can be used or tailored to test:

- systems with output uncertainty and/or timing uncertainty of outputs;
- systems where only possibly rather than surely winning game strategies exist for a given property (winning objective); and
- *partially observable* systems, from which the tester can obtain (or can infer) only imperfect information of the system under test.

For the case where a winning strategy can be synthesized for a class of real-time embedded system models which are previously considered *not* testable due to the characteristics of output uncertainty and timing uncertainty of outputs, we propose algorithms for reachability testing and safety validation via interpreting strategies for the relevant timed games. We continue with the case where *cooperative* rather than surely winning game strategies can be synthesized for a class of systems and requirements. Algorithms for test case generation and test execution are developed. Then we generalize the first case to the *partially observable* settings, and show how a strategy for timed game under partial observability can be used to conduct conformance testing. In each of these cases, the soundness and the (partial) completeness of the proposed test methods can be proved. Based on these work we conclude that game-theoretic approaches are conceptually and algorithmically well-suited to address the challenges in the aforementioned settings of real-time embedded system validation.

Experimental evaluations of test generation on some illustrating examples and case studies show the computational feasibility and applicability of the proposed methods. We evaluate how the different factors such as the system sizes and the levels of controllability and observability affect the performance of test generations. The results can help us to pinpoint the performance bottlenecks, to improve the system models and to define a sufficiently but not excessively large set of well-chosen observable predicates.

We also carry out comparative studies of test generation for systems with full observability and with partial observability. The results indicate that the latter method appears to be more cost-efficient and seems to yield smaller test cases.

We can make a corollary jointly from Paper B and Paper C that scenario-based synthesis and game-theoretic testing can be combined to horizontally "scale" the latter approach to inter-process behavioral models (Fig. 4.1, dashed line). The benefit is clear: in an earlier prototyping stage, designers will be able to generate tests (strategies) to make sure whether the interactions among the system processes satisfy some given property. Similarly, combining the work of Paper A and Paper C will horizontally "scale" game-theoretic testing to scenario-based requirements.

Our scenario-based and game-theoretic approaches both operate on the early stage behavioral models of real-time embedded systems, therefore they can achieve early validation. By building automated tool chains and by employing existing automatic timed game solver, we can do automated analysis, verification, synthesis and test generation. The rigorousness of our approaches lies in the underlying real-time model checker and timed game solver, and in our provably correct model transformation and test execution algorithms. Similarly, the debuggability is offered by the underlying model checker and game solver. In this sense, our approaches have many ingredients that characterize an ideal validation technique.

6.2 Future work

Continued efforts on thesis work

For the approaches that are proposed in this thesis to be applicable to more practically relevant systems in a convenient way, continued efforts need to be made both to enhance the methods and to build complete tool chains. Specifically, the LSC language that we define in Paper B needs to be extended with the features of e.g. simregion, co-region, symbolic instance, symbolic message, control constructs and scoping rules. Accordingly, these features need to be supported by the translator. For game-based testing, we need to fully implement the test execution algorithms/procedures for each setting (e.g., reachability testing/safety validation, testing based on surely-/cooperatively- winning strategies, testing based on full/partial observations, and combinations thereof), and to associate them with appropriate coverage metrics both to evaluate test adequacy and to guide the creation of further test purposes (winning objectives).

Beyond reachability and safety test purposes (winning objectives)

The test purposes (or winning objectives) in this thesis are specified as ACTL formulas. More precisely, they are restricted to the forms of: $\mathsf{A}\Diamond\,\varphi$, $\mathsf{A}[\varphi\,\mathsf{U}\,\phi]$, $\mathsf{A}[\varphi\,\mathsf{W}\,\phi]$ (weak "until") and $\mathsf{A}\Box\,\varphi$. A recent feature of UPPAAL-TIGA is the büchi acceptance condition (winning objective) for timed games, e.g., $\mathsf{A}\mathbf{B}\,\varphi$ which stands for $\mathsf{A}\Box\,(\mathsf{A}\Diamond\,\varphi)$, and $\mathsf{A}[\varphi\,\mathbf{B}\,\phi]$ which stands for $\mathsf{A}\Box\,(\varphi\,\wedge\,\mathsf{A}\Diamond\,\phi)$. This adds LTL expressivity to the test purposes, and will thus enable the characterization of some interesting properties such as the infinite alternation between certain system states, and the guaranteed time divergence. Validating systems against such a broader class of properties needs to be explored.

Scenario-based testing

As can be seen in Fig. 4.1, we test an implementation that is based on the timed automata model against an ACTL requirement (Paper C). One can naturally come up with the idea of testing an implementation that is based on the timed automata model against an *LSC requirement*. This can be achieved by first translating the LSC into an observer TA, then generating strategies for the parallelly composed TGA models and the extracted characteristic ACTL formula, and then using the strategies for testing. For existential charts, we can build a timed game with a reachability test purpose of the form $\mathsf{A}\Diamond\,\varphi$. However for universal charts, considering that UPPAAL-TIGA in its current version does not support winning objectives of the form $\varphi \leadsto \phi$, we need either to enhance the game solving algorithm, or to find alternative ways to tackle this problem.

Efficient timed games and efficient game solving

Strategy synthesis is known to be an inherently computationally expensive problem. This is confirmed by our case studies: to synthesize a strategy for even a moderately sized system will take considerable time and memory. What is even worse is that if there is an abortion of the game solving, the abortion due to the shortage of memory usually happens only after some thirty minutes running. This asks us to come up with improved (or reduced) game models or more efficient game solving algorithms. Since algorithms and tool support for checking alternating timed simulation between different timed games have been developed recently [BCDL09], we may wish to construct a cheaper timed game model that simulates the original (more expensive) one.

Achieving optimality in controller synthesis and testing

In this thesis, the timing aspect of game-theoretic approaches to conformance testing has been addressed. A follow-up work could be to use the *time-optimal* strategy [BCD$^+$08] that UPPAAL-TIGA generates for time-optimal conformance testing. Furthermore, for real-time embedded systems, there are a number of other quantitative aspects that deserve investigation, e.g., energy consumption, probabilistic and stochastic behaviors. The cost-optimality problems of priced timed game automata have been studied along the years. Controller synthesis for probabilistic processes and for stochastic games have also been reported. We envisage a common quantitative formal framework that accommodates (several of) these aspects, and a methodology with which we can synthesize strategies for problems in this domain and apply the strategies for conformance testing.

Partial observability as abstraction

The relation between fully observable and partially observable timed games and their test generations are worth further exploration. A partially observable timed game corresponds to a coarser partitioning of the state space of the system than that of its fully observable counterpart. It may be viewed as an "abstraction" of the fully observable timed game. Abstracting too much (i.e., defining too few observable predicates) might lead to game unsolvability, whereas abstracting too little (i.e., defining too many observable predicates) might lead to significantly increased time and memory consumptions. Therefore, systematic methods for defining a cost-efficient set of observable predicates are very desirable.

Paper A:
Verifying Real-Time Systems against Scenario-Based Requirements

Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, Saulius Pusinskas
Center for Embedded Software Systems (CISS)
Department of Computer Science
Aalborg University, Denmark

Abstract

We propose an approach to automatic verification of real-time systems against scenario-based requirements. A real-time system is modeled as a network of timed automata (TA), and a scenario-based requirement is specified as a monitored Live Sequence Chart (LSC). We make timed extensions to a kernel subset of the LSC language, and define a trace-based semantics. By equivalently transforming an LSC chart into an observer TA and then non-intrusively composing this observer automaton with the original system model, the problem of verifying a real-time system against a scenario-based requirement reduces to a CTL real-time model checking problem. We show how this is accomplished in the context of the UPPAAL model checker.

Keywords: Real-Time Systems, Model Checking, Scenarios, Live Sequence Charts (LSCs), Timed Automata (TA)

1 Introduction

A model checker typically needs two inputs: a model that characterizes the state/transition behaviors of a finite state concurrent system, and a temporal logic formula that specifies the property of interest. For real-time systems, a widely used modeling formalism is timed automata (TA) [AD94], and the temporal logics could be CTL, LTL, etc. While the enhanced versions of TA in popular real-time model checkers such as Kronos [Yov97] and Uppaal [BDL04] are usually made relatively expressive, the TCTL [Yov97] or the fragment of CTL [BDL04] logics thereof appear to be property specification languages of only limited capability, intuitiveness and convenience:

- Since their atomic propositions are interpreted over the semantic states of timed automata, these logics do not characterize message communications directly [Yov97, BDL04];
- There is no means for specifying complex timing constraints (e.g., there is no time-bounded temporal operator like $\Diamond_{\leq 3}$ in Uppaal [BDL04]. While Kronos can specify $\mathsf{E}\Diamond_{x\leq 3}$, it cannot specify the stricter ones such as $\mathsf{E}\Diamond_{1\leq x\leq 3}$ [Yov97]).

These limitations imply that straightforward characterizations of event synchronizations, causal relations, or timed scenarios such as "**if** process B sends message m_1 to process A, and C sends m_2 to D (in any order), **then** B **must** send m_3 to C within 1 to 3 time units" as a query in Kronos or Uppaal are not possible.

Essentially, the query languages of these model checkers describe only *intra*-process (or "state/transition-based") properties, i.e., whether all states ($\Box$) or at least one state ($\Diamond$) along all paths (A) or at least one path (E) of the individual processes or the product process (i.e., the parallelly composed system model) satisfy some particular properties. In contrast, the *inter*-process (or "scenario-based") properties describe how the system processes interact, collaborate and cooperate via message or rendezvous synchronizations.

Live Sequence Chart (LSC) [HM03] is a visual formalism for scenario-based specification and programming. It extends the classical Message Sequence Chart (MSC) [IT99] by adding modalities [1]. A universal chart can optionally contain a *prechart*, which specifies the scenario which, if successfully executed (or matched), forces the system to satisfy the scenario given in the actual chart body (i.e., the

[1]The *existential* and *cold* (resp. *universal* and *hot*) modalities represent the provisional (resp. mandatory) requirements at global (i.e., whole chart) and local (i.e., message, condition, location and cut) levels, respectively. For example, an existential (resp. universal) chart specifies restrictions over at least one satisfying (resp. all possible) system runs; a cold condition may be violated and thus lead to a pre-mature chart exit, whereas a hot one must be satisfied and otherwise will indicate an error.

main chart). The LSC language is unambiguous because it has strictly defined semantics, e.g., the executable (operational) semantics [HM03] and the trace-based semantics [DH01, KHP$^+$05].

We envisage LSC as a nice complement to the intra-process property specification languages of (real-time) model checkers in general and of Uppaal in particular:

- *Expressiveness.* It has been shown that a kernel subset of LSC can be embedded into CTL^*, *provided that* event occurrences can be used as atomic propositions [KHP$^+$05]. Compared with many temporal logics whose atomic propositions are restricted to be state formulas, LSC has the necessary language constructs (e.g., message and conditional synchronization) to describe process interactions; these together with its liveness nature enable the characterization of a variety of causality and non-trivial scenarios;

- *Intuitiveness.* As a visual formalism, LSC is more intuitive in capturing complex user requirements than the CTL fragment of Uppaal which is in textual form;

- *Counterexample display.* Compared with conventional counterexamples that are exhibited only on the system models, LSC provides the possibility of tracing the counterexamples also back to the requirement specifications.

In this paper we capture a scenario that is to be verified using an LSC chart. We obtain a behavior-equivalent observer TA from this chart by mapping the LSC cuts and discrete advancement steps to TA locations and edges, respectively. We let the observer TA spy on the relevant events of the original system via model instrumentation, semaphore locking and parallel composition. In this way, the problem of verifying a TA-modeled real-time system against a scenario-based requirement will be reduced to a CTL real-time model checking problem in Uppaal.

1.1 Related work

To model check real-time systems against complex properties or scenario-based requirements, various approaches have been proposed.

One solution is the *observer automata* (a.k.a. *test automata* [ABL98]) approach, i.e., to construct a number of auxiliary automata to capture the complex properties or scenario-based requirements, and then parallelly compose them with the original TA models. This approach has been used to model check practically relevant systems such as the B&O power controller [HLS99] and some timed safety instrumented systems [Lah08]. Case studies thereof indicate that the approach is effective. However, the approach also comes with some limitations: (1) Manual constructions of observer timed automata could be labor-intensive and error-prone. This is especially the case when the automata grow large; (2) To synchronize with the observer timed automata, the original system model may need to be modified and annotated. During this modification process, some new errors might be introduced. Newly introduced timing errors are especially difficult to diagnose; (3) The observer timed automata and the original system usually engage in "loose" channel synchronizations, thus specifying process interactions only *liberally* (i.e., no particular sending and receiving process is specified for a message). In our verification framework, automatic construction of observers from LSC charts overcomes all the above problems.

Another line of research is first to capture the scenario-based requirements using the assume-guarantee style visual formalisms such as Triggered MSC [SC02], Template MSCs [GMMP04], or the even richer LSC [HM03], and then transform them into more verifiable formalisms. In particular LSCs can be translated into timed büchi automata (TBA) [KW01], timed automata [Pus10], temporal logics [KW01, HK02, BGS05, KHP^{+}05, DTW06, BS07] or sequences of LSC elements [RAJGJ04], and the verification problem can be converted to a model checking problem in existing tools [KW01, BGS05], or solved directly [RAJGJ04].

Damm and Klose [DK01] propose to use LSCs to specify scenario-based requirements on STATEMATE models, and then carry out model checking. This methodology has been concretized and implemented in [KW01], where an LSC chart is transformed into a timed büchi automaton, which is further transformed into a temporal logic formula. In order to specify real-time requirements, timers [AHP96, IT99] and timing annotations (or delayed intervals) [AHP96] are added to the LSC charts. To enable the transformation, each location of the LSC chart is equipped with a discrete (integer) clock. Since timers can only express timing constraints within a *single* chart and within a *single* process, and delayed intervals can only express the minimal and maximal delays between two *consecutive* locations, these restrict the expression of timing constraints across processes and across charts. Our LSC charts use TA-like real-valued clock variables. This flavor of timing constraints agree well with the original TA system model, and thus enable smooth translation of timing information into the observer TA, and seamless embedding of the observer TA into the UPPAAL verification framework.

An LSC-to-TA translation is proposed in [Pus10]. This translation is similar to our translation in the sense that they are both based on the notion of LSC cut and its advancements. There are a few differences between the two approaches: (1) The approach of [Pus10] distinguishes between the conditions and updates on message heads and tails, thus a message that is associated with conditions/updates may be mapped to a sequence of TA locations and edges; whereas in our approach, we suggest a more straightforward mapping; (2) The approach of [Pus10] maintains multiple copies of the same translated TA to represent the different incarnations of the chart under the invariant activation mode; whereas we need only one copy of the translated TA, and the different incarnations are accommodated by our approach in terms of the pre-matching mechanism (cf. Section 3.3.5).

LSCs can also be translated into temporal logic formulas [HK02, KHP$^+$05, BGS05, DTW06, BS07]. For the kernel subset of LSC in [KHP$^+$05], it has been shown that existential charts can be expressed using the CTL logic, and universal charts can be expressed using ($\mathsf{LTL} \cap \mathsf{CTL}$) [HK02, KHP$^+$05]. Similar results are achieved by Damm and colleagues [DTW06]. However, these methods do not handle explicit time in the charts. Bunker and colleagues [BGS05] apply LSC in hardware verification, where the system models are given in Verilog and the user requirements are specified as LSCs, which are equipped with a discrete clock tick construct to explicitly represent the passage of system time. The LSCs are translated to LTL formulas and then fed into the verification environment FormalCheck. In general, verification techniques that are based on LSC-to-temporal logic translation tend to suffer from scalability problems. Industrial case studies [Klo03] show that the LTL formulas grow large even for LSCs of moderate size, and thus formal verification becomes expensive. To overcome this limitation, Klose and colleagues [KTWW06] investigate *efficient* model checking of Kripke structures against LSC requirements. The authors identify two sub-classes of LSCs that are easier to verify, although they are less powerful than full LTL model checking. In our method, since our observer automaton is tightly coupled with the original system, a very simple CTL property $\mathsf{A}\Box(l_{min} \Rightarrow \mathsf{A}\Diamond\, l_{max})$ can be extracted from the observer automaton to capture the LSC requirements (here, l_{min} and l_{max} represent the TA locations that correspond to the minimal cut and the maximal cut of the main chart, respectively). In this way we avoid translating LSCs to complex temporal logic formulas.

In [RAJGJ04] properties are extracted from LSCs as sequences of LSC elements, and algorithms have been developed to check whether these sequences are respected by the FSM computation graph of the TA model that is exported from Uppaal. However, simultaneous regions (simregions) in LSCs are used only to model broadcast communications, and conditions cannot be a part of simregions. Our notion of simregion uses the "[condition][message]/[update]" pattern, thus enables smooth translation to a TA edge.

1.2 Contributions

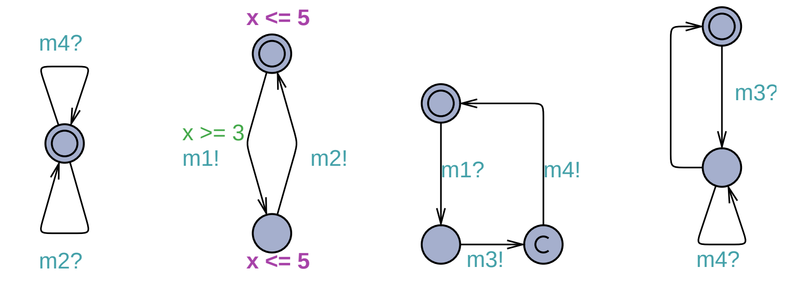
The contributions of this paper include: (1) we make timed extensions to a kernel subset of the LSC language such that it is suitable for capturing scenario-based requirements of real-time systems, and define a trace-based semantics; (2) we propose a behavior-equivalent translation of an LSC chart into an observer timed automaton; (3) we present a method of embedding the translated TA into Uppaal, thus reduce the problem of verifying real-time systems against LSC requirements to a CTL real-time model checking problem.

2 Modeling and specification of real-time systems

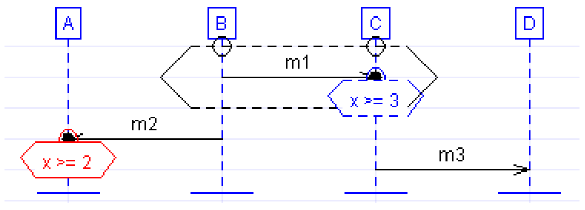
To describe a real-time system which consists of a number of concurrently running processes, a network of timed automata can be constructed, one for each process. These automata are parallelly composed using the operator $||$. Different automata in the system can synchronize on their common actions [AD94]. The product automaton has an interleaved execution semantics w.r.t. the internal actions.

Timed automata in its original form [AD94] is a simple, concise and yet expressive language. To better support the modeling and automatic verification of real-time systems, various syntactic sugar and extensions are added to the TA formalism. Specifically, UPPAAL [BDL04] strengthens TA with a number of features such as boolean and bounded integer variables, variable constraints and updates, urgent and committed locations [2], handshake and broadcast channel synchronizations, shared variable communications, etc.

Fig. 1(a)-1(d) give an example of a network of TAs in UPPAAL.



(a) TA A (b) TA B (c) TA C (d) TA D



(e) LSC requirement $\mathcal{L}$

Figure 1: A real-time system model (network of TAs) and its requirement.

Requirements on TA-modeled real-time systems can be specified by using temporal logics such as CTL, LTL or timed variants thereof. UPPAAL uses a fragment

[2] In an *urgent* location time is frozen and thus cannot elapse. Once an urgent location is entered, it should be exited with zero time delay. A *committed* location is a special urgent location where the outgoing transitions have higher priority to be taken than those from non-committed ones.

of the CTL logic as its property specification language. Atomic propositions take the form:

$$ap ::= automaton.location \mid guard\_on\_clocks \mid guard\_on\_variables,$$

and properties (queries) can be specified by using a number of patterns:

- reachability ($\mathsf{E}\Diamond\phi$);
- safety ($\mathsf{A}\Box\phi$, $\mathsf{E}\Box\phi$); and
- liveness properties ($\mathsf{A}\Diamond\phi$, $\phi \leadsto \varphi$).

In particular the *leads-to* (*responsiveness*) property $\phi \leadsto \varphi$ is a shorthand for $\mathsf{A}\Box(\phi \Rightarrow \mathsf{A}\Diamond\varphi)$, stating that whenever ϕ is satisfied, then eventually φ will be satisfied.

Although a lot of properties can be specified by using the above-mentioned property patterns, many others still cannot. Consider a user requirement on the TAs in Fig. 1(a)-1(d):

If *we observe that process B sends message m_1 to process C when clock x is no less than 3,* ***then*** *afterwards (and before m_1 can be observed again) we* ***must*** *observe that B sends m_2 to A when x is no less than 2, and C sends m_3 to D (in any order).*

This requirement cannot be specified as a UPPAAL CTL formula or a KRONOS TCTL formula. The reason is that the atomic propositions, which are restricted to be state propositions, do not characterize message passing directly. In other words, they lack the necessary mechanisms for specifying the process interactions and scenarios.

However, the above requirement can be easily captured by using LSC (Fig. 1(e)). For instance, the first block of diagrammatic elements $\{m_1, x \geq 3\}$ means that: when message m_1 in the real-time system model is observed, the value of clock x should be no less than 3 at this moment; and if this is the case, then the monitored execution continues, otherwise the prechart is cold-violated and exited.

3 From LSC to Uppaal timed automaton

3.1 Live Sequence Chart

We consider the following LSC elements: instance line, location, message, clock variable, condition (clock constraints), assignment (clock resets) and simregion.

An LSC chart can have a role, a type and an activation mode. In this paper we consider the role of *system property specification*, i.e., a monitored chart will just “listen to” the messages and read the clock variables in the original system models, but will never emit messages to those models or reset the clocks in those models. We consider the *universal* type charts. Furthermore, we consider the *invariant* activation mode, i.e., the prechart is being constantly monitored, and an incarnation will be created whenever a minimal event (i.e., an event that is minimal in the partial order induced by the chart) is matched, regardless of whether the main chart has been entered in some earlier incarnations.

Each LSC chart $\mathcal{L}$ describes a particular interaction scenario of a set I of processes (or instances, or agents). Along each instance line $I_i \in I$ there are a finite set of "positions" $pos(I_i) = \{0, 1, \ldots, p\_max_{I_i}\}$, which denote the possible points of communication and computation. We denote all *locations* of $\mathcal{L}$ as $L = \{\langle I_i, p\rangle \mid I_i \in I \ \wedge \ p \in pos(I_i)\}$.

Let ML be the set of **m**essage **l**abels (or "signals", or "channels" in Uppaal) of an LSC chart. A *message occurrence* $mo = (\langle I_i, p\rangle,\ m,\ \langle I_{i'}, p'\rangle) \in L \times ML \times L$ corresponds to instance I_i, while in its position $(p-1)$, sending signal $m \in ML$ to instance $I_{i'}$ at its position $(p'-1)$, and then arriving at positions p and p', respectively. We call m the message label, $\langle I_{i'}, p'\rangle$ and $\langle I_i, p\rangle$ the message head and tail locations, and I_i and $I_{i'}$ the source and destination instances, respectively. The set of all message occurrences in the chart are denoted as MO.

We use Σ to denote the projection of MO onto $I \times ML \times I$. In this way, we get the **m**essage **a**lphabet Σ, where each letter is a *message* which denotes that a particular signal is sent from one to another objects (instance lines). For a given message occurrence, we may overload its "message label" to also denote the corresponding letter in Σ.

In this paper, we make the *synchrony* hypothesis, i.e., messages are assumed to be instantaneous (thus we use the terms *message* and *event* interchangeably). Furthermore, this paper does not consider concurrent messages, thus each location can be the end point of at most one message occurrence in the chart.

Let the finite sets of real-valued *clock variables* (ranging over $\mathbb{R}_{\geq 0}$) of chart $\mathcal{L}$ and of the original system model $\mathcal{S}$ be $C_{\mathcal{L}}$ and $C_{\mathcal{S}}$, respectively. The set of readable clock variables in $\mathcal{L}$ will be $C = C_{\mathcal{L}} \cup C_{\mathcal{S}}$. Since $\mathcal{L}$ is a monitored chart, only clocks in $C_{\mathcal{L}}$ can be reset by the chart.

A *clock constraint* is of the form $x \bowtie n$ or $x - y \bowtie n$ where $x, y \in C$, $n \in \mathbb{Z}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. A *condition* is a finite conjunction of clock constraints. The set of conditions are denoted G. A condition $g \in G$ has a temperature, denoted $g.temp$, which may be either hot or cold in the main chart, and only cold in the prechart.

A *clock reset* is of the form $x := 0$ where $x \in C_{\mathcal{L}}$. An *assignment* a is a finite set of clock resets. For simplicity it is denoted as a set a of clocks to be reset. The set of all assignments is denoted $A \subseteq 2^{C_{\mathcal{L}}}$.

When there is a message occurrence $mo = (\langle I_i, p\rangle,\ m,\ \langle I_{i'}, p'\rangle)$, the message anchoring point on I_i or $I_{i'}$ could be associated with a condition g and/or an assignment a. The intuitive meaning of the message synchronization $[g]\, mo\, /a$

is that, if when mo occurs, the valuation v of all clock variables satisfies g, then this synchronization can fire; and immediately after the firing, v will be updated according to a. The message occurrence, condition and assignment can be collectively viewed as an atomic step of LSC execution, i.e., they take place at the same moment in time, hence the notion of *simultaneous region* (simregion), which is inspired by [KW01].

Definition 1 (simregion)**.** *A* simregion s *is a set of LSC message occurrence, condition and assignment,* $s \subseteq (MO \cup G \cup A)$*, which satisfy the following requirements:*

- *non-emptiness:* $\exists e \in (MO \cup G \cup A) \,.\, e \in s$;
- *uniqueness:* $\forall m, n \in MO \,.\, (m \in s \ \wedge \ n \in s) \Rightarrow m = n$; *(similarly for condition and assignment.); and*
- *non-overlapping: for any two simregions* s *and* s'*, we have* $\forall e \in (MO \cup G \cup A) \,.\, (e \in s \ \wedge \ e \in s') \Rightarrow s = s'$. □

We write a simregion as $s = \{mo, g, a\}$, where mo, g and a represent the message occurrence, condition and assignment, respectively. The set of all simregions is denoted $S \subseteq 2^{(MO \cup G \cup A)}$.

A message occurrence spans across two instance lines. A condition spans across one or more instance lines. In a simregion, the message occurrence, condition and assignment (if any) have a common anchoring point. If a simregion s has no message, then s consists of a condition test, or an assignment, or both of them combined and anchored together. In this case, s is called a *non-message* simregion. For such a simregion, we adopt the As-Soon-As-Possible (ASAP) semantics for its firing, i.e., the condition test (if any) will be evaluated immediately after the previous simregion.

Fig. 1(e) is an example LSC chart, where there are three simregions $s_1 = \{m_1, x \geq 3\}$, $s_2 = \{m_2, x \geq 2\}$, and $s_3 = \{m_3\}$. Note that here we abuse the message labels as the corresponding message occurrences.

3.2 Trace-based semantics

We define $\lambda : L \rightarrow S \cup \{nil\}$ as a labeling function. For location $l \in L$, if $\lambda(l) \in S$, then there is a simregion anchoring at l; if $\lambda(l) = nil$, then l represents an entry/exit point of the prechart(Pch)/main chart(Mch).

Locations in an LSC chart are partially ordered by the following rules:

- Along each instance line: location l is above $l' \Rightarrow (l \leq l') \wedge \neg(l' \leq l)$; and
- All locations in the same simregion have the same order, $\forall s \in S, \forall l, l' \in L \,.\, (\lambda(l) = s) \wedge (\lambda(l') = s) \Rightarrow (l \leq l') \wedge (l' \leq l)$. □

The partial order relation $\preccurlyeq \subseteq L \times L$ is defined as a transitive closure of $\leq$.

Definition 2 (cut of an LSC chart)**.** *A* cut *of a chart $\mathcal{L}$ is a set $c \subseteq L$ of locations that span across all the instance lines in $\mathcal{L}$ which satisfies the properties of:*

- Downward-closure*. If a location l is included in cut c, so are all of its predecessor locations: $\forall l,\, l' \in L.\, (l \in c \wedge l' \preccurlyeq l) \Rightarrow l' \in c$; and*

- Intra-chart coordination integrity*. If a top position in the main chart portion of a certain instance line is included in the cut, then the top positions in the main chart portions of all other instance lines are also included in the cut.* □

We define $loc : (S \cup 2^L) \rightarrow 2^L$ to map a simregion $s \in S$ to a set $loc(s) \in 2^L$ of locations that it anchors, and to map a cut $c \in 2^L$ to its *frontier* $loc(c) \in 2^L$, which is a set of locations that constitute the downward border line progressed so far.

Given a cut $c \subseteq L$ and a simregion $s \in S$, we say s is *enabled* at cut c (with respect to the partial order relation), denoted $c \xrightarrow{s}$, if, $\forall\, l \in c,\ l' \in loc(s)\,.\,((l \preccurlyeq l') \wedge \neg(l' \preccurlyeq l)) \wedge (\nexists\, l'' \in L \backslash (c \cup loc(s))\,.\,(l \preccurlyeq l'' \wedge l'' \preccurlyeq l'))$. The enabledness of message occurrences can be defined similarly.

A cut c' is an *s-successor* of c, denoted $c \xrightarrow{s} c'$, if s is enabled at c (w.r.t. the partial order), and c' is achieved by adding the set of locations that s anchors into c, or formally, $(c \xrightarrow{s}) \wedge (c' = c \cup loc(s))$.

A cut c is *minimal* (denoted $\top$) if each location in c is a top location of some instance line; and c is *maximal* (denoted $\perp$) if the bottom locations of all instance lines are included in c. The frontiers of cuts $\top$ and $\perp$ do not contain simregion anchoring points. Rather, each of these minimal or maximal cuts represents a compulsory synchronization for all involved instance lines. Thus the partial order relation $\preccurlyeq$ on L is extended as follows (and finally also extended to its transitive closure):

- All locations in the same minimal or maximal cut have the same order, $\forall c \in \{Pch.\top, Pch.\perp, Mch.\top, Mch.\perp\}\,.\,\forall l, l' \in loc(c)\,.\,(l \preccurlyeq l') \wedge (l' \preccurlyeq l)$. □

Specifically, we view the maximal cut of the prechart and the minimal cut of the main chart as the same cut, i.e., $Pch.\perp = Mch.\top$.

If cut c has $c' = Mch.\perp$ as its s-successor, then we override c' as $Pch.\top$ (if any) or $Mch.\top$ (otherwise), which represents the situation where a universal chart goes back to its initial state upon the successful completion of a round of monitoring.

For instance in Fig. 1(e), the possible cuts are: $\{\}$, $\{s_1\}$, $\{s_1, s_2\}$, $\{s_1, s_3\}$ and $\{s_1, s_2, s_3\}$, where e.g. $\{s_1\}$ is a shorthand for the cut where simregion s_1 has been stepped over. Clearly, cuts $\{s_1, s_2\}$ and $\{s_1, s_3\}$ are the s_2-successor and s_3-successor of cut $\{s_1\}$, respectively.

Definition 3 (configuration)**.** *A* configuration *of an LSC chart $\mathcal{L}$ is a tuple (c, v), where c is a cut, and v maps each clock variable to a non-negative real number, $v : C_{\mathcal{L}} \to \mathbb{R}_{\geq 0}$.* □

For $d \in \mathbb{R}_{\geq 0}$, notation $(v+d) : C_{\mathcal{L}} \to \mathbb{R}_{\geq 0}$ means that the function v is shifted by d such that $\forall x \in C_{\mathcal{L}} \,.\, v(x + d) = v(x) + d$.

A configuration at the minimal cut $\top$ with all clocks assigned their initial values (e.g., 0's) is called the *initial* configuration.

An assignment $a \in A$ can be viewed as a transformer for function v, thus $a(v)$ represents the new valuation after the assignment.

A configuration can be viewed as a "semantic state" of an LSC chart. A universal chart starts from the initial configuration, advances from one to a next configuration, until hot violation occurs, or until the chart arrives at the maximal cut and then starts all over again (i.e., to begin a next round execution).

There are three kinds of valid advancement steps between two configurations:

- *Synchronization step.* Given a chart configuration (c, v) and a simregion s which consists of an m-labeled message occurrence mo ($m \in \Sigma$), and optionally a condition g and/or an assignment a. There is a synchronization step $(c, v) \xrightarrow{m} (c', v')$ if,
  - (normal advancement). $c \xrightarrow{s} c'$, $v \models g$, and $v' = a(v)$; or
  - (cold violation). $c' = Pch.\top$, $v' = v$, and either
    - mo is not enabled at cut c in the prechart (w.r.t. the partial order relation); or
    - $v \not\models g \wedge g.temp = cold$;
- *Silent step.* Given a chart configuration (c, v) and a simregion s which consists of a condition g and/or an assignment a, there is a silent step $(c, v) \xrightarrow{\tau} (c', v')$ if either
  - (*normal advancement*). $c \xrightarrow{s} c'$, $v \models g$, and $v' = a(v)$; or
  - (*cold violation*). $g.temp =$ cold, $v \not\models g$, $c' = Pch.\top$, and $v' = v$;
- *Time delay step.* Given a chart configuration (c, v). There is a time delay step $(c, v) \xrightarrow{d} (c, v + d)$ where $d \in \mathbb{R}_{\geq 0}$ if: whenever there are message occurrences that are enabled at cut c (w.r.t. both the partial order relation and the guard), then after delay d there exists at least one of them that is still enabled at the same cut, i.e., $\exists s \in S \,.\, \exists\, mo, g \in s \,.\, \forall d' \in [0, d] \,.\, (c \xrightarrow{s}\ ) \wedge ((v + d') \models g)$. □

If in the main chart, an m-labeled message violates $\preccurlyeq$, or ($v \not\models g$, and $g.temp =$ hot), then the configuration (c, v) is said to be *hot-violated*, denoted $(c, v) \not\xrightarrow{m}$.

Definition 4 (run of an LSC chart)**.** *A* run *of a time-enriched LSC chart is a sequence of configurations* $(c_0, v_0)\cdot(c_1, v_1)\cdot\ldots$ *that are connected by the advancement steps, i.e.,* $\forall i \geq 0 \,.\, \exists u_i \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \,.\, (c_i, v_i) \xrightarrow{u_i} (c_{i+1}, v_{i+1})$. □

The transition relation $\rightarrow$ as mentioned above each time consumes only a single letter $u \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. We extend it to $\rightarrow^*$ such that it consumes a (finite or infinite) word $w \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$.

Let Π be the set of all messages in the original system model, which subsumes Σ and can in addition include other messages not ever appeared in Σ.

Definition 5 (satisfaction of a prechart/main chart)**.** *A timed trace* $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies *an LSC prechart or main chart* $\mathcal{C}$ *if its projection* $\gamma|_{(\Sigma\cup\{\tau\}\cup\mathbb{R}_{\geq 0})}$ *has a finite prefix* μ *which is the accepted word of a run that starts from the initial configuration and ends in a maximal cut configuration of* $\mathcal{C}$*, and no prefix of it leads to a hot violation, i.e.,* $\gamma \models \mathcal{C} \Leftrightarrow (\exists \mu \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, \xi \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, \exists v' \in {\mathbb{R}_{\geq 0}}^C \,.\, (\gamma|_{(\Sigma\cup\{\tau\}\cup\mathbb{R}_{\geq 0})} = \mu \cdot \xi) \wedge ((\top, v_0) \xrightarrow{\mu}{}^* (\bot, v'))) \;\wedge\; (\nexists \mu' \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, m \in \Sigma, \xi \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, (\gamma|_{(\Sigma\cup\{\tau\}\cup\mathbb{R}_{\geq 0})} = \mu' \cdot m \cdot \xi) \wedge (\top, v^0) \xrightarrow{\mu'}{}^* \bullet \overset{m}{\not\rightarrow})$. □

If a universal chart $\mathcal{L}$ has no prechart *Pch*, then it is treated as being satisfied by an empty word.

A finite trace $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*$ satisfies chart $\mathcal{C}$ *exactly*, denoted $\gamma \Vdash \mathcal{C}$, iff $(\gamma \models \mathcal{C}) \wedge \exists \mu \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*,\; v' \in {\mathbb{R}_{\geq 0}}^C \,.\, (\gamma|_{(\Sigma\cup\{\tau\}\cup\mathbb{R}_{\geq 0})} = \mu) \wedge ((\top, v^0) \xrightarrow{\mu}{}^* (\bot, v'))$.

Now we define the satisfaction relation for a full universal chart:

Definition 6 (satisfaction of a universal LSC chart)**.** *A timed trace* $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies *a universal chart* $\mathcal{L}$*, denoted* $\gamma \models \mathcal{L}$*, iff whenever a finite sub-trace of* γ *matches the prechart, then the main chart is matched immediately afterwards. Formally,* $\forall \alpha, \mu \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, \beta \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, (\alpha \cdot \mu \cdot \beta = \gamma) \wedge (\mu \Vdash Pch) \Rightarrow \beta \models Mch$. □

A timed language $Lang \subseteq (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies $\mathcal{L}$, denoted $Lang \models \mathcal{L}$, iff, $\forall \gamma \in Lang \,.\, \gamma \models \mathcal{L}$. Clearly, *Lang* characterizes the system behaviors that respect $\mathcal{L}$.

For a network $\mathcal{S}$ of timed automata, we use $\mathcal{S} \models \mathcal{L}$ to denote that the timed traces (language) of $\mathcal{S}$ satisfy LSC $\mathcal{L}$.

3.3 LSC to TA translation

For each LSC chart $\mathcal{L}$, we construct a UPPAAL observer timed automaton $O_{\mathcal{L}}$. The basic idea is that for each cut of the LSC, we assign a TA location in UPPAAL; for each discrete advancement step (i.e., a simregion) that connects two consecutive cuts, we assign a TA edge. The translation is conducted incrementally based on the partial order relation $\preccurlyeq$.

3.3.1 Determining the partial order on LSC simregions

By analyzing the graphical layout of the LSC chart, the partial order $\preccurlyeq$ on the set L of locations is determined according to the rules given in Section 3.2.

Since an advancement of a cut is caused by stepping over a simregion, the partial order $\preccurlyeq$ on L can thus be lifted to $\preccurlyeq'$ on $S \cup \{Pch.\top, Mch.\top, Mch.\bot\}$ as follows: $\forall s_1, s_2 \in (S \cup \{Pch.\top, Mch.\top, Mch.\bot\}) \,.\, (s_1 \preccurlyeq' s_2 \Leftrightarrow \exists l_1 \in loc(s_1), l_2 \in loc(s_2) \,.\, l_1 \preccurlyeq l_2)$.

For instance in Fig. 1(e), the partial order $\preccurlyeq'$ among the three simregions s_1 (middle), s_2 (left) and s_3 (right) is: $s_1 \preccurlyeq' s_2$ and $s_1 \preccurlyeq' s_3$.

3.3.2 Translating LSC cut into TA location

The initial cut of an LSC chart is the minimal cut $\top$ of the prechart (if any) or of the main chart (otherwise). While respecting $\preccurlyeq'$, the cut advances towards $Mch.\bot$ by stepping over simregions. Each time a simregion is stepped over, a new cut is reached.

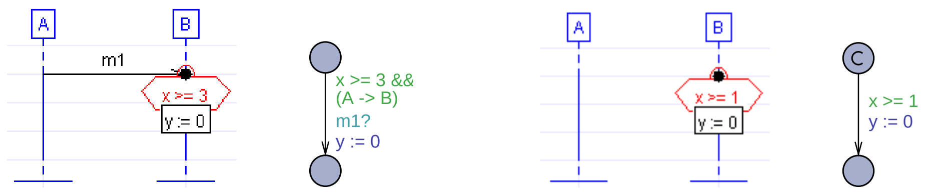
If we view all the instances of an LSC chart collectively as a whole system, then a cut can be viewed as a "location" of the TA of this whole system. For the minimal cut of the prechart (if any) and the minimal and maximal cuts of the main chart, we assign the TA locations l_{pmin}, l_{min} and l_{max}, respectively. Note that l_{max} is a committed location, which will be connected to l_{pmin} (if any) or l_{min} via an edge of internal action transition, meaning that a next round of monitoring will begin immediately. The l_{pmin}, l_{min} and l_{max} locations are three mandatory synchronization points for all the instances in the chart.

Time can elapse while staying in an LSC cut just like in a TA location. Specifically, a cut that is followed by a non-message simregion corresponds to a committed TA location. In that cut time is frozen and cannot elapse.

Since there are only finitely many instances and finitely many simregions in an LSC chart, the number of cuts will also be finitely many.

3.3.3 Translating LSC simregion into TA edge

If s is a message-simregion, then we map the message, condition (if any) and assignment (if any) of s into one edge of the TA. See Fig. 2(a)-2(b).



(a) A msg.-simregion (b) TA edge (c) A non-msg. simregion (d) TA edge

Figure 2: From LSC simregion to TA edge.

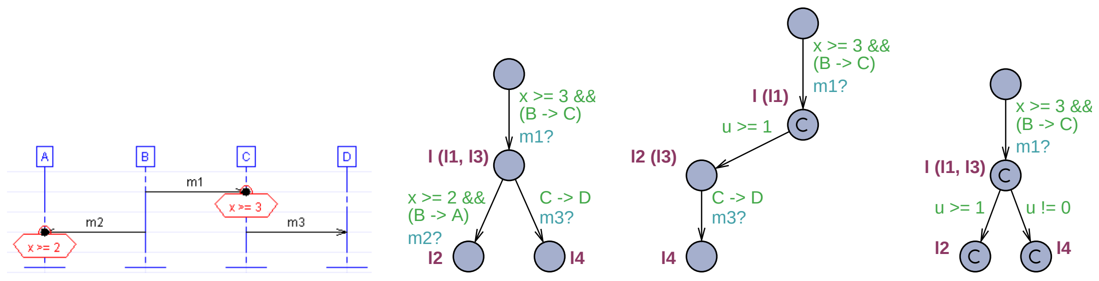
In an LSC chart, a message m in Σ is sent from one particular instance line to another one (e.g., from A to B). To preserve this sender/receiver information in the translated timed automaton, the TA edge will be further guarded by the predicate $(A \rightarrow B)$, which is a shorthand for "$(src = A)\,\&\&\,(dest = B)$". See Fig. 2(b).

If s is a non-message simregion, then the ASAP semantics is adopted. To enforce the ASAP semantics, the source location of the translated TA edge will be marked as a committed location. See Fig. 2(c)-2(d) for an example.

3.3.4 Incremental construction of the TA

The LSC to TA translation is carried out incrementally. Assume that a TA location l has already been created for the current LSC cut (see Fig. 3(b), location l, and Fig. 3(a), cut $\{s_1 = \{m_1, x \geq 3\}\}$). Following that cut there could be a number of simregions that can be stepped over. Each of them should correspond to an outgoing edge from TA location l. Without loss of generality, we assume that there are two such immediately following simregions, say $s_2 = \{m_2, x \geq 2\}$ and $s_3 = \{m_3\}$ in Fig. 3(a).

If s_2 and s_3 are both message-simregions (Fig. 3(a)), then the two new TA edges will be concatenated to location l. Let the two new edges be (l_1, l_2) and (l_3, l_4), respectively. Then l_1 and l_3 will be superposed on l. See Fig. 3(b).



(a) The simregions (b) case #1 (c) case #2 (d) case #3

Figure 3: TA edge construction for two subsequent simregions.

If in Fig. 3(a) s_2 is replaced by a non-message simregion, then according to the ASAP semantics, the edge (l_1, l_2) will be executed immediately, and edge (l_3, l_4) will follow, but cannot be the other way around. When concatenating these two edges to the TA, we mark l_1 as a committed location, and superpose it on l. There is only one possible interleaving where edge (l_3, l_4) follows (l_1, l_2). See Fig. 3(c).

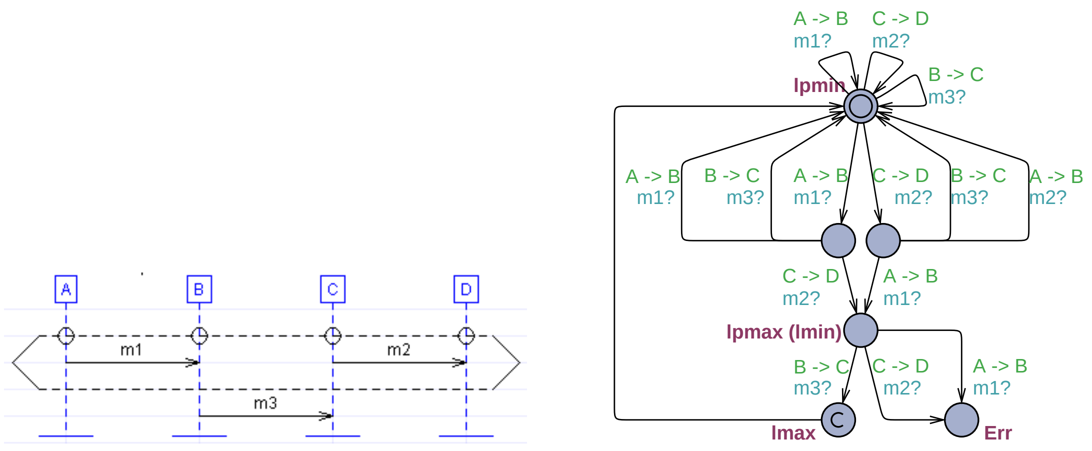
If in Fig. 3(a) s_2 and s_3 are both non-message simregions, then according to the ASAP semantics, both (l_1, l_2) and (l_3, l_4) will be executed immediately, therefore the executions will be interleaved. See Fig. 3(d).

3.3.5 Implicitly allowed behavior

In addition to the *explicitly* specified behaviors in the chart, there are also *implicitly* allowed behaviors that are due to unconstrained events and cold violations.

Let $Chan$ be the set of channels of $\mathcal{S}$, and $Chan' \subseteq Chan$ be the set of channels of $\mathcal{L}$. Clearly, channels in $(Chan \backslash Chan')$ are not constrained by chart $\mathcal{L}$. For each message m whose label belongs to $(Chan \backslash Chan')$, we add an m?-labeled self-loop edge to each non-committed location l of the translated TA $O_{\mathcal{L}}$. For readability they are not shown in Fig. 4.

According to the LSC semantics, cold violations of prechart or main chart are not failures. Rather, they just bring the chart back to the minimal cut. To model this, for a cut c and each following simregion s that has a cold condition g, we add edges from the corresponding TA location l to l_{pmin} (if Pch exists) or l_{min} (otherwise) to correspond to the $\neg g$ conditions (of DNF form). Similarly, given a cut c in the prechart, for each message m that occurs in $\mathcal{L}$ but does not follow c immediately, we also add an m?-labeled edge (l, l_{pmin}). See Fig. 4.



(a) An LSC chart (b) The translated TA

Figure 4: Translating LSC chart: implicitly allowed behaviors and invariant activation mode.

3.3.6 Chart activation mode

According to the LSC semantics, under the invariant activation mode the prechart is being continuously monitored. Normally we need to maintain multiple incarnations of the chart. In this way, a given message sequence will not be incorrectly rejected by the chart. For instance, given the chart in Fig. 4(a) and given a message sequence $m_1 \cdot m_1 \cdot m_2$, although the first incarnation of the chart cold-violates this sequence (i.e., $m_1 \cdot m_1$ does not match the prechart), the second incarnation works well with it (i.e., the latter two messages $m_1 \cdot m_2$ match the prechart).

To enforce the LSC semantics under the invariant activation mode, for each message occurrence that appears in the chart no later than a certain message occurrence that has the same label as a minimal event (i.e., an event that is minimal in the partial order induced by the chart), we add a corresponding self loop to location l_{pmin}. See Fig. 4(b), the m_1- and m_2-labeled self loops in location l_{pmin}. We call this kind of handling as prechart pre-matching.

3.3.7 Undesired behavior

The construction of the observer TA so far considers only the legal (or admissible) behaviors. When the current configuration (c, v) is in the main chart, if an observed message m is not enabled at cut c, or the hot condition of the simregion that immediately follows c evaluates to `false` under v, then there will be a hot violation. In this case, we add a dead-end (sink) location Err in the TA, and for each such violation we add an edge to Err.

3.3.8 Complexity

Let the number of simregions appearing in $\mathcal{L}$ be n. In the worst case, the number of locations in the translated TA $O_{\mathcal{L}}$ is $2^n + 1$. This happens when $\mathcal{L}$ consists of only the prechart or the main chart, and the messages in $\mathcal{L}$ are totally unordered.

The number of outgoing edges from a location l of $O_{\mathcal{L}}$ depends on: (1) the number of unconstrained events, ue; (2) the number of the following simregions in the corresponding cut c of $\mathcal{L}$, fs; (3) the length of the condition (in case the condition evaluates to `false`), lc; and (4) the number of messages that cause violations of the chart, cv. Therefore, the number of outgoing edges from a TA location is at the level $O(ue + fs + lc + cv)$.

3.4 Equivalence of LSC and TA

Since all the clocks in the original system model $\mathcal{S}$ are also visible to the LSC chart $\mathcal{L}$, we extend the configuration of $\mathcal{L}$ from $C_{\mathcal{L}}$ to $C_{\mathcal{L}} \cup C_{\mathcal{S}}$.

If in the translated timed automaton $O_{\mathcal{L}}$ of chart $\mathcal{L}$ we ignore the undesired and implicitly allowed behaviors, i.e., we ignore the edges that correspond to hot violations, unconstrained events, cold violations and prechart pre-matching, then we have:

Lemma 1. *If a configuration (c, v) of $\mathcal{L}$ corresponds to a semantic state (l, v) of $O_{\mathcal{L}}$, then: (1) each simregion s that follows (c, v) in $\mathcal{L}$ uniquely corresponds to an outgoing edge (l, l') in $O_{\mathcal{L}}$, and (2) the target configuration (c', v') of s in $\mathcal{L}$ uniquely corresponds to the target semantic state (l', v') in $O_{\mathcal{L}}$.*

Theorem 1. *For any trace tr in $O_{\mathcal{L}}$: $tr \models \mathcal{L} \Leftrightarrow (O_{\mathcal{L}}, tr) \models (l_{min} \leadsto l_{max})$.*

Proofs of the lemmas and theorems can be found in Appendix.

As we can anticipate, the prechart pre-matching mechanism does introduce undesired extra behaviors and non-determinacy. For instance in Fig. 4(b), $tr = m_1 \cdot m_2 \cdot m_1 \cdot m_2 \cdot m_3$ could be an accepted trace in $O_{\mathcal{L}}$. But since its subsequence $tr' = m_1 \cdot m_2 \cdot m_1$ can be rejected, thus tr does not really satisfy $\mathcal{L}$. It coincides that the particular trace tr in the model $O_{\mathcal{L}}$ does not satisfy the property $(l_{min} \leadsto l_{max})$.

Theorem 1 indicates that $O_{\mathcal{L}}$ has exactly the same set of *legal* traces as $\mathcal{L}$.

4 Embedding into Uppaal

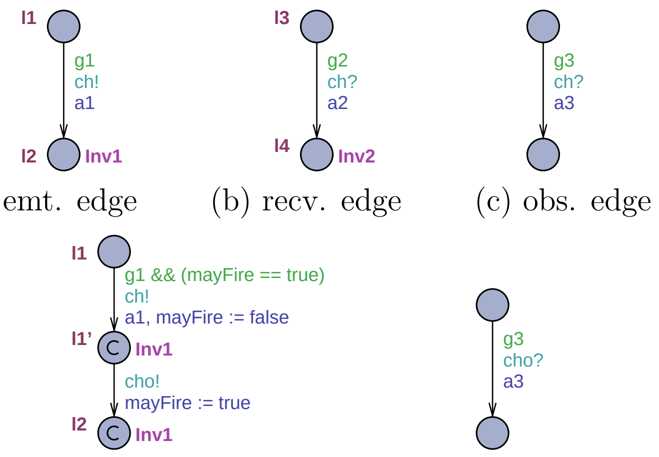
4.1 Synchronizing with the original system

When composing the observer timed automaton $O_{\mathcal{L}}$ with the original system $\mathcal{S}$, we wish that $O_{\mathcal{L}}$ would "observe" $\mathcal{S}$ in a *timely* and *non-intrusive* manner. A natural idea is to let the synchronization channels in $O_{\mathcal{L}}$ (and accordingly the relevant channels in $\mathcal{S}$) be broadcast channels to achieve this goal. However, this is not possible because UPPAAL has a restriction that broadcast channels cannot be guarded by timing constraints. To solve this problem, we propose to use spying techniques such that the translated observer TA will be notified of each message synchronization in the original system *immediately after* it occurs there. Specifically, for each channel $ch \in Chan$, we make the following modifications:

(1) In $\mathcal{S}$ (e.g., Fig. 5(a)-5(b)), for each edge (l_1, l_2) that is labeled with ch!, we add a committed location l_1' and a cho!-labeled edge in between edge (l_1, l_2) and location l_2. Here cho is a dedicated fresh channel which aims to notify $O_{\mathcal{L}}$ of the occurrence of the ch-synchronization in $\mathcal{S}$. The location invariant (if any) of l_2 will be copied on to l_1'. Furthermore, we use a global boolean flag variable (i.e., a binary semaphore) *mayFire* to further guard the ch-synchronization. This semaphore is initialized to `true` at system start. It is cleared immediately after the ch-synchronization in $\mathcal{S}$ is taken, and it is set again immediately after the cho-synchronization is taken. See Fig. 5(d).

(2) In $O_{\mathcal{L}}$, each synchronization label ch? is renamed to cho?. See Fig. 5(c), 5(e).

If $\mathcal{L}$ has non-message simregions, then $O_{\mathcal{L}}$ has committed locations. If in a certain state both $O_{\mathcal{L}}$ and some TA in $\mathcal{S}$ are in committed locations (e.g., l_{m+1} in Fig. 6(c), and l_2 in Fig. 6(a)), then there will be a racing condition. But according to the ASAP semantics of $\mathcal{L}$, the (internal action) edge in $O_{\mathcal{L}}$ has higher priority. To this end, for each edge (l_i, l_{i+1}) in $O_{\mathcal{L}}$, if l_{i+1} is a committed location, then we add "$NxtCmt :=$ `true`" to the assignment of the edge, otherwise we add "$NxtCmt :=$ `false`". Here the global boolean flag variable (i.e., a binary



(a) emt. edge (b) recv. edge (c) obs. edge

(d) mod. emt. edge (e) mod. obs. edge

Figure 5: Edge modifications in the original system $\mathcal{S}$ and the observer TA $O_{\mathcal{L}}$.

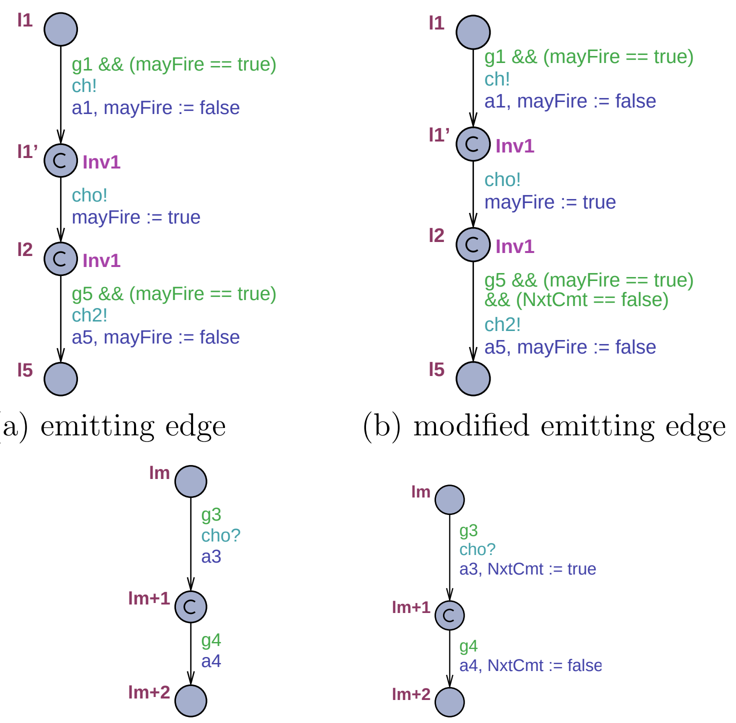
semaphore) *NxtCmt* denotes whether the observer TA will be in a committed location. This semaphore is initialized to `false` at system start. See Fig. 6(d). Accordingly, for each `ch`-labeled edge (l_i, l_{i+1}) in $\mathcal{S}$ where $ch \in Chan$ and l_i is a committed location, we add "$NxtCmt ==$ `false`" to the condition of the edge, see Fig. 6(b).

Our method of composing the observer timed automaton $O_{\mathcal{L}}$ with the original system model $\mathcal{S}$ is similar to that of [FHD+99]. While their method works only when the target state of a communication action is not a committed location in the original model, in our method, due to the first locking mechanism (using *mayFire*), we have no restrictions on whether a location in $\mathcal{S}$ is a normal, urgent or committed one. Broadcast channels can be handled the same way as binary synchronization channels in our method. Furthermore, due to the second locking mechanism (using *NxtCmt*), we guarantee the enforcement of the ASAP semantics in $O_{\mathcal{L}}$.

Since our method involves only syntactic scanning and manipulations, the method is not expensive. For each $ch \in Chan$, we need to introduce a dedicated fresh channel `cho`. For each occurrence of the emitting edge `ch!`, we need to introduce a fresh committed location in $\mathcal{S}$. Moreover, we need two global boolean flag variables (*mayFire*, *NxtCmt*) as the binary semaphores.

4.2 Verification problem

After the modifications, the original system model $\mathcal{S}$ becomes $\mathcal{S}'$, and the observer timed automaton $O_{\mathcal{L}}$ for chart $\mathcal{L}$ becomes $O'_{\mathcal{L}}$. Let the minimal and maximal cuts of the main chart of $\mathcal{L}$ correspond to locations l_{min} and l_{max} of $O'_{\mathcal{L}}$, respectively. Recall that the UPPAAL "leads-to" property $(\phi \leadsto \varphi)$ stands for $\mathsf{A}\Box(\phi \Rightarrow \mathsf{A}\Diamond\, \varphi)$, where ϕ, φ are state formulas.



(a) emitting edge (b) modified emitting edge

(c) in obs. TA (d) in modified obs. TA

Figure 6: Edge modifications when there are committed locations in the obs. TA.

Lemma 2. *If $O_{\mathcal{L}}$ has no committed location, and all $ch \in Chan$ are binary synchronization channels, then $\mathcal{S} \models \mathcal{L} \Leftrightarrow (\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \leadsto l_{max})$.*

In a more general form, we have:

Theorem 2. $\mathcal{S} \models \mathcal{L} \Leftrightarrow (\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \leadsto l_{max})$.

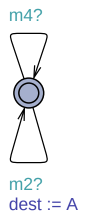
Theorem 2 indicates that the problem of checking whether a system model $\mathcal{S}$ satisfies an LSC requirement $\mathcal{L}$ can be equivalently transformed into a CTL real-time model checking problem ("ϕ leads-to φ") in UPPAAL.

5 An example

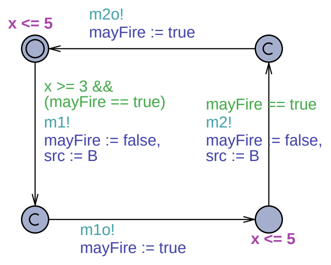
We put things together by using the example in Fig. 1. The original system $\mathcal{S}$ consists of timed automata A, B, C and D, having channels m_1, m_2, m_3 and m_4, and clock variable x. The scenario-based requirement $\mathcal{L}$ is given in Fig. 1(e).

After modifying $\mathcal{S}$ and the translated observer timed automaton $O_{\mathcal{L}}$, we get the newly composed network of TAs $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$, see Fig. 7.

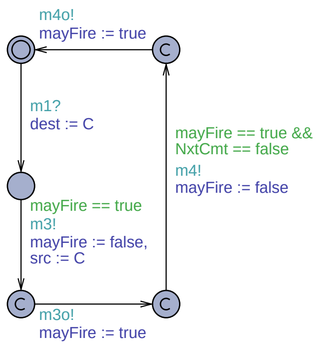
For this newly composed model, we check in UPPAAL the property $(l_{min} \leadsto l_{max})$, and it turns out to be satisfied. This indicates that $\mathcal{S}$ does satisfy the requirements that are specified in $\mathcal{L}$.



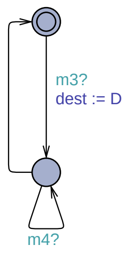
(a) TA A'



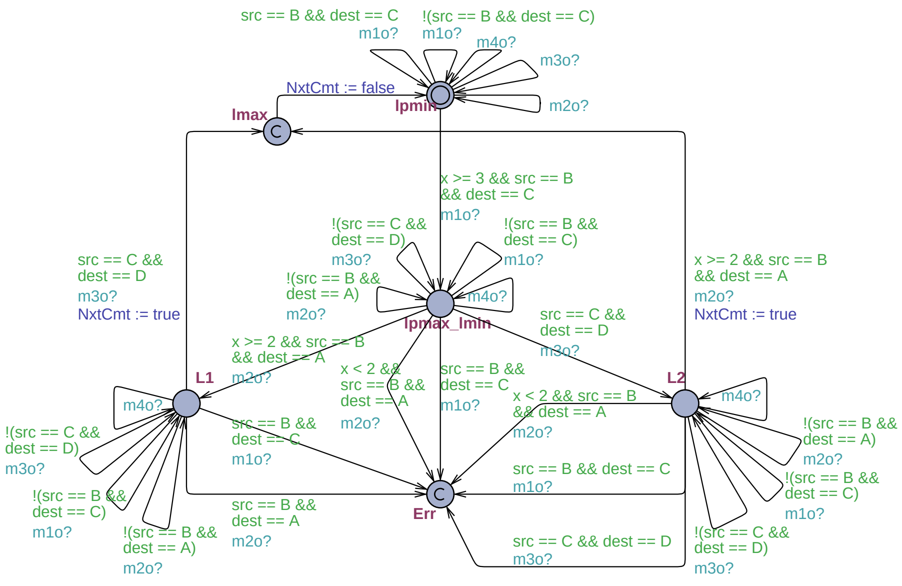
(b) TA B'



(c) TA C'



(d) TA D'



(e) Translated and modified observer TA $O'_{\mathcal{L}}$ of the LSC

Figure 7: The newly composed system $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$ that corresponds to Fig. 1.

If in $\mathcal{L}$ the condition of m_2 is changed from $x \geq 2$ to e.g. $x \geq 4$, then the property will not be satisfied. There will be a counterexample, e.g., when $O'_{\mathcal{L}}$ has to synchronize on the channel `m2o` in location `L2` of Fig. 7(e), but the value of clock x falls in $[3, 4)$, then it gets stuck there.

6 Conclusions

This paper deals with the verification of real-time systems against scenario-based requirements by using model transformation and event spying techniques. Since both the LSC to TA translation and the non-intrusive composing methods are automatic steps, our approach can be fully automated.

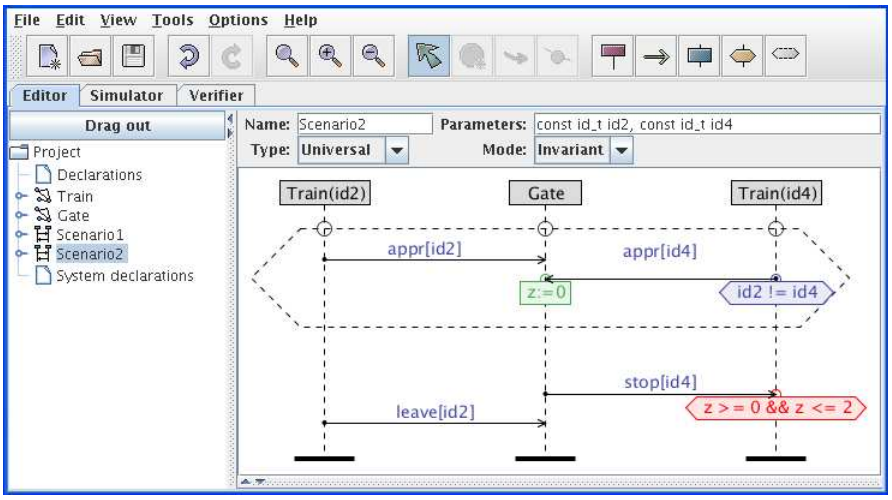


Figure 8: LSC requirements of the Train-Gate problem in Uppaal.

Based on previous work [Pus10], the translation algorithms in this paper have been implemented as an LSC-to-TA translator, which together with a newly developed LSC editor has been integrated into the Uppaal GUI and verification server [Bal09]. With the LSC editor, one can create LSC templates [3] (Fig. 8), which contain the LSC elements that are mentioned in this paper. Thanks to the LSC-to-TA translation and the subsequent non-intrusive composing mechanisms, one can check whether a network of timed automata satisfy the requirements that are specified in an LSC chart. Experiments with some examples such as the Train-Gate problems show the effectiveness of this method and tool.

Future work includes: (1) empirical evaluations for studying the applicability and scalability of this approach; (2) to support the translations of more chart elements such as subchart, if-then-else structure, loop, forbidden and ignored event, co-region, symbolic instances, and other chart modes; (3) to consider multiple charts for system modeling as well as for property specification; (4) to specify interaction scenarios for timed game solving and controller synthesis.

[3] In such an LSC template, the instance lines and the synchronization channels (messages) can be parameterized. When we make the system declarations, an LSC template can be instantiated to a concrete LSC chart. This is similar to we do with the TA templates in Uppaal.

Appendix: Proofs or lemmas and theorems

Lemma 1. If a configuration (c, v) of $\mathcal{L}$ corresponds to a semantic state (l, v) of $O_{\mathcal{L}}$, then: (1) each simregion s that follows (c, v) in $\mathcal{L}$ uniquely corresponds to an outgoing edge (l, l') in $O_{\mathcal{L}}$, and (2) the target configuration (c', v') of s in $\mathcal{L}$ uniquely corresponds to the target semantic state (l', v') in $O_{\mathcal{L}}$.

Proof. For each simregion s in $\mathcal{L}$ that immediately follows (c, v) w.r.t. the partial order $\preccurlyeq'$ of $\mathcal{L}$, according to Section 3.3.3, s uniquely corresponds to an outgoing edge (l, l') from l in $O_{\mathcal{L}}$. Since the valuation function v is the same in (l, v) as in (c, v), and the condition in s is straightforwardly copied onto the TA edge (l, l'), the simregion s can be stepped over if and only if the TA edge (l, l') can be taken. Moreover, the assignment (if any) in s is also straightforwardly copied onto the edge (l, l'). This indicates that the valuation function in the LSC target configuration will be still the same as in the TA target semantic state. Therefore, (c', v') uniquely corresponds to (l', v').

Specifically, if s is a non-message simregion that immediately follows (c, v) in $\mathcal{L}$, then according to the ASAP semantics, s will be stepped over immediately from (c, v). Accordingly, the source location l is a committed location, and the other outgoing edges that correspond to message-simregions will not be appended to l. All these ensure that the TA edge that corresponds to s is taken immediately from state (l, v). $\square$

Theorem 1: For any trace tr in $O_{\mathcal{L}}$: $tr \models \mathcal{L} \Leftrightarrow (O_{\mathcal{L}}, tr) \models (l_{min} \leadsto l_{max})$.

Proof. Let the initial cut of $\mathcal{L}$ be c_0. According to Section 3.3.2, c_0 corresponds to the initial location l_0 of $O_{\mathcal{L}}$. Since in the beginning all the clocks in $\mathcal{L}$ have the same initial values as in $O_{\mathcal{L}}$, the initial configuration (c_0, v_0) of $\mathcal{L}$ uniquely corresponds to the initial semantic state (l_0, v_0) of $O_{\mathcal{L}}$.

Only the legal behaviors (admissible traces) of $O_{\mathcal{L}}$ will be considered. We consider the following cases:

(1) $O_{\mathcal{L}}$ has only explicitly specified behaviors. By Lemma 1, each simregion that immediately follows (c_0, v_0) uniquely corresponds to an outgoing edge from TA location l_0, and the target configuration (c', v') in $\mathcal{L}$ uniquely corresponds to the target semantic state (l', v') in $O_{\mathcal{L}}$. On the other hand, in (c_0, v_0) of $\mathcal{L}$, there could be a time delay d if and only if (l_0, v_0) of $O_{\mathcal{L}}$ can have the same time delay d.

By recursively applying Lemma 1 and the above result, we can conclude that any timed trace tr in $O_{\mathcal{L}}$ is also a timed trace in $\mathcal{L}$.

Since $O_{\mathcal{L}}$ has only explicitly specified behaviors, we know that there is no undesired behavior in $O_{\mathcal{L}}$. If $tr \models \mathcal{L}$, then by definition this particular tr in $O_{\mathcal{L}}$ also satisfies the path formula $(l_{min} \leadsto l_{max})$, i.e., $(O_{\mathcal{L}}, tr) \models (l_{min} \leadsto l_{max})$. Therefore, we have $tr \models \mathcal{L} \Rightarrow (O_{\mathcal{L}}, tr) \models (l_{min} \leadsto l_{max})$.

The reverse implication is proved similarly.

(2) $O_{\mathcal{L}}$ include behaviors of unconstrained events or cold violations. In this case, each unconstrained event m at a particular cut c in $\mathcal{L}$ uniquely corresponds to an $\mathsf{m?}$-labeled self-loop edge at a corresponding location l in $O_{\mathcal{L}}$, and each cold violation uniquely corresponds to an edge leading to l_{pmin} (if any) or l_{min} (otherwise). The two-way implications are proved similarly.

(3) $O_{\mathcal{L}}$ include behaviors of prechart pre-matching. In this case, the semantics of $tr \models \mathcal{L}$ says whenever tr matches the prechart Pch, the main chart Mch will be matched afterwards (and must before Pch begins a next matching process). Considering that in $O_{\mathcal{L}}$, the locations l_{min} and l_{max} are two rendezvous points, thus $tr \models \mathcal{L}$ means exactly the satisfaction of $(l_{min} \rightsquigarrow l_{max})$ by tr.

To sum up, we conclude that for any trace tr in $O_{\mathcal{L}}$, we have $tr \models \mathcal{L} \Leftrightarrow (O_{\mathcal{L}}, tr) \models (l_{min} \rightsquigarrow l_{max})$. □

Lemma 2. If $O_{\mathcal{L}}$ has no committed location, and all $ch \in Chan$ are binary synchronization channels, then $\mathcal{S} \models \mathcal{L} \Leftrightarrow (\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \rightsquigarrow l_{max})$.

Proof. Let $(\bar{l}, v)$ be a semantic state of the network of TAs of $\mathcal{S}$, where $\bar{l}$ is a location vector, and v is the valuation of all clock variables. For each binary synchronization channel $ch \in Chan$, we have a transition $(\bar{l}, v) \xrightarrow{ch} (\bar{l}', v')$ if in two different processes of $\mathcal{S}$, there are two edges (l_i, l_{i+1}) and (l_j, l_{j+1}) labeled with $\mathsf{ch!}$ and $\mathsf{ch?}$, respectively, such that:

- $v \models g_i \wedge g_j$, where g_i and g_j are guards of the two edges, respectively;
- $\bar{l}' = \bar{l}[l_{i+1}/l_i, l_{j+1}/l_j]$;
- $v' = a_j(a_i(v))$, where a_i and a_j are the assignments of the two edges, respectively;
- $v' \models Inv_{i+1} \wedge Inv_{j+1}$, where Inv_{i+1} and Inv_{j+1} are the location invariants of the target locations of the two edges, respectively; and
- either (l_i or l_j or both are committed locations), or no other location in $\bar{l}$ is committed.

We need to show that the modifications of the original system model $\mathcal{S}$ and the observer TA $O_{\mathcal{L}}$ do not affect their legal (or admissible) behaviors (traces), i.e., the event notification mechanism and the locking mechanisms neither increase nor decrease the behaviors (traces) in $\mathcal{S}$ and $O_{\mathcal{L}}$. To this end, we prove that each synchronization in $\mathcal{S}$ uniquely corresponds to a pair of consecutive synchronizations in $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$.

$\Rightarrow$):

By $\mathcal{S} \models \mathcal{L}$ we know that the original system model $\mathcal{S}$ satisfies the requirements that are specified in the LSC chart $\mathcal{L}$. It follows that the observer TA $O_{\mathcal{L}}$ does not restrict the (legal) behaviors of $\mathcal{S}$.

If at a semantic state $(\bar{l}, v)$ of $\mathcal{S}$ there is a synchronization $(\bar{l}, v) \xrightarrow{ch} (\bar{l}', v')$, where $ch \in Chan$, we let the two coupling edges that carry ch! and ch? be (l_i, l_{i+1}) and (l_j, l_{j+1}), respectively. Clearly, they satisfy all the five requirements as listed above. According to the rules for modifying $\mathcal{S}$, edges (l_i, l_{i+1}) will correspond to two edges (l_i, l_i') and (l_i', l_{i+1}) in $\mathcal{S}'$, where l_i' is a newly added intermediate committed location. Also according to the rules, the semaphore *mayFire* evaluates to `false` only when the current control is in a newly added committed location (see Fig. 5(d)). Now that the control is in l_i in $\mathcal{S}'$, the semaphore *mayFire* should evaluate to `true`. This together with the item $v \models g_i \wedge g_j$ in the above requirements indicate that the guards for (l_i, l_i') and (l_j, l_{j+1}) in $\mathcal{S}'$ to synchronize on ch are both satisfied. Besides, items 3-5 in the above requirements also apply to the ch-synchronization at (l_i, l_i') and (l_j, l_{j+1}). Therefore, there exists a transition $(\bar{l}, v) \xrightarrow{ch} (\bar{l}'', v')$ in $\mathcal{S}'$ with (l_i, l_i') and (l_j, l_{j+1}) as the coupling edges, where $\bar{l}'' = \bar{l}'[l_i'/l_i, l_{j+1}/l_j]$.

The second edge (l_i', l_{i+1}) in $\mathcal{S}'$ will be immediately coupled with a corresponding edge in $O_{\mathcal{L}}'$. By the assumption $\mathcal{S} \models \mathcal{L}$, we know $O_{\mathcal{L}}$ does not restrict the behaviors of $\mathcal{S}$ via its own conditions (e.g., via g_3 in Fig. 5(e)). This means that the cho-synchronization between $\mathcal{S}'$ and $O_{\mathcal{L}}'$ will not get stuck there due to the restrictions of $O_{\mathcal{L}}'$. Since after this synchronization, the clock variables in $\mathcal{S}'$ remain unchanged, we know that the location invariant Inv_{i+1} on l_{i+1} of $\mathcal{S}'$ will still be satisfied. After this synchronization, the two target locations in $\mathcal{S}'$ will be l_{i+1} and l_{j+1}, thus coinciding with the corresponding target locations l_{i+1} and l_{j+1} in $\mathcal{S}$. Therefore, we can conclude that given a trace tr in $\mathcal{S}$, there exists a unique trace tr' in $(\mathcal{S}' \,||\, O_{\mathcal{L}}')$ such that tr' and tr correspond.

By the definition of $\mathcal{S} \models \mathcal{L}$ (see Section 3.2), we know that if a timed trace μ in $\mathcal{S}$ arrives at the minimal cut of the main chart of $\mathcal{L}$, then μ must always be able to reach the maximal cut of that main chart. By Theorem 1 and Section 3.3, we know that if μ arrives at location l_{min} of $O_{\mathcal{L}}'$, then μ must always be able to reach location l_{max} of $O_{\mathcal{L}}'$.

Since each trace μ in $\mathcal{S}$ can be equivalently mapped to a trace μ' in $(\mathcal{S}' \,||\, O_{\mathcal{L}}')$, clearly, if any μ' arrives at location l_{min} of $O_{\mathcal{L}}'$, then that μ' must always be able to reach location l_{max} of $O_{\mathcal{L}}'$.

Since l_{min} and l_{max} are two locations in $(\mathcal{S}' \,||\, O_{\mathcal{L}}')$, the above requirement can thus be formulated as a UPPAAL property $(\mathcal{S}' \,||\, O_{\mathcal{L}}') \models (l_{min} \leadsto l_{max})$.

$\Leftarrow$):

Similarly, we need to prove that each trace tr' in $(\mathcal{S}' \,||\, O_{\mathcal{L}}')$ uniquely corresponds to a trace tr in $\mathcal{S}$ such that tr' and tr are equivalent.

Assume that in $(\mathcal{S}' \,||\, O_{\mathcal{L}}')$ there is a synchronization $(\bar{l}, v) \xrightarrow{c} (\bar{l}', v')$.

If $c \in Chan$, then after removing "*mayFire* == `true`" from the condition and removing "*mayFire* := `false`" from the assignment of the emitting edge (see Fig. 5(d)), the edge becomes exactly the corresponding edge in $\mathcal{S}$. Note that the invariant (if any) at the target location of this emitting edge is irrelevant of the semaphore *mayFire*. This indicates that the synchronization between the corresponding edges in $\mathcal{S}$ can also fire.

If $c \in \{cho \mid ch \in Chan\}$, then the source location of the c!-emitting edge in $\mathcal{S}'$ must be a newly added committed location. This c! will be synchronized with a c?-receiving edge in $O'_{\mathcal{L}}$. And it will bring the control in $\mathcal{S}'$ from the committed location to the target location, which coincides with the corresponding target location in $\mathcal{S}$. Due to the use of semaphore *mayFire*, no other synchronizations in $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$ can preempt the execution of this c-synchronization.

The rest of the transitions in $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$ are just the same as those in $\mathcal{S}$. Therefore we can conclude that a trace tr' in $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$ uniquely corresponds to a trace tr in $\mathcal{S}$ such that tr' and tr are equivalent. Now that $(\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \leadsto l_{max})$, according to the semantics of LSC chart satisfaction, we have $\mathcal{S} \models \mathcal{L}$. $\square$

Theorem 2. $\mathcal{S} \models \mathcal{L} \Leftrightarrow (\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \leadsto l_{max})$.

Proof. This theorem is a generalization of Lemma 2 by canceling the restrictions.

If $ch \in Chan$ is a broadcast channel, the semantics of ch-synchronization is a little different. Since the modifications of the emitting edges in $\mathcal{S}$ do not affect the receiving edges in $\mathcal{S}$, we can still have a one-to-one mapping between the traces in $\mathcal{S}$ and in $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$.

If there are committed locations in $O'_{\mathcal{L}}$, then we use the second semaphore *NxtCmt* to guarantee the non-interrupted execution at those committed locations in $O'_{\mathcal{L}}$. Since an edge (l, l') starting from a committed location l in $O'_{\mathcal{L}}$ represents an internal action (local) transition, it needs no synchronization with $\mathcal{S}'$. Thus the edge does not affect the behavior of $\mathcal{S}'$.

To sum up, there is a one-to-one mapping of the traces in $\mathcal{S}$ and $(\mathcal{S}' \,||\, O'_{\mathcal{L}})$, even in the presences of broadcast channels in $\mathcal{S}$ and committed locations in $O_{\mathcal{L}}$. Thus we have $\mathcal{S} \models \mathcal{L} \Leftrightarrow (\mathcal{S}' \,||\, O'_{\mathcal{L}}) \models (l_{min} \leadsto l_{max})$. $\square$

Paper B: Scenario-Based Analysis and Synthesis of Real-Time Systems Using Uppaal

Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, Saulius Pusinskas
Center for Embedded Software Systems (CISS)
Department of Computer Science
Aalborg University, Denmark

Abstract

We propose an approach to scenario-based analysis and synthesis of real-time embedded systems. The inter-process behaviors of a system are modeled as a set of driving universal Live Sequence Charts (LSCs), and the scenario-based user requirement is specified as a separate monitored universal or existential LSC. By translating the set of LSCs into a behavior-equivalent network of timed automata (TA), we reduce the problems of model consistency checking and property verification to CTL real-time model checking problems. Similarly, we reduce the problem of centralized synthesis for open systems to a timed game solving problem. We implement a prototype LSC-to-TA translator, which can be linked to our LSC editor and the existing real-time model checker UPPAAL and timed game solver UPPAAL-TIGA. Preliminary experiments on a number of examples and a case study show the applicability and effectiveness of this approach.

Keywords: Real-Time Systems, Live Sequence Charts (LSCs), Timed Automata (TA), Consistency Checking, Verification, Synthesis

1 Introduction

In the early stage of model-based development of real-time embedded systems, it is desirable to prototype a system in question and its operating environment using a number of use cases and scenarios. A next round refinement of the system model and the transition from model to implementation can start only if the scenario-based model is checked to be consistent (i.e., implementable), and correct with respect to some specified scenario-based user requirements. Furthermore, from a correct-by-construction perspective, as a part of the long-known dream for "automated system design", we may wish to synthesize executable object systems from scenario-based descriptions.

Live Sequence Chart (LSC) [HM03] is a scenario-based specification and programming language, which can describe systems in an assume-guarantee style. A universal LSC chart can optionally contain a *prechart*, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body (the *main chart*). Essentially, LSC extends Message Sequence Chart (MSC) [IT99] by adding modalities. The *existential* and *cold* (resp. *universal* and *hot*) modalities represent the provisional (resp. mandatory) global and local requirements, respectively. An existential LSC chart (resp. universal LSC chart) specifies restrictions over at least one satisfying (resp. all possible) system runs. A cold condition may be violated, whereas a hot one must be satisfied. The rich language facilities and the unambiguous semantics make LSC a nice visual formalism for early-stage characterization of distributed, real-time and embedded systems.

Scenario-based approaches that use LSCs enjoy the advantages of piecewise incremental construction of system models, i.e., new pieces of scenarios can be added into the models during the development process. However, scenario-based analysis such as model consistency checking and property verification (i.e., to check whether an LSC-modeled system satisfies a scenario-based requirement) are difficult due to the need to consider both the explicitly as well as implicitly specified behaviors in each scenario, and the interplays among the different scenarios. Scenario-based synthesis is difficult because all possible scenario interactions have to be considered[1]. The problems become even more complicated for real-time systems, as time-enriched LSCs may contain subtle timing errors that are difficult to diagnose.

Powerful verification techniques and tools are utilized to assist scenario-based analysis and synthesis for LSCs, e.g. in smart play-out [HKMP02, HKP04, CHK08], satisfiability checking [SD05a, CHK08], consistency checking [SD05a] and synthesis [HK02, HKP05, SD05b, BS07, KPP09].

[1]It has been shown that for an untimed subset of the LSC language, consistency checking and the subsequent synthesis is at least EXPTIME-complete, and model checking a given system implementation (e.g. in I/O automata) against an LSC specification is complete for co-NP (for centralized implementation of a closed system) or PSPACE (otherwise) [Bon05].

Some of these efforts address the problems mainly from a theoretical viewpoint [HK02, BS07]. Some of them handle property verification with only existential charts [SD05a, CHK08]. A number of them have no real-time support [HK02, HKMP02, SD05a, SD05b, BS07]. While synthesis for LSC-modeled real-time systems is supported in [HKP05], it is *incomplete* in the sense that some systems are announced not synthesizable while they actually are. The reason is that the smart play-out mechanism [HKMP02, HKP04] that [HKP05] relies on implements only a *local* consistency checking where the Play-Engine looks only one super-step ahead in the LSC state space. A recent work [KPP09] tackles this by employing controller synthesis techniques to achieve complete consistency checking and subsequent complete synthesis. However as an extension to smart play-out, this method, like the earlier work [HKP05], is limited to discrete time and a restricted form of timing constraints [2].

In this paper we equip a kernel subset of the LSC language with timed automaton (TA)[AD94] -like real-valued clock variables and clock constraints. We use a set of driving universal LSC charts (collectively called an *LSC system*) to model the inter-process interaction behaviors of the system in question, and use a monitored universal/existential chart to specify the user requirement. We translate the LSC system into a behavior-equivalent network of interacting timed automata, which can properly mimic the important LSC features such as message sending and intra/inter-chart coordinations. The problems of consistency checking and property verification can be reduced to `CTL` real-time model checking problems in Uppaal [BDL04]. Furthermore, by viewing the interaction between the controllable system processes ($\mathcal{S}ys$) of an LSC system and their "hostile" (uncontrollable) environment processes ($\mathcal{E}nv$) as playing a (safety) game with the aim of constantly avoiding hot violations, we can reduce the problem of scenario-based synthesis for open systems (i.e., $\mathcal{S}ys$) to a game solving problem. The timed game solver Uppaal-Tiga [BCD$^+$07] can be employed to check the solvability, and if yes, to generate a strategy for $\mathcal{S}ys$. Compared with existing work, our approach of scenario-based analysis and synthesis features:

- TA-like real-valued clock variables and clock constraints (compared with [HKP04, HKP05, CHK08, KPP09]);
- Complete consistency checking and synthesis (compared with [HK02, HKP05]);
- Property verification with the properties being specified as universal charts (compared with [SD05a, CHK08]); and
- Automated, tool-supported approach (compared with [HK02, BS07]).

[2] In the original definition and the Play-Engine implementation of time-enriched LSC [HM03], the special `Clock` object has a property *Time* which is an integer variable, and a method *Tick* which each time increases *Time* by 1. Timing constraints take the form of only "*Time* **op** (time variable + delay expression)", where **op** is an relational operator.

The benefits of building the scenario-based analysis and synthesis methods on top of the well-developed real-time model checking and timed game solving engines are twofold: (1) for system analysts and designers who are interested in early-stage validations and automated system designs using live sequence charts, now they can carry out scenario-based analysis and synthesis of *timed* systems without having to develop and implement the underlying algorithms; (2) for the users of conventional model checkers that work on state/transition-based models and temporal logical properties, now they can horizontally scale up to *scenario-based* models and *scenario-based* user requirements.

1.1 Related Work

In addition to being used as a requirement specification language [DK01, KW01, KTWW06], LSC can also be used as a modeling language [HK00, HK02, HKMP02, HKP04, Bon05, SD05a, SD05b, HKP05, CHK08, KPP09]. In the latter case, problems of scenario-based analysis such as model consistency checking and property verification, and of scenario-based synthesis arise. This paper will mainly consider the latter usage of LSC charts.

Analysis of scenario-based behaviors can be carried out inside the Play-engine [HM03] according to the operational semantics of LSC [HK00, HK02, HKMP02, HKP04, HKP05, CHK08, KPP09], or achieved by first transforming LSCs into other formalisms such as CSP [SD05a, SD05b] and timed büchi automata (TBA) [KW01], and then calling for other mature techniques and tools to accomplish the tasks. In this paper we will follow the latter approach to take advantage of the TA formalism [AD94], the real-time model checker UPPAAL [BDL04] and the timed game solver UPPAAL-TIGA [BCD$^+$07]. Similar in spirit to the approaches of [HKMP02, SD05a], during the translation we will create one process for each instance line of the charts, and thus avoid the explicit construction of the global transition system.

A set of LSC charts are *consistent* if and only if these charts are not internally contradictory, i.e., they can be satisfied by a certain state-based object system [HK00, HK02]. Consistency checking of LSC systems is a prerequisite step towards *synthesis* [HKP05, Bon05, SD05a, SD05b, KPP09], i.e., to construct one such satisfying state-based object system. In this paper, we consider both the consistency checking and the synthesis problems, and will reduce them to the model checking and game solving problems, respectively.

Current work on verification of scenario-based requirement either has state/transition-based object systems as the subject, e.g., STATEMATE model implementations [DK01, KW01] and Kripke structures [KTWW06], or has an LSC system as the subject. In the latter case, existing work mainly concerns satisfiability checking [SD05a, CHK08], i.e., to check whether the requirement that is expressed as an existential chart can be satisfied by the LSC system. In this paper we will also check whether a requirement specified as a *universal* chart can always be respected by the LSC system.

1.2 Organization

Section 2 presents our timed extensions to a subset of the LSC language and defines a trace-based semantics. Section 3 describes how we translate LSCs into a network of timed automata, shows how complex the outcome of the translation is, and shows the behavior-equivalence of the translation. The scenario-based analysis and synthesis problems are formulated as model checking and game solving problems in Section 4. The prototype tool implementation and preliminary experiments are reported in Section 5. We conclude in Section 6. The detailed translation rules, the complexity analysis of the translation outcomes, and the proofs of lemmas and theorems in Sections 3.4 and 4 are provided in the appendices.

2 Live Sequence Charts

In this paper, LSC in its simplest form is a message-only untimed chart, i.e., there are only language elements of instance lines, locations, messages and precharts/ main charts.

We make the *synchrony* hypothesis, i.e., system events consume no real time and time may elapse only between events. In this way message synchronizations will be instantaneous, i.e., the sending and reception of a message are assumed to happen at the same moment in time. Therefore the terms of (message sending or receiving) *event* and *message* will be used interchangeably.

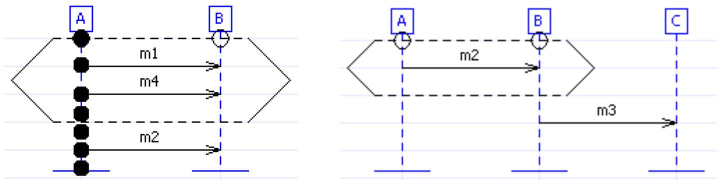
A chart has a *role*, either for system modeling (i.e., as a driving chart), or for system property specification (i.e., as a monitored chart). Driving charts can only be universal charts, whereas a monitored chart can be either a universal or an existential chart. In this paper we use a set of universal charts to model the behaviors of the system in question, and use a separate universal or existential chart to specify the system property.

A universal LSC chart has an *activation mode*, i.e., how often a chart should be activated. In this paper, we consider the invariant mode, i.e., the prechart is being constantly monitored, regardless of whether any incarnation of the chart has entered its main chart portion.

2.1 Syntax and semantics for a single universal chart

A universal LSC chart has a main chart (Mch) and optionally a prechart (Pch). If it has no prechart, then it can be simply treated as having a satisfying prechart. In this paper we assume that a universal chart has a prechart.

We start with message-only untimed charts, see Fig. 1 for the examples.



(a) chart 1 (b) chart 2

Figure 1: Two consistent untimed charts.

Given a universal chart $\mathcal{L}$, let $I = inst(\mathcal{L})$ be the set of instance lines (i.e., processes) in $\mathcal{L}$. Along each instance line $I_i \in I$ there are a finite set of "positions" $pos(\mathcal{L}, I_i) = \{0, 1, 2, \ldots, p\_max_{\mathcal{L},I_i}\} \subset \mathbb{N}_{\geq 0}$. See Fig. 1(a), black filled circles. Specifically, along each instance line I_i there are four "standard" positions $StdPos(\mathcal{L}, I_i) = \{Pch\_top, Pch\_bot, Mch\_top, Mch\_bot\} \subset pos(\mathcal{L}, I_i)$, denoting the entry/exit points of the prechart/main chart, respectively, such that:

- $0 = Pch\_top < Pch\_bot < Mch\_top < Mch\_bot = p\_max_{\mathcal{L},I_i}$; and
- $Pch\_bot + 1 = Mch\_top$. □

A chart *location* is a position on a certain instance line of the chart. The set of all locations of chart $\mathcal{L}$ are denoted as:

$$loc(\mathcal{L}) = \{\langle I_i, p\rangle \mid I_i \in inst(\mathcal{L}), p \in pos(\mathcal{L}, I_i)\}.$$

The set of all *message-anchoring* locations of $\mathcal{L}$ are denoted as:

$$loc_M(\mathcal{L}) = \{\langle I_i, p\rangle \mid I_i \in inst(\mathcal{L}), p \in pos(\mathcal{L}, I_i) \backslash StdPos(\mathcal{L}, I_i)\}.$$

Furthermore, we define function $psn : loc(\mathcal{L}) \rightarrow \bigcup_{I_i \in inst(\mathcal{L})} pos(\mathcal{L}, I_i)$ to project a location to its **p**o**s**itio**n** on its instance line.

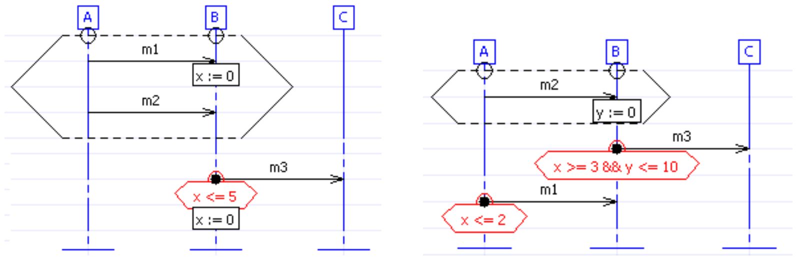
Let $ML(\mathcal{L})$ be the set of **m**essage **l**abels (or "signals", or "channels" in UPPAAL) of chart $\mathcal{L}$. A *message occurrence* $mo = (\langle I_i, p\rangle, m, \langle I_{i'}, p'\rangle) \in loc_M(\mathcal{L}) \times ML(\mathcal{L}) \times loc_M(\mathcal{L})$ corresponds to instance I_i, while in its position $(p-1)$, sending signal $m \in ML(\mathcal{L})$ to instance $I_{i'}$ at its position $(p'-1)$, and then arriving at positions p and p', respectively. We call $lab(mo) = m$ the message label, $head(mo) = \langle I_{i'}, p'\rangle$ and $tail(mo) = \langle I_i, p\rangle$ the message head and tail locations, and $src(mo) = I_i$ and $dest(mo) = I_{i'}$ the source and destination instances, respectively. We use $loc(mo) = \{head(mo), tail(mo)\}$ to denote the message anchoring locations. The set of all message occurrences in chart $\mathcal{L}$ are denoted as:

$$MO(\mathcal{L}) \subseteq \{(\langle I_i, p\rangle, m, \langle I_{i'}, p'\rangle) \in loc_M(\mathcal{L}) \times ML(\mathcal{L}) \times loc_M(\mathcal{L}) \mid i \neq i',$$
$$p \leq StdPos(\mathcal{L}, I_i).Pch\_bot \Leftrightarrow p' \leq StdPos(\mathcal{L}, I_{i'}).Pch\_bot\}.$$

We omit the parameter $\mathcal{L}$ in $MO(\mathcal{L})$ (and thus abbreviating it as MO) when it is clear from the context. Furthermore, we use $\Sigma = MA(\mathcal{L})$ to denote the projection of $MO(\mathcal{L})$ onto $inst(\mathcal{L}) \times ML(\mathcal{L}) \times inst(\mathcal{L})$. In this way, we get the **m**essage alphabet Σ, where each letter is a *message* which denotes that a particular signal is sent from one to another objects (instance lines). For a given message occurrence, we may overload its "message label" to also denote the corresponding letter in Σ.

This paper does not consider concurrent messages, thus each location can be the end point of at most one message occurrence in the chart.

Now we continue to define our timed extensions to the above kernel subset of the LSC language. In our time-enriched LSC charts, there are further elements of (clock) variables, conditions (clock constraints), assignments (clock resets) and simregion (i.e., "simultaneous region"). Fig. 2 gives two example time-enriched LSC charts.



(a) chart 1 (b) chart 2

Figure 2: Two consistent time-enriched charts.

Assume that in chart $\mathcal{L}$ there are a finite set X of real-valued *clock variables* that range over $\mathbb{R}_{\geq 0}$. A *clock valuation* is a function $v : X \longrightarrow \mathbb{R}_{\geq 0}$ that maps each clock variable to a non-negative real number, also denoted $v \in {\mathbb{R}_{\geq 0}}^X$.

A *clock constraint* is of the form $x \bowtie n$ or $x - y \bowtie n$ where $x, y \in X$, $n \in \mathbb{Z}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let $B(X)$ be the set of finite conjunctions over these constraints. The set of *conditions* (or guards) in the chart are denoted $G \subseteq B(X)$. A condition $g \in G$ has a temperature, denoted $g.temp$, which may be either hot or cold in the main chart, and only cold in the prechart.

A *clock reset* is of the form $x := 0$ where $x \in X$. An *assignment* (or update) a is the union of a finite set of clock resets. For simplicity we use a to denote the set of clocks to be reset. The set of all assignments in the chart is denoted $A \subseteq 2^X$. We can also view $a \in A$ as a transformer on the functions of clock valuations, and as such the new valuation of v after assignment a is denoted as $v' = a(v)$.

In our time-enriched LSCs, each message occurrence mo can be optionally associated with a condition g and/or an assignment a. The intuitive meaning of message synchronization $[g]\, mo\, /a$ from location $\langle I_i, p\rangle$ to $\langle I_{i'}, p'\rangle$ is that, if when mo occurs, the clock valuation v satisfies g, then this synchronization can fire; and immediately after the firing, v will be updated according to a. A message occurrence and the corresponding condition and/or assignment attached thereto can be collectively viewed as an atomic step of LSC execution, i.e., they take place at the same moment in time, hence they constitute a simregion. We denote the set of all simregions as *SR*. For simplicity, we do not consider stand-alone conditions or assignments, i.e., we assume that any condition and assignment is associated with a certain message.

In an LSC chart $\mathcal{L}$, every location is either associated with a simregion, or it is an entry/exit point of the prechart/main chart. We define a labeling function $\lambda : loc(\mathcal{L}) \to SR \cup \{nil\}$ to map a location of the former type to its corresponding simregion, and a location of the latter type to *nil*.

Locations in a chart $\mathcal{L}$ are partially ordered by the following rules:

- Along each instance line I_i: location l is above $l' \Rightarrow (l \leq l') \land \neg(l' \leq l)$; and
- All locations in the same simregion have the same order, $\forall s \in SR, \forall l, l' \in loc(\mathcal{L})\,.\,(\lambda(l) = s) \land (\lambda(l') = s) \Rightarrow (l \leq l') \land (l' \leq l)$. □

The partial order relation $\preccurlyeq \subseteq loc(\mathcal{L}) \times loc(\mathcal{L})$ is defined as a transitive closure of $\leq$.

Definition 1 (cut of an LSC chart)**.** *A* cut *of a chart $\mathcal{L}$ is a set $c \subseteq loc(\mathcal{L})$ of locations that span across all the instance lines in $\mathcal{L}$ which satisfies the properties of:*

- Downward-closure*. If a location l is included in cut c, so are all of its predecessor locations: $\forall l,\, l' \in loc(\mathcal{L}).\,(l \in c \,\land\, l' \preccurlyeq l) \Rightarrow l' \in c$; and*
- Intra-chart coordination integrity*. If a Mch_top position of a certain instance line is included in the cut, then the Mch_top positions of all other instance lines are also included in the cut: $\exists l \in loc(\mathcal{L}),\, I_i \in inst(\mathcal{L})\,.\,((StdPos(\mathcal{L}, I_i).Mch\_top \leq psn(l)) \land (psn(l) \leq StdPos(\mathcal{L}, I_i).Mch\_top) \land (l \in c) \Rightarrow \forall l' \in loc(\mathcal{L}), I_{i'} \in inst(\mathcal{L})\,.\,((StdPos(\mathcal{L}, I_{i'}).Mch\_top \leq psn(l')) \land (psn(l') \leq StdPos(\mathcal{L}, I_{i'}).Mch\_top) \Rightarrow l' \in c))$.* □

For a cut c, we use $loc(c)$ to denote its *frontier*, i.e., the set of locations that constitute the downward borderline progressed so far. The location where c "cuts" instance line $I_k \in I$ is denoted $loc(c)_{\langle k \rangle}$.

Given a cut $c \subseteq loc(\mathcal{L})$ and a simregion $s \in SR$, we say s is *enabled* at cut c (with respect to the partial order relation), denoted $c \xrightarrow{s}$, if, $\forall\, l \in c,\ l' \in loc(s)\,.\,((l \preccurlyeq l') \land \neg(l' \preccurlyeq l)) \land (\nexists\, l'' \in loc(\mathcal{L}) \backslash (c \cup loc(s))\,.\,(l \preccurlyeq l'' \,\land\, l'' \preccurlyeq l'))$. The enabledness of message occurrences can be defined similarly.

A cut c' is an *s-successor* of cut c, denoted $c \xrightarrow{s} c'$, if s is enabled at c (w.r.t. the partial order), and c' is achieved by adding the set of locations that s anchors into c, or formally, $(c \xrightarrow{s}) \land (c' = c \cup loc(s))$.

A cut c is *minimal*, denoted $\top$, if it "cuts" each instance line at its top location; and c is *maximal*, denoted $\bot$, if if it "cuts" each instance line at its bottom location. The minimal and maximal cuts of the prechart and main chart are denoted $Pch.\top$, $Pch.\bot$, $Mch.\top$ and $Mch.\bot$, respectively. The frontiers of minimal and maximal cuts do not contain simregion anchoring points. Rather the cuts $Pch.\bot$ and $Mch.\bot$ each represent a requirement for compulsory synchronization for all the instance lines in the chart. Thus the partial order relation $\preccurlyeq$ on $loc(\mathcal{L})$ is extended as follows (and finally also extended to its transitive closure):

- All locations in the frontier of the same minimal or maximal cut have the same order, $\forall c \in \{Pch.\top,\ Pch.\bot, Mch.\top, Mch.\bot\}\,, \forall l, l' \in loc(c)\,.\,(l \preccurlyeq l') \wedge (l' \preccurlyeq l)$. $\square$

Definition 2 (configuration)**.** *A* configuration *of an LSC chart is a tuple* (c, v)*, where* c *is a cut and* v *is a clock valuation.* $\square$

For $d \in \mathbb{R}_{\geq 0}$, notation $(v + d) : X \rightarrow \mathbb{R}_{\geq 0}$ means that the function v is shifted by d such that $\forall x \in X\,.\,(v(x + d) = v(x) + d)$.

A configuration at the minimal cut $\top$ with all clocks assigned their initial values (e.g., 0's) is called the *initial* configuration.

A configuration can be viewed as a "semantic state" of a time-enriched LSC chart. A universal chart starts from the initial configuration, advances from one to a next configuration, until a hot violation[3] occurs, or until the chart arrives at the maximal cut configuration and then starts all over again (i.e., to begin a next round execution).

There could be three kinds of advancement steps between two configurations (c, v) and (c', v') of a time-enriched LSC chart:

- *Message synchronization step.* Given a simregion s which consists of an m-labeled message occurrence mo $(m \in \Sigma)$, and optionally a condition g and/or an assignment a, there is a message synchronization step $(c, v) \xrightarrow{m} (c', v')$ if,
  - (normal advancement). $c \xrightarrow{s} c'$, $v \models g$, and $v' = a(v)$; or
  - (cold violation). $c' = Pch.\top$, $v' = v$, and either
    - mo is not enabled at cut c in the prechart (w.r.t. the partial order relation); or
    - $(v \not\models g) \wedge (g.temp = \text{cold})$;
- *Silent step.* There is a silent step $(c, v) \xrightarrow{\tau} (c', v')$ if either

[3]A *hot* violation means that some mandatory requirements are not satisfied. In this paper, it refers to the situations that in the main chart the event partial order is violated or a hot condition evaluates to false. In comparison, a *cold* violation means that some provisional requirements are not satisfied (therefore it is not a big deal). In this paper, it refers to the situations where the event partial order is violated in the prechart, or a cold condition evaluates to false.

  - $(c = Pch.\bot,\ \ c' = Mch.\top,\ \ v' = v)$; or
  - $(c = Mch.\bot,\ \ c' = Pch.\top,\ \ v' = v)$; or
  - c' is reached because an instance line moves to its bottom location in *Pch* or *Mch* autonomously (this happens when the instance line will not interact with other instance lines before it reaches its bottom location in *Mch* or *Pch*). Formally, there exists an instance I_k such that $v' = v$ and either
    - $loc(c')_{\langle k\rangle} = (loc(Pch.\bot))_{\langle k\rangle},\ \ psn(loc(c')_{\langle k\rangle}) = psn(loc(c)_{\langle k\rangle}) + 1$, and $loc(c')_{\langle i\rangle} = loc(c)_{\langle i\rangle}$ for all $i \neq k$; or
    - $loc(c')_{\langle k\rangle} = (loc(Mch.\bot))_{\langle k\rangle},\ \ psn(loc(c')_{\langle k\rangle}) = psn(loc(c)_{\langle k\rangle}) + 1$, and $loc(c')_{\langle i\rangle} = loc(c)_{\langle i\rangle}$ for all $i \neq k$;

- *Time delay step.* There is a time delay step $(c, v) \xrightarrow{d} (c', v')$ where $d \in \mathbb{R}_{\geq 0}$ if: $c' = c$, $v' = v + d$, and whenever there are message occurrences that are enabled at cut c (w.r.t. both the partial order relation and the guard), then after delay d there exists at least one of them that is still enabled at the same cut, i.e., $\exists s \in SR \,.\, \exists\, mo, g \in s \,.\, \forall d' \in [0, d] \,.\, (c \xrightarrow{s}) \wedge (v + d') \models g$. $\square$

Similarly, if in the main chart, an m-labeled message violates $\preccurlyeq$, or $(v \not\models g \wedge g.temp = \text{hot})$, then the configuration (c, v) is said to be *hot-violated*, denoted $(c, v) \overset{m}{\not\rightarrow}$.

Definition 3 (run of an LSC chart)**.** *A* run *of a time-enriched universal LSC chart is a sequence of configurations* $(c^0, v^0) \cdot (c^1, v^1) \cdot \ldots$ *that are connected by the advancement steps, i.e.,* $\forall i \geq 0 \,.\, \exists\, u_i \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \,.\, (c^i, v^i) \xrightarrow{u_i} (c^{i+1}, v^{i+1})$. $\square$

The transition relation $\rightarrow$ as mentioned above each time consumes only a single letter $u \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. We extend it to $\rightarrow^*$ such that it consumes a (finite or infinite) word $w \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$.

Let Π correspond to the set of all possible messages that occur in a state/transition-based system model (i.e., a network of timed automata), or be the set of all messages in an object interaction-based system model (i.e., a set of driving universal LSC charts). In the latter case, the message alphabet for the LSC system model $\mathcal{LS} = \{\mathcal{L}_i \mid 1 \leq i \leq n\}$ is $\Pi = \bigcup_{i=1}^{n} \Sigma_i = \bigcup_{i=1}^{n} MA(\mathcal{L}_i)$.

Definition 4 (satisfaction of a prechart/main chart)**.** *A timed trace* $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies *an LSC prechart or main chart* $\mathcal{C}$*, denoted* $\gamma \models \mathcal{C}$*, if its projection* $\gamma|_{(\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})}$ *has a prefix* μ *which is the accepted word of a run that successfully exercises* $\mathcal{C}$*, and no prefix of it ever leads to a hot violation. Formally,* $(\exists \mu \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, \xi \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, \exists v' \in {\mathbb{R}_{\geq 0}}^X \,.\, (\gamma|_{(\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})} = \mu \cdot \xi) \wedge (\top, v^0) \xrightarrow{\mu}{}^* (\bot, v'))\ \wedge\ (\nexists \mu' \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, \xi \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, \forall m \in \Sigma \,.\, ((\gamma|_{(\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})} = \mu' \cdot m \cdot \xi) \wedge (\top, v^0) \xrightarrow{\mu'}{}^* \bullet \overset{m}{\not\rightarrow}))$. $\square$

A finite trace $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*$ satisfies chart $\mathcal{C}$ *exactly*, denoted $\gamma \Vdash \mathcal{C}$, iff $(\gamma \models \mathcal{C}) \wedge \exists \mu \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*,\ v' \in {\mathbb{R}_{\geq 0}}^X . (\gamma|_{(\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})} = \mu) \wedge ((\top, v^0) \overset{\mu}{\rightarrow}{}^* (\bot, v'))$.

Now we define the satisfaction relation for a full universal chart (under the invariant activation mode):

Definition 5 (satisfaction of a universal LSC chart)**.** *A timed trace* $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies *(passes) a universal chart $\mathcal{L}$, denoted $\gamma \models \mathcal{L}$, iff whenever a finite sub-trace matches the prechart, then the main chart is matched immediately afterwards. Formally,* $\forall \alpha, \mu \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*, \beta \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega . (\alpha \cdot \mu \cdot \beta = \gamma) \wedge (\mu \Vdash Pch) \Rightarrow (\beta \models Mch)$. □

A timed language $Lang \subseteq (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies $\mathcal{L}$, denoted $Lang \models \mathcal{L}$, iff $\forall \gamma \in Lang . \gamma \models \mathcal{L}$. Clearly, $Lang$ characterizes the system behaviors that respect $\mathcal{L}$.

When $\mathcal{L}$ is used as a monitored chart, then for a network $\mathcal{S}$ of timed automata, we use $\mathcal{S} \models \mathcal{L}$ to denote that the timed traces (language) of $\mathcal{S}$ satisfy LSC $\mathcal{L}$.

2.2 Semantics for a set of universal charts

For an LSC system $\mathcal{LS}$ which consists of a set of driving universal charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$, we denote $\bar{c} = (c_1, c_2, \ldots, c_n)$ as a *cut vector*, and v as a *valuation* of all of the clock variables in $\mathcal{LS}$. Each member cut of $\bar{c}$ is denoted as $c_i = (\bar{c})_i$, $1 \leq i \leq n$. We call $(\bar{c}, v)$ a *global configuration* of $\mathcal{LS}$.

Let $(\bar{c}, v)$ be a global configuration of an LSC system $\mathcal{LS}$. Assume that there are message occurrences $mo_1, \ldots, mo_k$ ($1 \leq k \leq n$, each in a different chart) that are simultaneously enabled at $((\bar{c})_i, v)$, $1 \leq i \leq k$, and that these message occurrences are the same message, i.e., they have exactly the same message label and the same source and destination instances, i.e., $\exists m \in \Pi$, $\mathcal{L}_j \in \mathcal{LS} . \exists I_a, I_b \in inst(\mathcal{L}_j) . \forall 1 \leq i \leq k . (lab(mo_i) = m) \wedge (src(mo_i) = I_a) \wedge (dest(mo_i) = I_b)$. In this case, these identically labeled message occurrences are said to be *enabled* at global configuration $(\bar{c}, v)$ w.r.t. their respective partial order relations.

Given a global configuration $(\bar{c}, v)$ of $\mathcal{LS}$ and a message $m \in \Pi$, there is a message synchronization step $(\bar{c}, v) \xrightarrow{m} (\bar{c}', v')$ in $\mathcal{LS}$ if:

- A maximal set of m-labeled message occurrences are enabled at $(\bar{c}, v)$, and there is no chart $\mathcal{L}_i$ whose local configuration $((\bar{c})_i, v)$ will be hot-violated by an m-labeled message. In this case, for all charts $\mathcal{L}_j$ which each has an m-labeled message occurrence enabled at $(\bar{c}, v)$, the $\xrightarrow{m}$ message synchronization steps will occur simultaneously; and

there is a silent step $(\bar{c}, v) \xrightarrow{\tau} (\bar{c}', v)$ in $\mathcal{LS}$ if:

- There is a chart $\mathcal{L}_i$ such that $((\bar{c})_i, v) \xrightarrow{\tau} ((\bar{c}')_i, v)$. In this case, for all $j \neq i$, we have $\bar{c}'_j = \bar{c}_j$; and there is a time delay step $(\bar{c}, v) \xrightarrow{d} (\bar{c}, v + d)$ in $\mathcal{LS}$ if:

- For all $1 \leq i \leq n$, we have $((\bar{c})_i, v) \xrightarrow{d} ((\bar{c})_i, v + d)$. □

In the first case above, the *global condition* for all m-labeled message occurrences is the conjunction of all individual conditions, and the *global assignment* is the union of all individual assignments.

Similarly, we can define runs and $\rightarrow^*$ for a set of time-enriched LSC charts.

Definition 6 (satisfaction of an LSC system). *A timed trace* $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$ satisfies *(passes) an LSC system* $\mathcal{LS}$ *iff,* γ *corresponds to an infinite run of* $\mathcal{LS}$*, and it satisfies each chart* $\mathcal{L}_i$ *in* $\mathcal{LS}$ *separately.* □

3 LSC to TA translation

3.1 Motivation

Similar to Paper A, in this paper our scenario-based analysis and synthesis methods rely on a translation of the LSC charts to timed automata. However, unlike in Paper A where each monitored chart specifies a user requirement individually, in this paper a set of driving charts are supposed to characterize the inter-object behaviors of the system *collectively.* When the system consists of a large number of driving charts, then the cut-based LSC-to-TA translation will encounter the state explosion problem: the number of possible global cuts (i.e., the number of possible system states) will increase rapidly, and explicit encoding and storing these information need a lot of space. Furthermore, the outcome of the translation as a single huge timed automaton will be difficult to visualize, to debug and to diagnose.

To overcome the above problems, in this paper we propose a different method for translating LSC charts to timed automata. For each driving LSC chart $\mathcal{L}$ in the system model, we view the instance lines in $\mathcal{L}$ as a set of parallelly running processes that communicate with one another and collaborate to achieve a common goal as specified by chart $\mathcal{L}$. Since Uppaal also operates on a network of parallelly composed processes (TAs) that communicate with each other, this motivates us to translate each instance line of $\mathcal{L}$ to a timed automaton. In this way we avoid the explicit construction of a global automaton. This idea in spirits resembles the approaches of [HKMP02, SD05a]. Thanks to the Uppaal features of broadcast channels, boolean and integer variables and committed locations in timed automata, we are able to appropriately translate the LSC features such as message sending, intra/inter-chart coordinations and cold/hot violations to timed automata. Compared with the "one-TA-per-chart" approach that can be viewed as a kind of *centralized* translation (Paper A, Section 3.3), the "one-TA-per-instance line" approach of this section can be viewed as a kind of *distributed* translation.

3.2 Mapping LSC instance to Uppaal timed automaton

Basic structure mapping

Each instance line I_i in chart $\mathcal{L}_u$ of the LSC system $\mathcal{LS}$ is mapped into a timed automaton $A_{u,i}$, where each position on I_i corresponds to a TA location in $A_{u,i}$, and each discrete advancement step (i.e., a message synchronization step or a silent step) on I_i corresponds to a TA edge in $A_{u,i}$. The sending (resp. receiving) of an m-labeled message on I_i corresponds to an $m!$ (resp. $m?$)-labeled TA edge in $A_{u,i}$.

Handling intra/inter-chart coordinations

In the prechart (resp. main chart) of an LSC chart, once all the participating instance lines have progressed to their *Pch_bot* (resp. *Mch_bot*) positions, then the prechart (resp. main chart) is successfully matched. In this case, the prechart (resp. main chart) will be exited immediately, and the main chart (resp. prechart) will be entered immediately afterwards. To synchronize all the participating instance lines for such a prechart/main chart (resp. main chart/prechart) transfer, for each LSC chart $\mathcal{L}$, we create a dedicated (auxiliary) coordinator automaton $Coord_{\mathcal{L}}$. This automaton will communicate with the timed automata for the instance lines by using auxiliary binary synchronization channels such that it can bookkeep how many instance lines are done with their prechart (resp. main chart) portions. Once the coordinator automaton realizes that the prechart (resp. main chart) has been successfully matched, it will immediately launch a broadcast synchronization with the timed automata for all the relevant instance lines.

In scenario-based modeling, the same message may well appear in two or more charts. To be more specific, it is possible that an m-labeled message from instance line A to instance line B has its occurrences in LSC charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$. If these messages are all enabled and one of them is chosen to be fired, then all the others must also be fired simultaneously. Clearly, this requires an inter-chart coordination. To achieve this, we use broadcast synchronization channels rather than binary synchronization channels for these messages. In the translated timed automata, if there is an $m!$-labeled edge from one to another TA locations, then we add an $m?$-labeled edge between these two TA locations to "accompany" the $m!$-labeled edge.

Handling cold and hot violations

Once a cold violation occurs, we let the timed automaton that corresponds to the message sending instance line report to the coordinator automaton in charge, which in turn immediately synchronizes all the timed automata that correspond to the relevant instance lines in the chart for a reset (i.e., to go back to their initial TA locations).

In the translated network of timed automata *NTA*, we maintain a global flag boolean variable *hotviolated*, which indicates whether a hot violation has occurred in the LSC system. This variable is initialized to `false`, and it will be set to `true` whenever a hot violation occurs.

Dealing with time

To mimic the behaviors of a clock constraint and clock reset in an LSC chart, we use a linked sequence of TA edges, whose atomicity is ensured by the Uppaal feature of committed location[4]. Upper bound constraints in the conditions can be extracted and used as TA location invariants to ensure that the constrained messages are sent out within the specified time frames; lower bound and clock difference constraints can be extracted and tested immediately after the message synchronizations.

Translating environment processes

The processes (instance lines) in an LSC system can be partitioned into two sets: the environment processes ($\mathcal{E}nv$) and the system processes ($\mathcal{S}ys$). From the system's perspective, the messages sent from $\mathcal{E}nv$ to $\mathcal{S}ys$ processes are uncontrollable, whereas other message sendings are controllable. To model this edge controllability for the purpose of timed game solving, we mark the message-sending edges in the translated timed automata of the $\mathcal{E}nv$ processes as *uncontrollable* edges (in dashed lines), and other edges as *controllable* edges (in solid lines). In this way, we obtain the timed game automata [MPS95] models.

Translating monitored chart

In comparison with a driving universal chart, the translation of a monitored chart to timed automata is different in the point that a monitored chart only "listens to" the messages in the LSC system and never emits messages by itself. When translating such a chart into a network of timed automata, if at position s of instance line I_k there is a sending of an m-labeled message, then we add an $m?$-labeled TA edge from l_{s-1} to l_s, and not an $m!$-labeled one.

The translation rules, the detailed explanations and the translation examples can be found in Appendix B.

[4]A *committed* location is an urgent location whose outgoing transitions have higher priority to be taken than those from non-committed ones.

3.3 Complexity of translated timed automata

Let $\mathcal{LS}$ be a set of LSC charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$, and let $NTA_{\mathcal{LS}}$ be the translated network of timed automata. Let $inst(\mathcal{L}_i)$, $ML(\mathcal{L}_i)$, $MA(\mathcal{L}_i)$ and $MO(\mathcal{L}_i)$ denote the set of instance lines, the set of message labels (i.e., "signals"), the message alphabet and the set of message occurrences of chart $\mathcal{L}_i$, respectively.

Table 1 summarizes the complexity of the outcomes of the translation in different settings, namely, a single LSC chart or an LSC system; untimed LSC chart or time-enriched LSC chart.

How we obtain these analysis results can be found in Appendix C.

Table 1: The complexity of the outcomes of LSC-to-TA translation

number of	A single chart $\mathcal{L}$	
	untimed chart	time-enriched chart
TAs	$\|inst(\mathcal{L})\| + 1$	$\|inst(\mathcal{L})\| + \|MA(\mathcal{L})\| + 1$
channels	$\|ML(\mathcal{L})\| + 2 \cdot \|inst(\mathcal{L})\| + 4$	$\|ML(\mathcal{L})\| + 2 \cdot \|inst(\mathcal{L})\| + 4 + 3 \cdot \|MA(\mathcal{L})\|$
auxiliary variables	$2 \cdot \|MA(\mathcal{L})\| + 2$	$4 \cdot \|MA(\mathcal{L})\| + 2 \cdot \|MO(\mathcal{L})\| + 2$
number of	A set of driving charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$	
	untimed charts	time-enriched charts
TAs	$\sum_{i=1}^{n}(\|inst(\mathcal{L}_i)\| + 1)$	$\sum_{i=1}^{n}(\|inst(\mathcal{L}_i)\| + 1) + \|\bigcup_{i=1}^{n} MA(\mathcal{L}_i)\|$
channels	$\|\bigcup_{i=1}^{n} ML(\mathcal{L}_i)\| + \sum_{i=1}^{n}(2 \cdot \|inst(\mathcal{L}_i)\| + 4)$	$\|\bigcup_{i=1}^{n} ML(\mathcal{L}_i)\| + \sum_{i=1}^{n}(2 \cdot \|inst(\mathcal{L}_i)\| + 4) + 3 \cdot \|\bigcup_{i=1}^{n} MA(\mathcal{L}_i)\|$
auxiliary variables	$2 \cdot \|\bigcup_{i=1}^{n} MA(\mathcal{L}_i)\| + 2n$	$4 \cdot \|\bigcup_{i=1}^{n} MA(\mathcal{L}_i)\| + 2 \cdot \sum_{i=1}^{n} \|MO(\mathcal{L}_i)\| + 2n$

3.4 Behavior equivalence of LSC and translated TAs

Theorem 1. *Let $\mathcal{LS}$ be a set of time-enriched LSC charts whose message alphabet is Π, and let $NTA_{\mathcal{LS}}$ be the translated network of timed automata which have a set Act of normal and auxiliary channels. Then $\forall \gamma_1 \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, ((\gamma_1 \models \mathcal{LS}) \Rightarrow \exists \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, (\gamma_2 \models NTA_{\mathcal{LS}}) \wedge (\gamma_2|_{(\Pi \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Pi \cup \mathbb{R}_{\geq 0})}))$, and $\forall \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, ((\gamma_2 \models NTA_{\mathcal{LS}}) \Rightarrow \exists! \gamma_1 \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega \,.\, (\gamma_1 \models \mathcal{LS}) \wedge (\gamma_2|_{(\Pi \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Pi \cup \mathbb{R}_{\geq 0})}))$.*

Theorem 1 indicates that each accepted timed trace γ_1 in $\mathcal{LS}$ uniquely corresponds to a cluster of accepted timed traces in $NTA_{\mathcal{LS}}$. All these traces project to exactly the same trace on the message alphabet and time delays $(\Pi \cup \mathbb{R}_{\geq 0})$ as γ_1 does.

The lemmas for theorems in Sections 3.4 and 4, and the proofs of them can be found in Appendix D.

4 Analysis and synthesis problems

4.1 Consistency checking

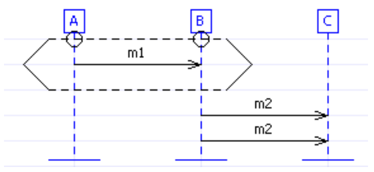
An LSC system is *inconsistent* if and only if the system model has internal contradictions [HK02], i.e., there does not exist an infinite message sequence such that it satisfies all driving universal LSC charts. Alternatively, an LSC system is inconsistent iff, along all possible paths there will eventually be a hot violation of the main chart of a certain LSC chart (i.e., the flag boolean variable *hotviolated*, which has been initialized to `false`, will eventually be set `true` in the translated network of timed automata).

UPPAAL uses a fragment of the CTL logic as its query language. Formulas could take the forms $\mathsf{E}\Diamond\phi$, $\mathsf{E}\Box\phi$, $\mathsf{A}\Diamond\phi$, $\mathsf{A}\Box\phi$, $\phi_1 \leadsto \phi_2$, where ϕ, ϕ_1 and ϕ_2 are state formulas. In particular $\phi_1 \leadsto \phi_2$ ("ϕ_1 leads-to ϕ_2") is a shorthand for $\mathsf{A}\Box(\phi_1 \Rightarrow \mathsf{A}\Diamond\phi_2)$, which characterizes the assume-guarantee style liveness (or responsiveness).

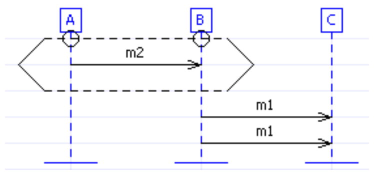
Theorem 2. *$\mathcal{LS} = \{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}$ are inconsistent $\Leftrightarrow$ $NTA_{\mathcal{LS}} \models$ `true` $\leadsto$ (hotviolated $==$ `true`).*

Theorem 2 indicates that in order to check the inconsistency of a set of driving LSC charts, we can instead check whether it is true that the translated network of timed automata will eventually have its boolean flag variable *hotviolated* set to `true`.

For example, by model checking the corresponding translated network of timed automata, we find out that the two driving universal LSC charts in Fig. 1 are consistent, whereas the two charts in Fig. 3 are inconsistent.



(a) chart 1



(b) chart 2

Figure 3: Two universal charts which are inconsistent.

4.2 Property verification

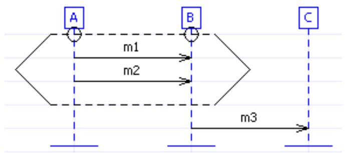
Property verification asks whether a system that is modeled as a set of driving universal LSC charts $\mathcal{LS}$ satisfies the requirement that is specified as a separate monitored universal or existential chart $\mathcal{L}'$. Here $\mathcal{L}'$ will be translated into a network of "observer" timed automata $NTA_{\mathcal{L}'}$, i.e., they only "listen to" the messages in the network of timed automata $NTA_{\mathcal{LS}}$ for $\mathcal{LS}$, and never emit messages by themselves.

Theorem 3. *Let $\mathcal{LS}$ be an LSC system, and $\mathcal{L}'$ be a monitored universal chart.* $\mathcal{LS} \models \mathcal{L}' \Leftrightarrow (NTA_{\mathcal{LS}} \,||\, NTA_{\mathcal{L}'}) \models (Coord_{\mathcal{L}'}.Mch\_top \leadsto Coord_{\mathcal{L}'}.Mch\_bot)$.

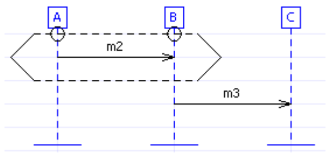
In the above theorem, $Coord_{\mathcal{L}'}.Mch\_top$ and $Coord_{\mathcal{L}'}.Mch\_bot$ say that the coordinator timed automaton $Coord_{\mathcal{L}'}$ for chart $\mathcal{L}'$ is in its locations Mch_top and Mch_bot, respectively.

Theorem 3 indicates that in order to check whether a system $\mathcal{LS}$ satisfies the requirement in a universal chart $\mathcal{L}'$, we only need to check whether the parallelly composed translated network of timed automata satisfy the aforementioned responsiveness property.

For example, after model checking the corresponding translated network of timed automata, we find out that the (single chart) LSC system in Fig. 4(a) satisfies the requirement that is specified in Fig. 4(b).



(a) chart 1



(b) chart 2

Figure 4: Chart 1 (model) satisfies chart 2 (property).

Theorem 4. *Let $\mathcal{LS}$ be an LSC system, and $\mathcal{L}'$ be a monitored existential chart.* $\mathcal{LS} \models \mathcal{L}' \Leftrightarrow (NTA_{\mathcal{LS}} \,||\, NTA_{\mathcal{L}'}) \models (\mathsf{E}\Diamond\ Coord_{\mathcal{L}'}.Mch\_bot)$.

Theorem 4 indicates that in order to check whether a system $\mathcal{LS}$ satisfies the requirement in an existential chart $\mathcal{L}'$, we only need to check whether $\mathcal{LS}$ have a trace that can be observed by $\mathcal{L}'$ as a satisfying run.

4.3 Centralized synthesis for open systems

A timed automaton with its edges partitioned into controllable and uncontrollable ones is called a *timed game automaton* (TGA) [MPS95]. A network of parallelly composed timed game automata for $\mathcal{E}nv$ and $\mathcal{S}ys$ can be viewed as a timed game structure: as a player, the timed game automata for $\mathcal{S}ys$ (noted $NTA_{\mathcal{S}ys}$) master the set A_c of controllable edges; as the opponent, the timed game automata for $\mathcal{E}nv$ (noted $NTA_{\mathcal{E}nv}$) master the set A_u of uncontrollable edges. Given a winning objective, $NTA_{\mathcal{S}ys}$ will take moves in order to win the game (i.e., to bring the system into a winning state, or to prevent the system from entering a losing state), whereas $NTA_{\mathcal{E}nv}$ may spoil the game.

Let S be the state space of $NTA_{\mathcal{E}nv} \,||\, NTA_{\mathcal{S}ys}$, and $\epsilon \notin (A_c \cup A_u \cup \{\tau\})$ be an empty action which means "do nothing at this moment in time". A state-based (or memoryless) strategy for $NTA_{\mathcal{S}ys}$ is a (partial) function

$$\rho : S \to (A_c \cup \{\epsilon\}),$$

which constantly guides the timed automata in $NTA_{\mathcal{S}ys}$ to take appropriate controllable actions, or just delay (and wait for the semantic state to be changed by an uncontrollable action of $NTA_{\mathcal{E}nv}$, or by the elapse of time).

Uppaal-Tiga [BCD+07] is a timed game solver. Its inputs include a set of timed game automata, and a winning objective that is formulated as an extended ACTL (the universal fragment of CTL) formula. For example, property "control: $\mathsf{A}\Box\ \varphi$" asks whether there exists a strategy ρ for $NTA_{\mathcal{S}ys}$ such that if $NTA_{\mathcal{S}ys}$ is supervised (or "restricted", "guided") by ρ, then the system $NTA_{\mathcal{LS}}$ is guaranteed to always (i.e. invariantly) satisfy φ. If the property is satisfied, then Uppaal-Tiga will be able to synthesize a winning strategy for $NTA_{\mathcal{S}ys}$.

Synthesis for $\mathcal{S}ys$ is possible only if the entire system $\mathcal{LS}$ can be guaranteed not to be hot-violated no matter how the $\mathcal{E}nv$ processes behave. By means of trace-wise behavior equivalence, this boils down to finding a winning strategy ρ for $NTA_{\mathcal{S}ys}$. Since the strategy (if ever exists) will oversee all $\mathcal{S}ys$ processes rather than being distributed to supervise each individual $\mathcal{S}ys$ process, it is a kind of *centralized* synthesis. It is clear that $NTA_{\mathcal{LS}}$ as supervised by ρ constitute one such desired executable (state/transition-based) object system.

Theorem 5. *An executable object system for $\mathcal{S}ys$ can be synthesized $\Leftrightarrow NTA_{\mathcal{LS}} \models$ (control: $\mathsf{A}\Box$ ($hotviolated == \texttt{false}$)).*

Theorem 5 indicates that the problem of centralized synthesis for open systems can be reduced to a timed game solving problem in Uppaal-Tiga.

5 Experiments

Fig. 5 shows our scenario-based analysis and synthesis framework. Among those inputs (the shaded elements), the $\mathcal{E}nv/\mathcal{S}ys$ partitioning directives specify which processes in the charts of $\mathcal{LS}$ belong to $\mathcal{E}nv$ and which belong to $\mathcal{S}ys$, respectively.

We build a GUI-based LSC editor, with which we can construct either universal or existential charts. A prototype command line LSC-to-TA translator has been implemented, which is capable of batch translation of monitored and driving charts. The translator-generated timed automata and CTL formulas comply with the Uppaal timed automaton, Uppaal-Tiga timed game automaton and their query language syntaxes, and can thus be fed into Uppaal and Uppaal-Tiga directly.

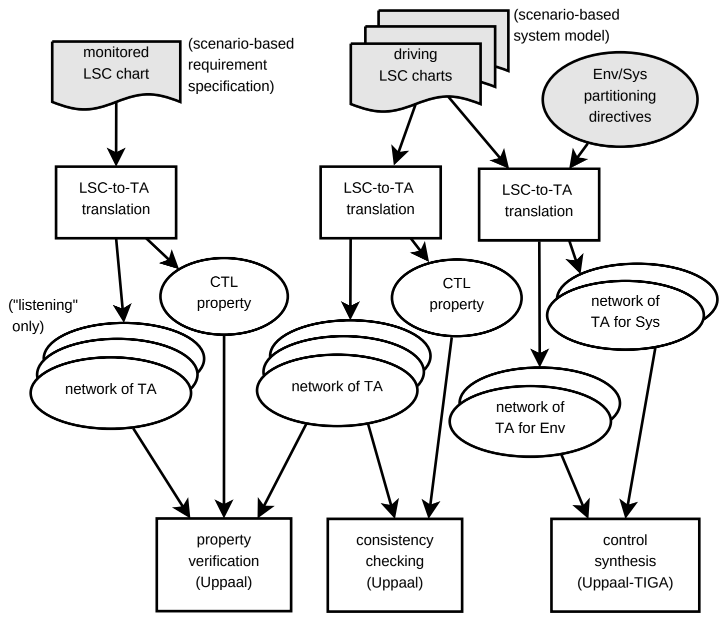


Figure 5: Scenario-based analysis and synthesis framework.

Preliminary translation experiments have been conducted on an Intelligent Mouse (# of charts, instance lines, message labels, message occurrences, clocks: 4, 3, 3, 12, 2), a refrigeration (4, 3, 3, 14, 1) and an ATM Machine (12, 3, 6, 21, 0) examples, and a DHCP (Dynamic Host Configuration Protocol) (34, 3, 17, 44, 0) case studies. Some comparative experiments reveal that the translation has negligible time overheads and memory consumptions than the subsequent model checking and game solving of the translated network of timed automata. This is reasonable, because the complexity of scenario-based analysis and synthesis mainly lies in the LSC models themselves. As a syntactical level manipulation, the translation only introduces some auxiliary channels and bookkeeping variables (not clock variables) to implement the LSC semantics.

6 Conclusions

We present timed extensions to a kernel subset of the LSC language and define a trace-based semantics. We show how to transform LSC charts into a network of behavior-equivalent timed automata. The LSC consistency checking, property verification and synthesis problems can be reduced to CTL real-time model checking and timed game solving problems. We implement a prototype LSC-to-TA translator. When linked with our LSC editor and the existing real-time model checker Uppaal and timed game solver Uppaal-Tiga, they constitute a tool chain for automated, scenario-based analysis and synthesis of real-time systems. Preliminary experiments on a number of examples show that it is a viable approach.

Scenario-based approaches enjoy the advantages of incremental construction of models, i.e., new pieces of scenarios can be added into existing ones during the development process. In order to keep the driving LSC charts and their complexity within human readable and manageable levels, a single LSC chart needs not to be very large and complex. Rather the complexity of scenario-based models mainly lies in the interplays of a large number of relatively simple charts. Our "one-TA-per-instance line" translation is in accordance with this philosophy. Instead of constructing a complex global state machine that handles all possible activities explicitly, we leave the intricate semantics of LSC chart progress and intra/inter-chart coordinations mostly up to UPPAAL.

As future work, we may consider the translation of more LSC constructs into timed automata, such as co-region, symbolic instance, control structures, etc. Other chart activation modes also need to be dealt with. The implementation of a full-fledged translator and the application of the tool chain to industrial case studies are desirable. Furthermore, scenario-based synthesis for systems with imperfect information (e.g., some uncontrollable actions are not observable) is also worth investigation.

Appendix A: Timed automata in Uppaal

We use the following notations: C is a set of real-valued clocks, and $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Definition 7 (timed automaton, TA [BDL04])**.** *A* timed automaton *is a tuple* (L, l_0, C, Act, E, Inv)*, where* L *is a set of locations,* $l_0 \in L$ *is the initial location,* C *is a set of clocks,* Act *is the alphabet of actions,* $E \subseteq L \times (Act \cup \{\tau\}) \times B(C) \times 2^C \times L$ *is a set of edges between locations, each of which has an action, a guard and a set of clocks to be reset, and* $Inv : L \to B(C)$ *assigns invariants to locations.* □

Uppaal has defined a number of extensions to the standard notations of timed automata. Specifically, an *urgent* location is such a TA location that freezes time, i.e., time is not allowed to elapse when a process is in an urgent location. A *committed* location is a special kind of urgent location whose outgoing transitions always have higher priority to be fired than those from non-committed locations.

Uppaal uses a mixture of handshake communication and broadcast communication. The CBS (Calculus of Broadcasting Systems [Pra95])-style broadcast channels allow 1-to-many synchronization. If the emitting edge is enabled, then it can always fire. If the emitting edge is fired, then all enabled receiving edges (might be 0 edge) will synchronize.

In Uppaal an *urgent* channel means that if it is possible to trigger a synchronization over that channel, then it cannot delay in the source state.

Furthermore, Uppaal also supports bounded-range integer and boolean data variables, which can be used in the guards, assignment and location invariants.

A clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative real numbers. Let ${\mathbb{R}_{\geq 0}}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. We will abuse the notation by considering guards and invariants as sets of clock valuations, writing $u \in Inv(l)$ to mean that valuation u satisfies $Inv(l)$.

Definition 8 (semantics of TA [BDL04])**.** *Let* (L, l_0, C, Act, E, Inv) *be a timed automaton. The* semantics *is defined as a labeled transition system* $\langle S, s_0, \to \rangle$*, where* $S \subseteq L \times {\mathbb{R}_{\geq 0}}^C$ *is the set of states,* $s_0 = (l_0, u_0)$ *the initial state, and* $\to \subseteq S \times (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \times S$ *the transition relation such that:*

- $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \,.\, u + d' \in Inv(l)$*; and*
- $(l, u) \xrightarrow{a} (l', u')$ *if there exists* $e = (l, a, g, r, l') \in E$ *such that* $u \in g$*,* $u' = [r \to 0]u$*, and* $u' \in Inv(l')$*,*

where for $d \in \mathbb{R}_{\geq 0}$*,* $u + d$ *maps each clock* x *in* C *to the value* $u(x) + d$*, and* $[r \to 0]u$ *denotes the clock valuation which maps each clock in* r *to 0 and agrees with* u *over* $C \setminus r$*.* □

Definition 9 (run of TA)**.** *A* run *of a TA* $(L, l_0, C, \mathit{Act}, E, \mathit{Inv})$ *is a sequence of states* $s^0 \cdot s^1 \cdot \ldots$ *that are connected by the transitions, i.e.,* $\forall i \geq 0 \,.\, \exists u_i \in (\mathit{Act} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \,.\, s^i \xrightarrow{u_i} s^{i+1}$. □

The transition relation $\rightarrow$ as mentioned above each time consumes only a single letter $u \in (\mathit{Act} \cup \{\tau\} \cup \mathbb{R}_{\geq 0})$. We extend it to $\rightarrow^*$ such that it consumes a (finite or infinite) word $w \in (\mathit{Act} \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^* \cup (\mathit{Act} \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega$. A word w that corresponds to a run of the TA is called a *timed trace* of the TA.

A number of timed automata can be parallelly composed into a network of timed automata over a common set of clocks and actions, $A_i = (L_i, l_{0,i}, C, \mathit{Act}, E_i, \mathit{Inv}_i)$, $1 \leq i \leq n$. A location vector $\bar{l} = (l_1, \ldots, l_n)$ is a vector of locations of the member TA. We compose the invariant functions into a common function over location vectors $\mathit{Inv}(\bar{l}) = \bigwedge_i \mathit{Inv}_i(l_i)$. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the i-th element l_i of $\bar{l}$ is replaced by l'_i.

Definition 10 (semantics of a network of TAs [BDL04])**.** *Let* $A_i = (L_i, l_{0,i}, C, \mathit{Act}, E_i, \mathit{Inv}_i)$ *be a network of timed automata,* $1 \leq i \leq n$*. Let* $\bar{l_0} = (l_{0,1}, \ldots, l_{0,n})$ *be the initial location vector. The* semantics *is defined as a transition system* $\langle S, s_0, \rightarrow \rangle$*, where* $S = (L_1 \times \ldots \times L_n) \times \mathbb{R}_{\geq 0}{}^C$ *is the set of global states,* $s_0 = (\bar{l_0}, u_0)$ *the initial global state, and* $\rightarrow \subseteq S \times (\mathit{Act} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \times S$ *the transition relation defined by:*

- $(\bar{l}, u) \xrightarrow{d} (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \,.\, u + d' \in \mathit{Inv}(\bar{l})$*;*

- $(\bar{l}, u) \xrightarrow{\tau} (\bar{l}[l'_i/l_i], u')$ *if there exists* $l_i \xrightarrow{\tau, g, r} l'_i$ *such that* $u \in g$*,* $u' = [r \rightarrow 0]u$ *and* $u' \in \mathit{Inv}(\bar{l}[l'_i/l_i])$*;*

- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_i/l_i, l'_j/l_j], u')$ *if* a *is a binary channel and there exist* $l_i \xrightarrow{c!, g_i, r_i} l'_i$ *and* $l_j \xrightarrow{c?, g_j, r_j} l'_j$ *such that* $u \in (g_i \wedge g_j)$*,* $u' = [r_i \cup r_j \rightarrow 0]u$ *and* $u' \in \mathit{Inv}(\bar{l}[l'_i/l_i, l'_j/l_j])$*; and*

- $(\bar{l}, u) \xrightarrow{a} (\bar{l}[l'_i/l_i, l'_j/l_j, l'_k/l_k, \ldots], u')$ *if* a *is a broadcast channel and there exist an* $l_i \xrightarrow{c!, g_i, r_i} l'_i$ *and a maximal set* $\{j, k, \ldots\}$*:* $l_j \xrightarrow{c?, g_j, r_j} l'_j$*,* $l_k \xrightarrow{c?, g_k, r_k} l'_k$*,* $\ldots$*, such that* $u \in (g_i \wedge g_j \wedge g_k \wedge \ldots)$*,* $u' = [r_i \cup r_j \cup r_k \cup \ldots \rightarrow 0]u$ *and* $u' \in \mathit{Inv}(\bar{l}[l'_i/l_i, l'_j/l_j, l'_k/l_k, \ldots])$*.* □

Runs and traces of a network of TAs are defined similarly as those for a single TA.

Appendix B: Rules for LSC-to-TA translation

B1. Translating message-only charts

As mentioned in Section 2.1, along each instance line I_i in chart $\mathcal{L}$, there are a set $pos(\mathcal{L}, I_i)$ of positions, among which there are a set $StdPos(\mathcal{L}, I_i) \subset pos(\mathcal{L}, I_i)$ of four "standard" positions. For example in instance line A of Fig. 1(a), there are 7 positions (black filled circles), where the four standard ones are $Pch\_top\,(0)$, $Pch\_bot\,(3)$, $Mch\_top\,(4)$ and $Mch\_bot\,(6)$.

Given $I_i \in inst(\mathcal{L})$, $p \in StdPos(\mathcal{L}, I_i)$, we use $\mathcal{L}.I_i.p$ to denote the position ID of standard position p on instance line I_i of chart $\mathcal{L}$. For example in Fig. 1(a), we have $\mathcal{L}.A.Pch\_bot = 3$.

Fig. 6 shows the translated network of timed automata for the chart $\mathcal{L}_1$ of Fig. 1(a).

(1) Basic mapping rules

Let $\mathcal{LS}$ be an LSC system, $\mathcal{L}_u$ be a chart in $\mathcal{LS}$, and I_i be an instance line in $\mathcal{L}_u$. We map each such I_i to a timed automaton $A_{u,i}$ using the following rules:

R1 Each position k on I_i of $\mathcal{L}_u$ corresponds to a TA location l_k in $A_{u,i}$, $0 \leq k \leq p\_max_{\mathcal{L}_u, I_i}$. See Fig. 6(a), locations l_0 - l_6.

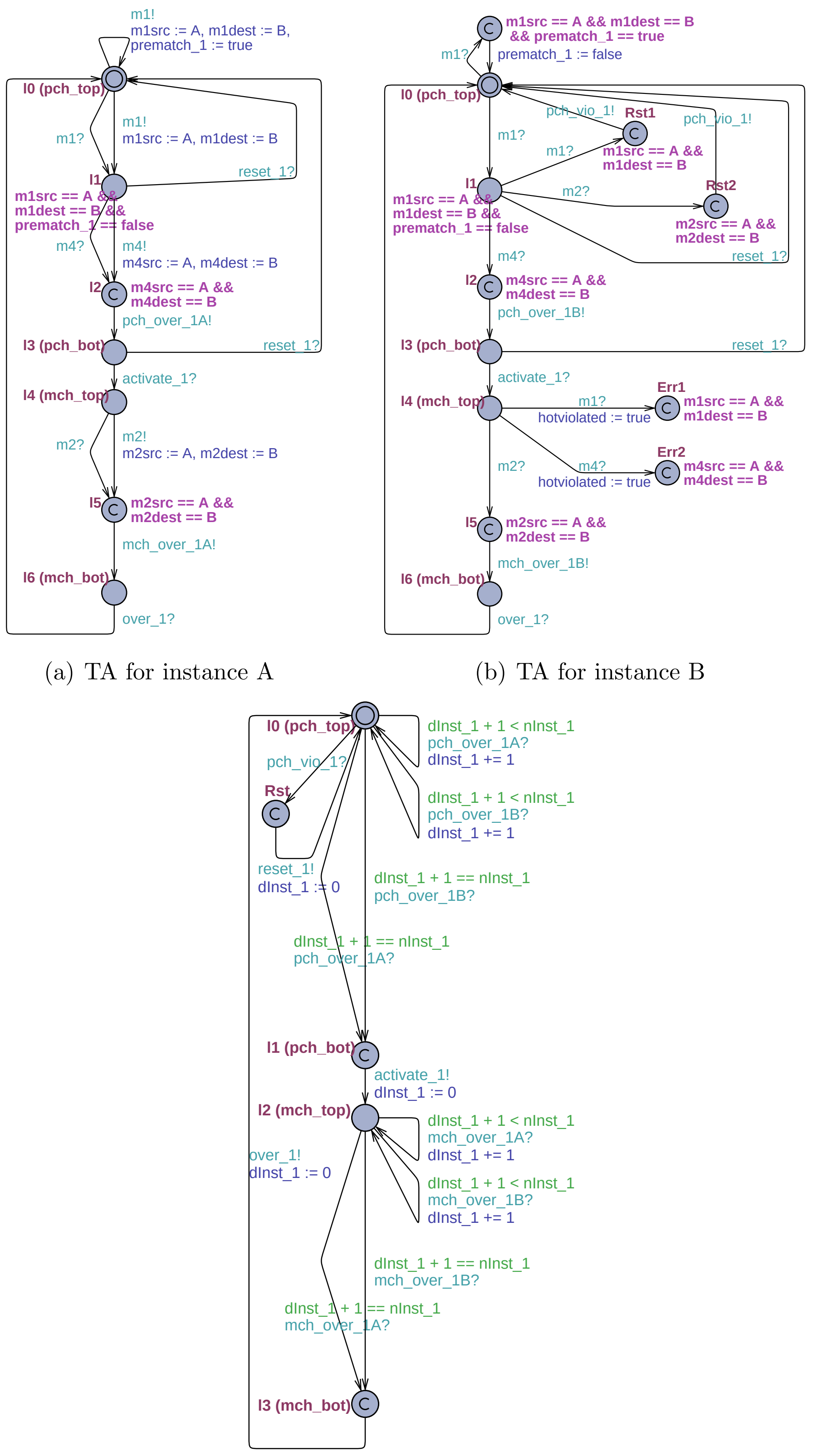
R2 If at position k on I_i of $\mathcal{L}_u$ there is a sending of an m-labeled message to instance I_j, then there will be assigned an $m!$-labeled TA edge from location l_{k-1} to l_k in $A_{u,i}$. See Fig. 6(a), straight line edges $(l_0,\, l_1), (l_1,\, l_2), (l_4,\, l_5)$.

R3 If at position k on I_i of $\mathcal{L}_u$ there is a reception of an m-labeled message from instance I_j, then there will be assigned an $m?$-labeled TA edge from location l_{k-1} to l_k in $A_{u,i}$. See Fig. 6(b), straight line edges $(l_0,\, l_1), (l_1,\, l_2), (l_4,\, l_5)$.

We abuse the notations Pch_top, Pch_bot, Mch_top and Mch_bot to also denote the TA locations that correspond to these LSC positions. For any position k other than the aforementioned four, it corresponds to a TA location l_k, meaning that upon sending/receiving the message that anchors at position k, we now arrive at l_k. Furthermore, position 0 (i.e., Pch_top) corresponds to the initial TA location l_0 (i.e., Pch_top).

When R2 is applied, the TA edge can be associated with an assignment "$m\_src := I_i,\ m\_dest := I_j$", where m_src and m_dest are fresh auxiliary (bounded integer) variables, meaning that an m-labeled message is sent from instance I_i to I_j in chart $\mathcal{L}_u$. In R2 and R3, the destination location l_k will have invariant "$(m\_src == I_i) \wedge (m\_dest == I_j)$" and "$(m\_src == I_j) \wedge (m\_dest == I_i)$", respectively. See Fig. 6(a), locations l_1, l_2, l_5, and Fig. 6(b), l_1, l_2, l_5.

(2) Handling intra-chart coordinations



(a) TA for instance A

(b) TA for instance B

(c) The coordinator TA

Figure 6: Translated TAs for the untimed chart $\mathcal{L}_1$ of Fig. 1(a)

In an LSC chart, if an instance line (process) in its prechart portion has no more interactions with the other instance lines (e.g., it has successfully sent/received the last message, or it has no interactions with other instance lines at all), then it will *immediately* progress to the bottom position *Pch_bot* of its prechart portion, to be ready for a next mandatory synchronization that involves all the instance lines in that chart.

R4 At position k on the prechart portion of I_i of $\mathcal{L}_u$, if $k = \mathcal{L}_u.I_i.Pch\_bot - 1$, then we mark l_k as a committed location in $A_{u,i}$, and we add a $pch\_over_{u,i}!$-labeled edge from l_k to l_{k+1} in $A_{u,i}$. See Fig. 6(a), location l_2.

The auxiliary channel $pch\_over_{u,i}$ (meaning “prechart portion is over”) is used to notify the coordinator automaton $Coord_u$ (explained below) of the completion of instance line I_i with its prechart portion in chart $\mathcal{L}_u$.

When all the instance lines in chart $\mathcal{L}_u$ progress to their respective *Pch_bot* positions, the prechart is now successfully matched. Once this happens, all these instance lines must immediately synchronize and progress to their respective *Mch_top* positions, meaning that the main chart is now activated. To model this kind of *intra-chart* coordination at the prechart/main chart interface, for each chart $\mathcal{L}_u$, we create a dedicated (auxiliary) coordinator automaton $Coord_u$. This automaton will communicate with the automata that correspond to the instance lines of $\mathcal{L}_u$ by using auxiliary binary channels such that it can bookkeep how many instance lines are done with their prechart portions. Once the coordinator automaton realizes that the prechart has been successfully matched, it will immediately launch a broadcast synchronization with the automata that correspond to the instance lines.

Fig. 6(c) gives an example of the coordinator timed automaton for chart $\mathcal{L}_1$ of Fig. 1(a), where $pch\_over_{1,A}$ and $pch\_over_{1,B}$ are binary channels, $activate_1$ is a broadcast channel (meaning that the main chart is to be activated), $nInst_1$ is a constant that denotes the $\underline{n}$umber of instance lines that participate in chart $\mathcal{L}_1$, and $dInst_1$ is an integer variable that denotes the number of instance lines that are $\underline{d}$one with their prechart (or main chart) portions of $\mathcal{L}_1$.

The coordinator TA synchronizes with the timed automata that correspond to the instance lines in the prechart/main chart according to the following rule:

R5 At position k of I_i of $\mathcal{L}_u$, if $k = \mathcal{L}_u.I_i.Pch\_bot$, then there will be assigned an $activate_u?$-labeled TA edge from l_k to l_{k+1} in $A_{u,i}$. See Fig. 6(a), l_3, and Fig. 6(b), l_3.

Similarly, intra-chart coordination upon main chart completion will correspond to the channels $mch\_over_{u,i}$ and $over_u$ (meaning that the main chart has been successfully matched).

(3) Handling inter-chart coordinations

In scenario-based modeling, the same message may well appear in two or more charts. For example given an LSC system $\mathcal{LS}$, in chart $\mathcal{L}_1$ there is an m-labeled message occurrence mo_1 from instance I_1 to I_2, and in chart $\mathcal{L}_2$ there is an m-labeled message occurrence mo_2, also from I_1 to I_2. If at a certain global configuration $(\bar{c}, v)$ these message occurrences (in the above example mo_1 and mo_2) are all enabled, then their firings should be synchronized, i.e., either all of them are chosen to be fired, or none of them is chosen. This is considered a kind of *inter-chart* coordination.

In the translated network of timed automata, this can be accomplished by using a broadcast synchronization. Recall that in a broadcast synchronization, there is only one sender. Therefore, when translating the message occurrences (in the above example mo_1 and mo_2) to edges in their respective timed automata, only one of the LSC positions that are associated with the message tails (i.e., sending locations) in $\mathcal{LS}$ can correspond to the TA location that has the sole outgoing message-emitting TA edge in the translated TAs, and all others will correspond to TA locations that have outgoing message-receiving TA edges. Since all message-sending instance lines in the relevant charts should have the equal possibility to initiate the message synchronization, we consider a universal and symmetric solution: for each $m!$-labeled edge from one to another TA locations, we add an $m?$-labeled edge between these two TA locations to "accompany" the $m!$-labeled edge. In other words, we let all translated TA locations that correspond to the message-sending locations (in the above example two locations in the translated TAs $A_{1,1}$ and $A_{2,1}$) have the equal chance to act also as the broadcast synchronization initiator.

R6 If at position k on I_i of $\mathcal{L}_u$ there is a sending of an m-labeled message, then there will be added an $m?$-labeled TA edge from l_{k-1} to l_k in $A_{u,i}$. In the translated TAs, m will be changed from a binary to a broadcast channel. See Fig. 6(a), polyline edges (l_0, l_1), (l_1, l_2), (l_4, l_5).

(4) Handling cold and hot violations

Along an instance line of an LSC chart, if an arriving message is not enabled at the current cut in the prechart, then there will be a cold violation. In this case, all participating instance lines in this chart should be reset (i.e., brought back to their initial positions) immediately. In our translation, this is implemented by letting the timed automaton that corresponds to the message receiving instance line "report" the cold violation to the coordinator automaton in charge, which in turn immediately initiates a broadcast synchronization to ask the timed automata that correspond to all other instance lines of the chart to do a reset.

R7 Assume that at position k on the prechart portion of instance line I_i of chart $\mathcal{L}_u$, there is a reception of an m-labeled message from instance I_j. If $k \geq \mathcal{L}_u.I_i.Pch\_top + 2$, then for all m'-labeled message in $\mathcal{L}_u$ such that $m' \neq$

m (note that $m, m' \in \Pi$), there will be added first an m'?-labeled outgoing TA edge from l_{k-1}, then a fresh intermediate committed TA location with invariant $m'\_src == src(m') \wedge m'\_dest == dest(m')$, and then a pch_vio_u!-labeled TA edge that leads to l_0 in $A_{u,i}$. See Fig. 6(b), TA location `Rst1`, and TA edges (l_1, `Rst1`), (`Rst1`, l_0).

In the above rule, the auxiliary binary channel pch_vio_u (meaning "prechart violation" of chart $\mathcal{L}_u$) is used to notify the coordinator TA $Coord_u$ of the cold violation. The $reset_u$!-labeled broadcast edge will be added in $Coord_u$. See Fig. 6(c), TA edges (l_0, `Rst`), (`Rst`, l_0). In the prechart of $\mathcal{L}_u$, for all positions s on all instance lines I_t such that $\mathcal{L}_u.I_t.Pch\_top + 1 \leq s \leq \mathcal{L}_u.I_t.Pch\_bot$, we add a $reset_u$?-labeled edge from l_s to l_0 in $A_{u,t}$. See Fig. 6(a), TA edges (l_1, l_0), (l_3, l_0).

If a message violates the partial order in the main chart, then it is a hot violation. Once this happens, the corresponding TA will immediately go to a deadend error location (`Err`).

R8 If at position k on the main chart portion of instance line I_i of chart $\mathcal{L}_u$, there is a reception of an m-labeled message from instance I_j, then for all m'-labeled message in $\mathcal{L}_u$ such that $m' \neq m$, there will be added to location l_{k-1} an m'?-labeled outgoing TA edge, which arrives at a deadend error location. See Fig. 6(b), locations `Err1`, `Err2` and edges (l_4, `Err1`), (l_4, `Err2`).

(5) Prechart pre-matching

According to the semantics for invariant mode LSC chart, minimal events in the prechart are constantly being matched for. For example in Fig. 1(a), $m_1{\cdot}m_1{\cdot}m_4{\cdot}m_2$ is a matching sequence for the second incarnation of chart $\mathcal{L}_1$ under the invariant mode.

R9 If at position 1 on the prechart portion of instance line I_i of chart $\mathcal{L}_u$, there is a sending of an m-labeled message to instance line I_j at its position 1, then there will be added to location l_0 of $A_{u,i}$ an m!-labeled self loop edge with assignment "$m\_src := I_i, m\_dest := I_j, prematch_u :=$ `true`". If location l_0 in $A_{u,i}$ has an invariant, then it will be enhanced with a further constraint "$prematch_u ==$ `false`". See Fig. 6(a), location l_0.

Similarly, if there is a reception of an m-labeled message from I_j, then we add to l_0 an m?-labeled edge, followed by an intermediate committed location which has invariant "$(m\_src == I_j) \wedge (m\_dest == I_i) \wedge (prematch_u ==$ `true`$)$", and then an internal transition edge with assignment "$prematch_u :=$ `false`" leading back to l_0. See Fig. 6(b).

The flag boolean variable $prematch_u$ is initialized to `false`. Once it is set `true`, it means that chart $\mathcal{L}_u$ is currently undergoing a process of prechart pre-matching.

For simplicity, the semantics of prechart pre-matching has not been considered in Section 2.1. A remedy to this is to add one more bullet to the “silent step” case, stating that an m-consuming advancement step will just remain at the top cut $\top$.

B2. Dealing with time

For time-enriched LSCs, there are further constructs (i.e., clock constraints and clock resets) to be considered during the translation. To mimic the behaviors of each clock constraint and clock reset in an LSC chart, we use a linked sequence of edges in the corresponding time automaton. The atomicity of executing this sequence is ensured by the Uppaal feature of committed location.

(1) Translation of guards (clock constraints)

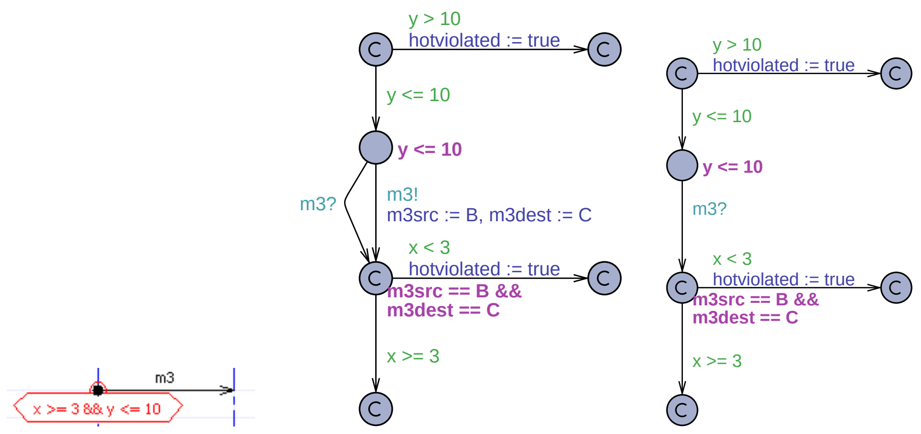
If an instance line has an m-labeled message sending that is guarded by a clock constraint, then a natural idea is to put this constraint on the $m!$-labeled edge of the translated TA. While this is feasible in the “one-TA-per-chart” translation method, it does not work in the “one-TA-per-instance line” method of this section. The reason is that we need to use broadcast channel m to handle the inter-chart coordination (see Section 6); however, due to the restriction of Uppaal, broadcast channels cannot carry clock constraints [BDL04]. To overcome this problem, in the translated TA, the upper bound constraint (if any) such as $x \leq 5$ will be tested prior to the message sending, and the lower bound and/or clock difference constraints (if any) such as $x \geq 3$ and $x - y \leq 2$ will be tested immediately after the message sending.

R10 If at position k on the main chart portion of instance line I_i of chart $\mathcal{L}_u$ there is a sending of an m-labeled message which is guarded by a clock constraint (see Fig. 7(a)), then in $A_{u,i}$ there will be first an intermediate committed location for upper bound constraint test. If true, then the next will be a normal location l_k with the upper bound constraint as the location invariant, which will in turn be immediately followed by a message sending edge. Finally, there will be another intermediate committed location for lower bound or clock difference constraint test. See Fig. 7(b).

For the receiving position of the guarded message, the translation is similar, see Fig. 7(c).

(2) Translation of assignments (clock resets)

In a time-enriched LSC chart, an assignment (i.e., clock resets) should take place immediately after the synchronization of the message occurrence that it is attached to. But in the translated TA, it cannot be put on the very edge that corresponds to the message sending/receiving, because clock resets should



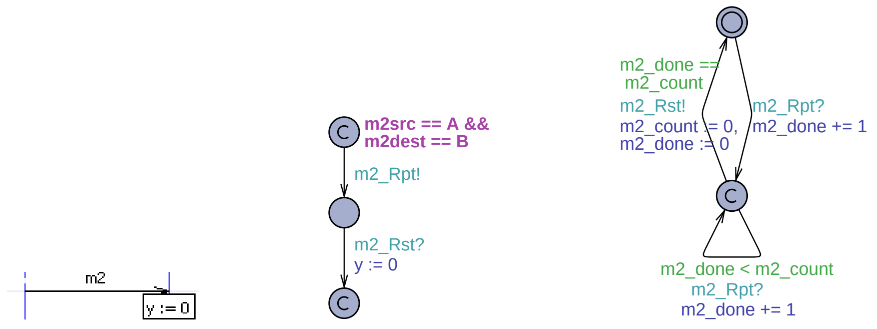
(a) a guarded message (b) TA for the sending instance line (c) TA for the receiving instance line

Figure 7: Translating a guarded message to TA fragments

not occur before the lower bound or clock difference constraint test which is supposed to happen immediately *after* the message synchronization. Neither can we append the TA edge that carries the assignment to the destination locations of the lower bound or clock difference constraint test, because if several identically-labeled message occurrences are simultaneously enabled in their respective charts where those charts have different guards and/or assignments for those message occurrences (see Fig. 2, the m_3-labeled message occurrences), then there could be racing conditions (e.g., the assignment $x := 0$ that is attached to m_3 in Fig. 2(a) should happen before the lower bound test $x \geq 3$ in Fig. 2(b); however, we are unable to guarantee this).

To model the clock resets properly, for each message $m \in \Pi$ in an LSC system, we use a dedicated process (TA) A_m to coordinate the clock resets of the corresponding message occurrences that are engaged in the same broadcast synchronization on m. When the broadcast synchronization happens, we use an integer variable m_count to bookkeep how many instance lines have participated in this broadcast synchronization. Whenever one of these instance lines is done with its lower bound constraint test (if any), it will immediately notify A_m of its completion using a binary channel m_Rpt ("reporting" to A_m), and after that it will wait for a synchronization on the broadcast channel m_Rst ("resetting clocks" command from A_m), along with which it can carry out its clock resets. In A_m, an integer variable m_done is increased by 1 each time when A_m is notified by an instance line (via $m\_Rpt?$). Once m_done rises up to m_count, A_m will immediately initiate the broadcast synchronization (via $m\_Rst!$).

R11 If at position k on instance line I_i of chart $\mathcal{L}_u$ there is a reception of an m-labeled message which has clock resets (see Fig. 8(a), instance line on the right), then there will be first an $m\_Rpt!$-labeled outgoing edge from location l_k in $A_{u,i}$, then a normal location, and then an m_Rst?-labeled outgoing TA edge that carries the clock resets. See Fig. 8(b).



(a) a message with clock reset

(b) TA for the receiver

(c) the dedicated TA A_{m_2}

Figure 8: Translating a message with clock reset to TAs

The dedicated TA A_m just waits for all the relevant instance lines to be done with their lower bound constraint tests, and then synchronizes them for clock resets. See Fig. 8(c).

(3) Just-in-Time message upper bound constraint test

When time-enriched LSC charts have upper-bound clock constraints, there are conditional tests before message sending/receiving in the translated timed automata. Given an m-labeled message occurrence in a chart, a potential problem is that in the translated timed automata for the sending and receiving instance lines, the TA locations that correspond to the sending and the receiving positions of this message may not be ready for this message synchronization at the same time (see Fig. 2(b), the m_1-labeled message occurrence). In the symbolic exploration of the state space of the translated network of timed automata, problems will arise if the upper bound of some message sending/receiving is tested when actually it *should not*. For example in Fig. 2, assume that a message sequence $m_1 \cdot m_2$ has been observed, and both charts have just entered their main charts, respectively. Note that a next m_1 is not enabled at the current cut (w.r.t. $\preccurlyeq$). But its guard will incorrectly add further constraint "$x \leq 2$" to "$(x \leq 5) \wedge (x \geq 3 \wedge y \leq 10)$". Consequently, according to our translation method mentioned earlier in this section, all possible paths will end up with hot violations.

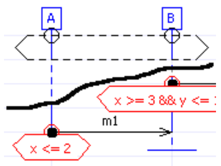
To avoid this kind of premature tests of upper bound constraints for message occurrences, we associate each message occurrence mo in each chart $\mathcal{L}_u$ with two flag boolean variables mo_u_maySnd and mo_u_mayRcv, denoting whether this message may be sent or received in chart $\mathcal{L}_u$, respectively. The upper bound constraint of mo can be tested only if both flag variables evaluate to `true`.

R12 If at position k on instance line I_i of chart $\mathcal{L}_u$ there is a sending of message occurrence mo which has a clock constraint (see Fig. 7(a)), then there will be a preceding edge carrying the predicate "$mo\_u\_mayRcv ==$ `true`". Once this message synchronization is fired, mo_u_maySnd will be cleared.

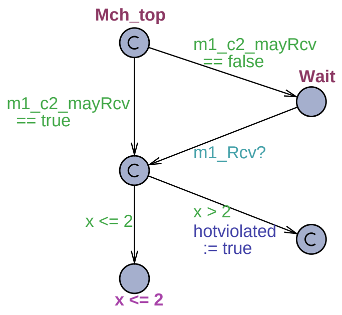
For the receiving instance line of message occurrence mo, the corresponding predicate will be "$mo\_u\_maySnd ==$ `true`".

If mo is a minimal event in the prechart (resp. main chart), then mo_u_maySnd and mo_u_mayRcv will have initial values `true` (resp. will be set to `true` by the $activate_u$ synchronization). If mo is not a minimal event, then the flag variables will be set to `true` by its predecessor event.

Given a message mo, if the predecessor positions of the head/tail positions of mo are also the head/tail (or tail/head) positions of another message occurrence, or mo is a minimal event, then mo_u_maySnd and mo_u_mayRcv will be both `true` prior to the constraint tests. Otherwise, their truth values may differ, e.g. in Fig. 9(a), message occurrence m_1 for the current cut (the solid free line). In this case, the translated TA $A_{2,A}$ will go to location `Wait` to "sleep", and will then be woken up by a dedicated message mo_1_Rcv that is sent by the TA $A_{2,B}$. See Fig. 9(b).



(a) an LSC fragment



(b) a fragment of the TA for instance line A

Figure 9: An LSC fragment in Fig. 2(b) and the corresponding TA fragment for its instance line A

Appendix C: Complexity of the outcomes of translation

For a time-enriched LSC system, we analyze the complexity of the translated network of timed automata as follows:

Let the set $\mathcal{LS}$ of time-enriched charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$ have messages $m_1, m_2, \ldots, m_k$, message occurrences $mo_1, mo_2, \ldots, mo_s$, and instance lines $I_{i,1}, I_{i,2}, \ldots, I_{i,in_i}$, where $1 \leq i \leq n$, $in_i = \#(\mathit{inst}(\mathcal{L}_i)) = |\mathit{inst}(\mathcal{L}_i)|$.

According to Section 6 ("Handling intra-chart coordinations") and Section 6 ("Translation of assignments"), the translated network of timed automata will be $\mathit{NTA}_{\mathcal{LS}} = \{A_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq \#(\mathit{inst}(\mathcal{L}_i))\} \cup \{\mathit{Coord}_i \mid 1 \leq i \leq n\} \cup \{A_{m_i} \mid 1 \leq i \leq k\}$. Therefore, the number of timed automata is $\sum_{i=1}^{n}(|\mathit{inst}(\mathcal{L}_i)|) + n + |\bigcup_{i=1}^{n} \mathit{MA}(\mathcal{L}_i)|$. See Table 1, lower part right column.

According to rule R2, each message label corresponds to a channel in $\mathit{NTA}_{\mathcal{LS}}$. According to R4, R5 and R7, there will be a set of auxiliary channels $\mathit{Aux}_1 = \{\mathit{pch\_over}_{u,i}, \mathit{mch\_over}_{u,i} \mid 1 \leq u \leq n,\ 1 \leq i \leq \#(\mathit{inst}(\mathcal{L}_u))\} \cup \{\mathit{activate}_u, \mathit{over}_u, \mathit{pch\_vio}_u, \mathit{reset}_u \mid 1 \leq u \leq n\}$ that will be used in $\mathit{NTA}_{\mathcal{LS}}$. According to Section 6 ("Translation of assignments"), in the worst case, there will be a set of auxiliary channels $\mathit{Aux}_2 = \{m_i\_\mathit{Rpt}, m_i\_\mathit{Rst},\ m_i\_\mathit{Rcv} \mid 1 \leq i \leq k\}$ for translating clock resets. Therefore, in the worst case, the number of channels in $\mathit{NTA}_{\mathcal{LS}}$ will be $|\bigcup_{i=1}^{n} \mathit{ML}(\mathcal{L}_i)| + \sum_{i=1}^{n}(2 \cdot |\mathit{inst}(\mathcal{L}_i)| + 4) + 3 \cdot |\bigcup_{i=1}^{n} \mathit{MA}(\mathcal{L}_i)|$.

According to Section 6 ("Basic mapping rules"), there will be a set of auxiliary variables $\{m_i\_\mathit{src},\ m_i\_\mathit{dest} \mid 1 \leq i \leq k\}$. According to rules R4, R8 and R9, there will be a set of auxiliary variables $\{\mathit{prematch}_u,\ \mathit{dInst}_u \mid 1 \leq u \leq n\}$. According to Section 6 ("Translation of assignments"), there will be auxiliary variables $\{m_i\_\mathit{count},\ m_i\_\mathit{done} \mid 1 \leq i \leq k\}$. Furthermore, according to R12, there will be auxiliary variables $\{mo_i\_\mathit{maySnd},\ mo_i\_\mathit{mayRcv} \mid 1 \leq i \leq s\}$. Therefore, the total number of auxiliary variables in $\mathit{NTA}_{\mathcal{LS}}$ will be $(2 \cdot |\bigcup_{i=1}^{n} \mathit{MA}(\mathcal{L}_i)|) + 2n + 1 + (2 \cdot |\bigcup_{i=1}^{n} \mathit{MA}(\mathcal{L}_i)|) + (2 \cdot \sum_{i=1}^{n} |\mathit{MO}(\mathcal{L}_i)|)$. $\square$

The complexities for the other three settings can be analyzed similarly.

Appendix D: Proof of Lemmas and Theorems

Let $\mathcal{L}$ be an untimed LSC chart whose instance lines $I_1, I_2, \ldots, I_n$ correspond to timed automata $A_1, A_2, \ldots, A_n$, respectively, then the translated network of TAs will be $\mathit{NTA}_{\mathcal{L}} = \{A_i \mid 1 \leq i \leq n\} \cup \{\mathit{Coord}\}$. According to rules R4, R5 and R7, there will be a set of auxiliary channels $\mathit{Aux} = \{\mathit{pch\_over}_i, \mathit{mch\_over}_i \mid 1 \leq i \leq n\} \cup \{\mathit{activate}, \mathit{over}, \mathit{pch\_vio},\ \mathit{reset}\}$ that will be used in $\mathit{NTA}_{\mathcal{L}}$. Let the message alphabet of $\mathcal{L}$ be Σ, then the alphabet of observable actions in $\mathit{NTA}_{\mathcal{L}}$ will be $\mathit{Act} = (\Sigma \cup \mathit{Aux})$.

Lemma 1. *Let $\mathcal{L}$ be an untimed LSC chart whose message alphabet is Σ, and let $\mathit{NTA}_{\mathcal{L}}$ be the translated network of timed automata which have a set Act of observable actions. Then* $\forall \gamma_1 \in (\Sigma \cup \{\tau\})^{\omega}.\ ((\gamma_1 \models \mathcal{L}) \Rightarrow \exists \gamma_2 \in (\mathit{Act} \cup \{\tau\})^{\omega}.(\gamma_2 \models$

$NTA_{\mathcal{L}}) \wedge (\gamma_2|_\Sigma = \gamma_1|_\Sigma))$, *and* $\forall \gamma_2 \in (Act \cup \{\tau\})^\omega.((\gamma_2 \models NTA_{\mathcal{L}}) \Rightarrow \exists! \gamma_1 \in (\Sigma \cup \{\tau\})^\omega.(\gamma_1 \models \mathcal{L}) \wedge (\gamma_2|_\Sigma = \gamma_1|_\Sigma))$.

Proof. We can prove the above two implications by proving that each cut of chart $\mathcal{L}$ uniquely corresponds to a location vector in the network of timed automata $NTA_{\mathcal{L}}$, and each advancement step in $\mathcal{L}$ uniquely corresponds to either a message synchronization transition (ranging on $\Sigma \cup Aux$) or a sequence of concatenated message synchronization and internal action transitions in $NTA_{\mathcal{L}}$, such that they consume exactly the same letter from Σ if they are both projected to Σ. Note that we restrict the LSC advancement steps to represent only legal (i.e. admissible) behaviors.

Let the instance lines in chart $\mathcal{L}$ be $I_1, I_2, \ldots, I_n$. They will be translated into timed automata $A_1, A_2, \ldots, A_n$, respectively. Together with the auxiliary timed automaton *Coord* they constitute $NTA_{\mathcal{L}}$.

The initial cut c^0 of chart $\mathcal{L}$ corresponds to the LSC initial position vector $(0_1, 0_2, \ldots, 0_n)$, where i_j means that instance $I_j \in inst(\mathcal{L})$ is currently in its position $i \in pos(\mathcal{L}, I_j)$. In the translated network of timed automata $NTA_{\mathcal{L}}$, automaton *Coord* is initially in its location l^0_{coord}. By rule R1, each 0_i in position vector $(0_1, 0_2, \ldots, 0_n)$ corresponds to a TA location l^0_i (denoting location 0 in timed automaton A_i). Therefore, cut c_0 uniquely corresponds to the $NTA_{\mathcal{L}}$ initial location vector $\bar{l}^0 = (l^0_1, l^0_2, \ldots, l^0_n, l^0_{coord})$.

We show how the advancement steps from the LSC initial position vector correspond to the transitions in the network of timed automata. At LSC position vector $(0_1, 0_2, \ldots, 0_n)$, there are two kinds of possible advancement steps:

- If there is an m-labeled message occurrence mo from position 1_i of instance I_i to position 1_j of instance I_j (i.e., mo is a minimal event), then:

  On one hand, by rules R2 and R3, there will be an $m!$-labeled TA edge from location l^0_i to l^1_i in A_i, and an $m?$-labeled TA edge from location l^0_j to l^1_j in A_j. According to the LSC semantics, there is a message synchronization advancement step on m in $\mathcal{L}$ from $(0_1, \ldots, 0_i, \ldots, 0_j, \ldots, 0_n)$ to $(0_1, \ldots, 1_i, \ldots, 1_j, \ldots, 0_n)$. Accordingly, in $NTA_{\mathcal{L}}$ there exists exactly a corresponding binary synchronization on channel m between A_i and A_j, and the location vector of $NTA_{\mathcal{L}}$ will change from $(l^0_1, \ldots, l^0_i, \ldots, l^0_j, \ldots, l^0_n, l^0_{coord})$ to $(l^0_1, \ldots, l^1_i, \ldots, l^1_j, \ldots, l^0_n, l^0_{coord})$.

  On the other hand, according to the semantics of the invariant mode universal chart, the message as a minimal event can be constantly matched for with $\mathcal{L}$ staying in the initial cut. By rule R9, in $NTA_{\mathcal{L}}$ there will be first a binary synchronization on channel m, i.e., $(l^0_1, \ldots, l^0_i, \ldots, l^0_j, \ldots, l^0_n, l^0_{coord}) \xrightarrow{m} (l^0_1, \ldots, l^0_i, \ldots, l^{PM}_j, \ldots, l^0_n, l^0_{coord})$, and then an immediately following internal action transition that leads back to the initial location vector, i.e., $(l^0_1, \ldots, l^0_i, \ldots, l^{PM}_j, \ldots, l^0_n, l^0_{coord}) \xrightarrow{\tau}$

$(l_1^0, \ldots, l_i^0, \ldots, l_j^0, \ldots, l_n^0, l_{coord}^0)$. Here l_j^{PM} is an auxiliary TA location that is specially used for prechart pre-matching. In this sub-case of pre-matching, the m-synchronization advancement step in $\mathcal{L}$ uniquely corresponds to a sequence of the tightly concatenated $\xrightarrow{m}$ and $\xrightarrow{\tau}$ transitions.

Since a dedicated flag boolean variable *prematch* has been used to strengthen the TA transition guards, assignments and the location invariants, it follows that at $NTA_{\mathcal{L}}$ location vector $(l_1^0, \ldots, l_i^0, \ldots, l_j^0, \ldots, l_n^0, l_{coord}^0)$, there are only the two above-mentioned possible interleaved executions between the two m!-labeled outgoing edges from l_i^0 in I_i and the two m?-labeled outgoing edges from l_j^0 in I_j.

- If instance I_i has no interactions with other instance lines in the prechart, then there is an immediate silent advancement step from $(0_1, \ldots, 0_i, \ldots, 0_n)$ to $(0_1, \ldots, 1_i, \ldots, 0_n)$. By rule R4, l_i^0 will be a committed location in $NTA_{\mathcal{L}}$, and there will be a pch_over_i!-labeled edge from l_i^0 to l_i^1. Furthermore, in automaton *Coord* there will be a coupling pch_over_i?-labeled edge either

  - from l_{coord}^0 to l_{coord}^1, corresponding to the case where I_i is the very last instance to complete the prechart; or
  - from l_{coord}^0 to l_{coord}^0, corresponding to the case where I_i is not yet the last instance to complete the prechart.

  In the two cases, the location vector of $NTA_{\mathcal{L}}$ will be changed from $(l_1^0, \ldots, l_i^0, \ldots, l_n^0, l_{coord}^0)$ to $(l_1^0, \ldots, l_i^1, \ldots, l_n^0, l_{coord}^1)$, and from $(l_1^0, \ldots, l_i^0, \ldots, l_n^0, l_{coord}^0)$ to $(l_1^0, \ldots, l_i^1, \ldots, l_n^0, l_{coord}^0)$, respectively. However, in both cases, there will be exactly one binary synchronization transition on pch_over_i in $NTA_{\mathcal{L}}$.

The above two kinds of possible advancement steps indicate that there is an initial correspondence between the position vector of $\mathcal{L}$ and the location vector of $NTA_{\mathcal{L}}$ [5]. Since an untimed chart is a message-only chart, a cut vector is itself an LSC configuration, and a location vector is itself a semantic state of the translated network of timed automata [6]. Therefore, there is an initial "cut-to-location vector", and "advancement step-to-(sequence of) transition" correspondence between $\mathcal{L}$ and $NTA_{\mathcal{L}}$.

The above correspondence can be generalized by using induction. Assume that at a cut c that corresponds to a position vector $(p1_1, \ldots, pi_i, \ldots, pj_j, \ldots, pn_n)$ in the prechart of $\mathcal{L}$, there is an m-labeled message occurrence sent from position

[5]More precisely the sub-location vector of $NTA_{\mathcal{L}}$ that is projected to $A_1||A_2||\ldots||A_n$. Note that the edges in *Coord* correspond only to auxiliary messages rather than the observable messages in Σ or the internal (τ) action.

[6]Note that in the LSC chart, the message sender/receiver and other relevant information are not defined as a part of the chart configuration. Accordingly, the auxiliary and bookkeeping variable information are excluded from the semantic states of the translated timed automata.

$(pi+1)_i$ of instance I_i to position $(pj+1)_j$ of instance I_j. If for cut c, there uniquely exists a corresponding location vector $\bar{l}$ in $NTA_{\mathcal{L}}$, then similar to the case of the initial cut, we can prove that the message synchronization advancement step on m in $\mathcal{L}$ uniquely corresponds to a binary synchronization transition in $NTA_{\mathcal{L}}$; and after this message synchronization advancement step, the new cut c' uniquely corresponds to the destination location vector $\bar{l}'$ in $NTA_{\mathcal{L}}$. Proof by induction ensures that any normal (i.e., other than the prechart pre-matching ones) message synchronization advancement step in the prechart of $\mathcal{L}$ uniquely corresponds to a message synchronization transition in $NTA_{\mathcal{L}}$.

In case that $(p1_1,\ldots,pi_i,\ldots,pj_j,\ldots,pn_n)$ is a position vector in the main chart of $\mathcal{L}$, the unique correspondence relation can be proved similarly.

Now we prove the unique correspondence for the case that involves the intra-chart coordination (e.g., the prechart to main chart transition). Assume that in the prechart of $\mathcal{L}$, a cut c corresponds to position vector $(p1_1,\ldots,pi_i,\ \ldots,pn_n)$, where $pi+1=\mathcal{L}.I_i.Pch\_bot$. If $(p1_1,\ldots,pi_i,\ldots,pn_n)$ uniquely corresponds to a location vector $(l_1^{p1},\ldots,l_i^{pi},\ldots,l_n^{pn},l_{coord}^0)$, then by rule R4, the internal advancement step $(p1_1,\ldots,pi_i,\ldots,pn_n)\xrightarrow{\tau}(p1_1,\ldots,(pi+1)_i,\ldots,pn_n)$ in $\mathcal{L}$ corresponds to either

- transition $(l_1^{p1},\ldots,l_i^{pi},\ldots,\ l_n^{pn},l_{coord}^0)\xrightarrow{pch\_over_i}(l_1^{p1},\ldots,l_i^{pi+1},\ldots,l_n^{pn},l_{coord}^1)$ in $NTA_{\mathcal{L}}$, in which case I_i is the very last instance to complete the prechart; or
- transition $(l_1^{p1},\ldots,l_i^{pi},\ldots,l_n^{pn},\ l_{coord}^0)\xrightarrow{pch\_over_i}(l_1^{p1},\ldots,l_i^{pi+1},\ldots,l_n^{pn},l_{coord}^0)$ in $NTA_{\mathcal{L}}$, in which case I_i is not yet the last instance to complete the prechart.

The above-mentioned first case will be followed by an intra-chart coordination, i.e., there will be an immediately following silent advancement step in $\mathcal{L}$, i.e., all instance lines will move from their *Pch_bot* positions to their *Mch_top* positions at the same time. By rule R5, the binary synchronization transition will be immediately followed by a broadcast synchronization transition $(l_1^{p1},\ldots,l_i^{pi+1},\ldots,l_n^{pn},$ $l_{coord}^1)\xrightarrow{activate}(l_1^{p1+1},\ldots,l_i^{pi+2},\ldots,l_n^{pn+1},l_{coord}^2)$, where $p1+1=\mathcal{L}.I_1.Mch\_top,\ldots,$ $pi+2=\mathcal{L}.I_i.Mch\_top,\ \ldots,pn+1=\mathcal{L}.I_n.Mch\_top$. Therefore in this case, there is a correspondence between the behaviors of $\mathcal{L}$ and $NTA_{\mathcal{L}}$.

In case that $(p1_1,\ldots,pi_i,\ldots,pn_n)$ is a position vector in the main chart of $\mathcal{L}$, the unique correspondence for the case that concerns main chart completion can be proved similarly.

Now we prove the unique correspondence for the case that involves cold violations. Since an untimed chart has no conditions, a cold violation is caused only by the violation of the partial order in the prechart. In this case, all the instance lines in the prechart of $\mathcal{L}$ will be brought from where they are back to their initial positions. Recall that $psn: loc(\mathcal{L})\rightarrow\bigcup_{I_i\in inst(\mathcal{L})}pos(\mathcal{L},I_i)$ projects a location to its position on its instance line. Formally, let us assume that $\mathcal{L}$ is

in the cut c which corresponds to the position vector $(p1_1, \ldots, pi_i, \ldots, pj_j, \ldots, pn_n)$ such that $pk_k < \mathcal{L}.I_k.Pch\_bot, 1 \leq k \leq n$. For any message label $m \in \Sigma$, if $\nexists mo \in MO(\mathcal{L}).(lab(mo) = m) \wedge (\exists I_i, I_j \in inst(\mathcal{L}).((src(mo) = I_i) \wedge (dest(mo) = I_j) \wedge (psn(tail(mo)) = pi + 1) \wedge (psn(head(mo)) = pj + 1)))$, then at cut c, the partial order will be cold-violated by any m-labeled message from I_i to I_j. For such an m-labeled message occurrence mo, by rule R7, the cold violation step $(p1_1, \ldots, pi_i, \ldots, pj_j, \ldots, pn_n) \xrightarrow{m} (0_1, \ldots, 0_i, \ldots, 0_j, \ldots, 0_n)$ in $\mathcal{L}$ uniquely corresponds to a sequence of three concatenated synchronizations in $NTA_{\mathcal{L}}$:
$(l_1^{p1}, \ldots, l_i^{pi}, \ldots, l_j^{pj}, \ldots, l_n^{pn}, l_{coord}^0) \xrightarrow{m}$
$(l_1^{p1}, \ldots, l_i^{pi+1}, \ldots, l_j^{Rst}, \ldots, l_n^{pn}, l_{coord}^0) \xrightarrow{pch\_vio}$
$(l_1^{p1}, \ldots, l_i^{pi+1}, \ldots, l_j^{0}, \ldots, l_n^{pn}, l_{coord}^{Rst}) \xrightarrow{reset}$
$(l_1^{0}, \ldots, l_i^{0}, \ldots, l_j^{0}, \ldots, l_n^{0}, l_{coord}^0)$.

Note that according to rule R8, a hot violation in the main chart of $\mathcal{L}$ will end up with a semantic state that has a deadend location in a certain TA of $NTA_{\mathcal{L}}$. This transition will *not* be considered as a part of an accepted trace of $NTA_{\mathcal{L}}$.

In conclusion, each possible advancement step in $\mathcal{L}$ uniquely corresponds to a sequence of concatenated message synchronization and internal action transitions in $NTA_{\mathcal{L}}$. They consume exactly the same message label in Σ. Therefore, each accepted trace in $\mathcal{L}$ uniquely corresponds to an accepted trace in $NTA_{\mathcal{L}}$ modulo the message alphabet Σ. □

Let $\mathcal{LS}$ be a set of untimed LSC charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$. Each chart $\mathcal{L}_i$ contains the instance lines $I_{i,1}, I_{i,2}, \ldots, I_{i,in_i}$, where $1 \leq i \leq n$, and $in_i = \#(inst(\mathcal{L}_i))$ denotes the number of instance lines in $\mathcal{L}_i$. The entire translated network of TAs will be $NTA_{\mathcal{LS}} = \{A_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq \#(inst(\mathcal{L}_i))\} \cup \{Coord_i \mid 1 \leq i \leq n\}$. The message alphabet of $\mathcal{LS}$ will be the union of all the message alphabets for the individual charts, i.e., $\Pi = \bigcup_{i=1}^{n} \Sigma_i$. The alphabet of observable actions will be $Act = (\Pi \cup Aux)$.

Lemma 2. *Let $\mathcal{LS}$ be a set of untimed LSC charts whose message alphabet is Π, and let $NTA_{\mathcal{LS}}$ be the translated network of timed automata which have a set $Act = \Pi \cup Aux$ of normal and auxiliary channels. Then $\forall \gamma_1 \in (\Pi \cup \{\tau\})^\omega.((\gamma_1 \models \mathcal{LS}) \Rightarrow \exists \gamma_2 \in (Act \cup \{\tau\})^\omega.(\gamma_2 \models NTA_{\mathcal{LS}}) \wedge (\gamma_2|_\Pi = \gamma_1|_\Pi))$, and $\forall \gamma_2 \in (Act \cup \{\tau\})^\omega.((\gamma_2 \models NTA_{\mathcal{LS}}) \Rightarrow \exists! \gamma_1 \in (\Pi \cup \{\tau\})^\omega.(\gamma_1 \models \mathcal{LS}) \wedge (\gamma_2|_\Pi = \gamma_1|_\Pi))$.*

Proof. In this case, in order to prove the above two implications, we need to prove that each *cut vector* of $\mathcal{LS}$ uniquely corresponds to a location vector in $NTA_{\mathcal{LS}}$, and each advancement step in $\mathcal{LS}$ uniquely corresponds to an equivalence class of sequences of concatenated (broadcast) synchronization and internal action transitions in $NTA_{\mathcal{LS}}$. Although elements in the equivalence class have different intermediate location vectors, they have the same initial and final location vectors. They consume exactly the same message in Π. Note that an advancement step in $\mathcal{LS}$ always represents a legal behavior.

By Lemma 1, for each untimed chart $\mathcal{L}_i$ in $\mathcal{LS}$, each cut in $\mathcal{L}_i$ uniquely corresponds to a location vector in the corresponding network of timed automata $NTA_{\mathcal{L}_i}$, and each advancement step in $\mathcal{L}_i$ uniquely corresponds to either a single message synchronization transition, or a sequence of concatenated message synchronization and internal action transitions in $NTA_{\mathcal{L}_i}$.

The only semantic difference between the advancement steps of a single untimed chart and of a set of untimed charts is that in the latter case there exist *inter-chart* coordinations, i.e., across-chart broadcast synchronization on message occurrences of the same message is possible. This implies that:

(1) At a cut vector of $\mathcal{LS}$, if in more than one chart there are enabled message occurrences of the same message, then either all of them are chosen to be fired simultaneously, or none of them is chosen to be fired;

(2) Due to the nature of broadcast synchronization in the translated network of TAs, while a message at a cut vector of $\mathcal{LS}$ could correspond to a legal message synchronization advancement step in a certain chart, meanwhile it could also lead another chart to be reset by cold-violating the prechart of that chart (case 2.1), or lead another chart to a deadlocked situation by hot-violating the main chart of that chart (case 2.2).

In case (1), given a set $\mathcal{LS}$ of untimed LSC charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$, we let $in_i = \#(inst(\mathcal{L}_i))$, $1 \leq i \leq n$. We assume that the current cut vector $\bar{c}$ of $\mathcal{LS}$ uniquely corresponds to the position vector $(p_{1,1}, p_{1,2}, \ldots, p_{1,in_1}, p_{2,1}, p_{2,2}, \ldots, p_{2,in_2}, \ldots, p_{n,1}, p_{n,2}, \ldots, p_{n,in_n})$, where $p_{i,j} \in pos(\mathcal{L}_i, I_j)$ denotes the current position on instance I_j of chart $\mathcal{L}_i$. Without loss of generality, we assume that two m-labeled message occurrences mo_1 and mo_2 are enabled at cut vector $\bar{c}$ in two charts $\mathcal{L}_i$ and $\mathcal{L}_j$, respectively. Specifically, let $(p_{i,a}+1)$ and $(p_{i,b}+1)$ be the sending and receiving positions of mo_1 in $\mathcal{L}_i$, where $1 \leq a, b \leq in_i$, and let $(p_{j,c}+1)$ and $(p_{j,d}+1)$ be the sending and receiving positions of mo_2 in $\mathcal{L}_j$, where $1 \leq c, d \leq v_j$. According to the trace-based semantics for a set of charts, these two message synchronization advancement steps in $\mathcal{L}_i$ and $\mathcal{L}_j$ will occur simultaneously. By rules R2 and R3, there will be an $m!$-labeled edge from location $l_{i,a}^{p_{i,a}}$ to $l_{i,a}^{p_{i,a}+1}$ in $A_{i,a}$, and an $m?$-labeled edge from location $l_{i,b}^{p_{i,b}}$ to $l_{i,b}^{p_{i,b}+1}$ in $A_{i,b}$, and similarly for chart $\mathcal{L}_j$. By rule R6, there will be added an extra $m?$-labeled edge from location $l_{i,a}^{p_{i,a}}$ to $l_{i,a}^{p_{i,a}+1}$ in $A_{i,a}$, and similarly in chart $\mathcal{L}_j$. Consequently, there will be a broadcast synchronization on m among $A_{i,a}$, $A_{i,b}$, $A_{j,c}$, $A_{j,d}$, initiated either by $A_{i,a}$, or by $A_{j,c}$. In either case, after this broadcast synchronization on m in $NTA_{\mathcal{LS}}$, the locations of $A_{i,a}$, $A_{i,b}$, $A_{j,c}$ and $A_{j,d}$ will progress to $l_{i,a}^{p_{i,a}+1}$, $l_{i,b}^{p_{i,b}+1}$, $l_{j,c}^{p_{j,c}+1}$ and $l_{j,d}^{p_{j,d}+1}$, respectively. Therefore, the message synchronization advancement step on m in $\mathcal{LS}$ corresponds to two possible interleaved executions among $A_{i,a}$, $A_{i,b}$, $A_{j,c}$ and $A_{j,d}$. Since both interleavings consume the same message label m, they correspond to the same portion of the accepted trace in $NTA_{\mathcal{LS}}$. These two interleaved executions constitute an equivalence class with respect to the message synchronization advancement step on m.

In case (2.1), assume that the current cut vector $\bar{c}$ of $\mathcal{LS}$ corresponds to position vector $(p_{1,1}, p_{1,2}, \ldots, p_{1,in_1}, p_{2,1}, p_{2,2}, \ldots, p_{2,in_2}, \ldots, p_{n,1}, p_{n,2}, \ldots, p_{n,in_n})$. Without loss of generality, we assume that an m-labeled message occurrence mo is currently enabled in $\mathcal{L}_i$, but not in $\mathcal{L}_j$, and that $\bar{c}$ "cuts" $\mathcal{L}_j$ in the prechart of $\mathcal{L}_j$. According to the semantics for a set of LSC charts, when message m is encountered, there will be a normal advancement step in $\mathcal{L}_i$, and a cold violation advancement step in $\mathcal{L}_j$. By Lemma 1, such a cold violation advancement step uniquely corresponds to a sequence of synchronizations in the relevant timed automata. Therefore, the system-wide synchronization on m will also uniquely correspond to a system-wide sequence of synchronizations in $NTA_{\mathcal{LS}}$.

In case (2.2), assume that the current cut vector $\bar{c}$ of $\mathcal{LS}$ corresponds to position vector $(p_{1,1}, p_{1,2}, \ldots, p_{1,in_1}, p_{2,1}, p_{2,2}, \ldots, p_{2,in_2}, \ldots, p_{n,1}, p_{n,2}, \ldots, p_{n,in_n})$. Without loss of generality, we assume that an m-labeled message occurrence mo is currently enabled in $\mathcal{L}_i$, but not in $\mathcal{L}_j$, and that $\bar{c}$ "cuts" $\mathcal{L}_j$ in the main chart of $\mathcal{L}_j$. According to the semantics for a set of LSC charts, when message m is encountered, there will be a normal message synchronization advancement step in $\mathcal{L}_i$, and a hot violation in $\mathcal{L}_j$. Specifically, let $p_{i,a}$ and $p_{i,b}$ be the sending and receiving positions of mo in $\mathcal{L}_i$, where $1 \leq a, b \leq in_i$. We let the sub-position vector in $\mathcal{L}_j$ be $c_j = (p_{j,1}, p_{j,2}, \ldots, p_{j,in_j})$. Obviously, mo is not enabled at subcut c_j. Since $\mathcal{L}_j$ is hot-violated by mo, there must exist a position, say $p_{j,x}$, $1 \leq x \leq in_j$, such that there is an $m?$-labeled edge from $p_{j,x}$ to a sink error location `Err` in $A_{j,x}$. Furthermore, there could possibly exist another position, say $p_{j,y}$, $1 \leq y \leq in_j$, such that there is an $m!$-labeled edge from position $p_{j,y}$ to $(p_{j,y} + 1)$ in $A_{j,y}$. This means that there could be one or two possible initiating TAs of the broadcast synchronization. Whichever case could it be, the same label (m) will be consumed, and the same next semantic state of $NTA_{\mathcal{LS}}$ will be reached. This semantic state will have a deadend location `Err`, which indicates that the system will be deadlocked. Therefore, the TA transition step leading to this semantic state will not be considered as a part of the accepted trace. In this case, m will not be allowed to occur at cut vector $\bar{c}$. This demonstrates how the different charts constrain the behaviors of each others. In summary, in case (2.2), a to-be-hot violating message in $\mathcal{LS}$ uniquely corresponds to a to-be-deadlocked TA transition in $NTA_{\mathcal{LS}}$.

Based on the above discussions, we conclude that there exists a unique correspondence between the observable traces of a set of untimed LSC charts and their corresponding network of timed automata. □

Let $\mathcal{L}$ be a time-enriched chart whose instance lines $I_1, I_2, \ldots, I_n$ correspond to timed automata $A_1, A_2, \ldots, A_n$, respectively. Let the message alphabet of $\mathcal{L}$ be $\{m_1, m_2, \ldots, m_k\}$. According to Section 6 ("Translation of assignments"), there will be an auxiliary timed automaton A_{m_i} for each m_i, $1 \leq i \leq k$. Consequently, the translated network of TAs will be $NTA_{\mathcal{L}} = \{A_i \mid 1 \leq i \leq n\} \cup \{Coord\} \cup \{A_{m_i} \mid 1 \leq i \leq k\}$.

According to rules R4, R5 and R7, there will be auxiliary channels $Aux = \{pch\_over_i,\ mch\_over_i \mid 1 \leq i \leq n\} \cup \{activate, over, pch\_vio, reset\}$ used in $NTA_{\mathcal{L}}$. According to rule R11, there will be auxiliary channels $Aux' = \{m_i\_Rpt, m_i\_Rst, m_i\_Rcv \mid 1 \leq i \leq k\}$ used in $NTA_{\mathcal{L}}$. Let the message alphabet of $\mathcal{L}$ be Σ, then the alphabet of observable actions in $NTA_{\mathcal{L}}$ will be $Act = \Sigma \cup Aux \cup Aux'$.

Lemma 3. *Let $\mathcal{L}$ be a time-enriched LSC chart whose message alphabet is Σ, and let $NTA_{\mathcal{L}}$ be the translated network of timed automata which have a set $Act = \Sigma \cup Aux \cup Aux'$ of normal and auxiliary channels. Then $\forall \gamma_1 \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega . ((\gamma_1 \models \mathcal{L}) \Rightarrow \exists \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega . (\gamma_2 \models NTA_{\mathcal{L}}) \wedge (\gamma_2|_{(\Sigma \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Sigma \cup \mathbb{R}_{\geq 0})}))$, and $\forall \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega . ((\gamma_2 \models NTA_{\mathcal{L}}) \Rightarrow \exists! \gamma_1 \in (\Sigma \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^\omega . (\gamma_1 \models \mathcal{L}) \wedge (\gamma_2|_{(\Sigma \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Sigma \cup \mathbb{R}_{\geq 0})}))$.*

Proof. In order to prove the above two implications, we need to show that each configuration of chart $\mathcal{L}$ uniquely corresponds to a certain semantic state of $NTA_{\mathcal{L}}$, and each advancement step in $\mathcal{L}$ uniquely corresponds to a sequence of concatenated message synchronization transitions, and/or internal action transitions, and/or time delay transitions in $NTA_{\mathcal{L}}$ such that they either consume exactly the same letter from Σ, or undergo exactly the same period of time delay.

By Lemma 1, each cut of an untimed chart $\mathcal{L}$ uniquely corresponds to a semantic state in $NTA_{\mathcal{L}}$, and each advancement step in $\mathcal{L}$ uniquely corresponds to either a message synchronization transition, or a sequence of concatenated message synchronization and internal action transitions in $NTA_{\mathcal{L}}$. For a time-enriched LSC chart, we keep this skeleton correspondence, i.e., we map position pi_i of instance line I_i to location l_i^{pi} of the timed automaton A_i. Note that along an instance line of the time-enriched chart, two adjacent LSC positions typically do not correspond to two adjacent locations in the corresponding translated TA. Between location l_i^{pi} and l_i^{pi+1}, where $0 \leq pi \leq (p\_max_{\mathcal{L},I_i} - 1)$, according to rules R10, R11 and R12, we will add some intermediate auxiliary TA locations, and add some TA edges to connect them.

Now we prove that a message synchronization advancement step on m in $\mathcal{L}$ uniquely corresponds to a sequence of transitions in $NTA_{\mathcal{L}}$ that consumes m exactly. Assume that at a configuration c which corresponds to position vector $(p1_1, \ldots, pi_i, \ldots, pj_j, \ldots, pn_n)$ in the prechart of $\mathcal{L}$ and clock valuation v, there is an m-labeled message occurrence mo with condition (clock constraints) g and assignment (clock resets) a sent from position $(pi+1)_i$ of instance I_i to position $(pj+1)_j$ of instance I_j. Assume that position pi_i corresponds to location l_i^{pi} in A_i, and position $(pi+1)_i$ corresponds to location l_i^{pi+1} in A_i, then there will be 5 intermediate locations between l_i^{pi} and l_i^{pi+1} in A_i, which we denote as $l_i^{pi,1}, l_i^{pi,2}, l_i^{pi,3}, l_i^{pi,4}$ and $l_i^{pi,5}$. Here

- between l_i^{pi} and $l_i^{pi,1}$, there is a TA edge with the guard "$m\_mayRcv ==$ `true`";
- between $l_i^{pi,1}$ and $l_i^{pi,2}$, there is a TA edge which tests the upper bound constraints;
- between $l_i^{pi,2}$ and $l_i^{pi,3}$, there is an $m!$-labeled TA edge;
- between $l_i^{pi,3}$ and $l_i^{pi,4}$, there is a TA edge which tests the lower bound and/or clock difference constraints;
- between $l_i^{pi,4}$ and $l_i^{pi,5}$, there is an $m\_Rpt!$-labeled TA edge;
- between $l_i^{pi,5}$ and l_i^{pi+1}, there is an $m\_Rst!$-labeled TA edge.

Similarly, there will be 5 intermediate locations and edges that connect them in A_j. Specifically, if there are positions on other instance line that are waiting for the completion of this message synchronization according to the partial order relation, then there will be one more intermediate location $l_i^{pi,6}$, and an $m\_Rcv!$-labeled edge connecting $l_i^{pi,6}$ to l_i^{pi+1}. According to rule R12, the position sub-vector (pi_i, pj_j) corresponds to the TA location sub-vector (l_i^{pi}, l_j^{pj}), where both locations are committed locations. After these two transitions from (l_i^{pi}, l_j^{pj}), the new location sub-vector $(l_i^{pi,1}, l_j^{pj,1})$ will be reached, which are also committed locations. Since a legal advancement step in $\mathcal{L}$ will not violates the upper bound of the clock constraints, the upper bound constraint will evaluate to true and thus the next location sub-vector will be $(l_i^{pi,2}, l_j^{pj,2})$. From $(l_i^{pi,2}, l_j^{pj,2})$ there will be the message synchronization on m leading to $(l_i^{pi,3}, l_j^{pj,3})$, which are again committed locations. After comparing the lower bound of clock constraints, the location sub-vector $(l_i^{pi,4}, l_j^{pj,4})$ will be reached. Now instance lines I_i and I_j will immediately report to the automaton A_m, telling it that the instances are done with testing the guarding flag boolean variables, testing the upper bound, message synchronization, and testing the lower bound or clock difference. Once both instance lines have notified A_m of their completions, A_m will immediately initiate an m_Rst-labeled broadcast synchronization which brings A_i from $l_i^{pi,5}$ to l_i^{pi+1}, and brings A_j from $l_j^{pj,5}$ to l_j^{pj+1}. Specifically, if there is an $l_i^{pi,6}$ in A_i, then the $m\_Rst?$-labeled edge will be from $l_i^{pi,5}$ to $l_i^{pi,6}$ in A_i, and there will be an $m\_Rcv!$-labeled edge from $l_i^{pi,6}$ to l_i^{pi+1}. This latter edge can occur autonomously without synchronizing with a message-receiving TA, because m_Rcv is declared as a broadcast channel. In summary, the message synchronization step on m in $\mathcal{L}$ will uniquely correspond to such a sequence of transitions in $NTA_{\mathcal{L}}$.

For a silent advancement step in $\mathcal{L}$, it is the same as in the untimed case. In other words, the corresponding proof for Lemma 1 also applies here.

For a time delay advancement step in $\mathcal{L}$, since the upper bounds and lower bounds of clock constraints are properly translated to tests that are prior to and after the message synchronization in $NTA_{\mathcal{L}}$, a time delay of a period of $d \in \mathbb{R}_{\geq 0}$ is allowed in $NTA_{\mathcal{L}}$ if and only if the same period d of time delay is allowed in $\mathcal{L}$.

In all the three possible cases of an advancement step in $\mathcal{L}$, there will be a uniquely corresponding sequence of transitions in $NTA_{\mathcal{L}}$ such that this sequence consumes exactly the same message, or amount of time delay as that step in $\mathcal{L}$. □

Let $\mathcal{LS}$ be a set of time-enriched LSC charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$. Each chart $\mathcal{L}_i$ contains the instance lines $I_{i,1}, I_{i,2}, \ldots, I_{i,in_i}$, where $in_i = \#(inst(\mathcal{L}_i))$. Let the message alphabet Π of $\mathcal{LS}$ be $\Pi = \bigcup_{i=1}^{n} \Sigma_i = \{m_1, m_2, \ldots, m_k\}$. Then the translated network of TAs will be $NTA_{\mathcal{LS}} = \{A_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq \#(inst(\mathcal{L}_i))\} \cup \{Coord_i \mid 1 \leq i \leq n\} \cup \{A_{m_i} \mid 1 \leq i \leq k\}$. Similarly to Lemma 3, we let $Act = \Pi \cup Aux \cup Aux'$.

Theorem 1. Let $\mathcal{LS}$ be a set of time-enriched LSC charts whose message alphabet is Π, and let $NTA_{\mathcal{LS}}$ be the translated network of timed automata which have a set Act of normal and auxiliary channels. Then $\forall \gamma_1 \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^{\omega}.\ ((\gamma_1 \models \mathcal{LS}) \Rightarrow \exists \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^{\omega}.(\gamma_2 \models NTA_{\mathcal{LS}}) \wedge (\gamma_2|_{(\Pi \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Pi \cup \mathbb{R}_{\geq 0})}))$, and $\forall \gamma_2 \in (Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^{\omega}.\ ((\gamma_2 \models NTA_{\mathcal{LS}}) \Rightarrow \exists! \gamma_1 \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^{\omega}.(\gamma_1 \models \mathcal{LS}) \wedge (\gamma_2|_{(\Pi \cup \mathbb{R}_{\geq 0})} = \gamma_1|_{(\Pi \cup \mathbb{R}_{\geq 0})}))$.

Proof. We need to prove that each cut vector of $\mathcal{LS}$ uniquely corresponds to a location vector in $NTA_{\mathcal{LS}}$, and each message synchronization advancement step in $\mathcal{LS}$ uniquely corresponds to a sequence of concatenated message synchronization transitions, and internal action transitions in $NTA_{\mathcal{LS}}$. These transitions are connected by committed locations in $NTA_{\mathcal{LS}}$. Because any committed location appears as a junction location only when it will be immediately followed (only) by a condition test, these concatenated transitions can be viewed as an atomic step. Although for the sake of inter-chart coordination, the outgoing transitions from locations of different TAs may be executed in an interleaved manner, the order of the consumed words in $(\Pi \cup \{\tau\})^*$ remains the same. In other words, an accepted timed trace $\gamma \in (\Pi \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^{\omega}$ may correspond to an equivalence class of timed traces in $(Act \cup \{\tau\} \cup \mathbb{R}_{\geq 0})^*$. They consume exactly the same timed trace in $(\Pi \cup \mathbb{R}_{\geq 0})^*$. Proof details concerning the translations of inter-chart message coordinations and message occurrences that are associated with conditions and/or assignments are similar to that for Lemmas 2 and 3, respectively. □

Let $\mathcal{LS}$ be an LSC system which consists of a set of (untimed or timed) driving universal charts $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$. We translate $\mathcal{LS}$ to a network of timed automata $NTA_{\mathcal{LS}}$. Let $\mathcal{L}'$ be a separate monitored universal chart (the “property chart”), which will be translated to another network of timed automata $NTA_{\mathcal{L}'}$. As explained earlier, the TA locations $Coord_{\mathcal{L}'}.Mch\_top$ and $Coord_{\mathcal{L}'}.Mch\_bot$ denote that the main chart of $\mathcal{L}'$ has just been activated and has just been successfully matched, respectively. We have:

Theorem 3. $\mathcal{LS} \models \mathcal{L}' \Leftrightarrow (\mathit{NTA}_{\mathcal{LS}} \,||\, \mathit{NTA}_{\mathcal{L}'}) \models \mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_top} \leadsto \mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_bot}.$

Proof. By Theorem 1, each accepted trace in $\mathcal{LS}$ uniquely corresponds to a cluster of accepted traces in $\mathit{NTA}_{\mathcal{LS}}$ which consume exactly the same string from $(\Pi \cup \mathbb{R}_{\geq 0})^{\omega}$. And similarly for $\mathcal{L}'$ and $\mathit{NTA}_{\mathcal{L}'}$.

The TA location $\mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_top}$ represents the situation where the property chart $\mathcal{L}'$ is activated, and $\mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_bot}$ the situation where $\mathcal{L}'$ is satisfied (i.e., successfully matched).

Since $\mathcal{L}'$ is a property chart, its corresponding network of timed automata $\mathit{NTA}_{\mathcal{L}'}$ will never interfere with (or "drive") the network of timed automata $\mathit{NTA}_{\mathcal{LS}}$. This means that after parallelly composing the TAs in $\mathit{NTA}_{\mathcal{L}'}$ with the TAs in $\mathit{NTA}_{\mathcal{LS}}$, the behaviors in $\mathit{NTA}_{\mathcal{LS}}$ will not be further constrained. Since both $\mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_top}$ and $\mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_bot}$ are locations in the product automaton of $(\mathit{NTA}_{\mathcal{LS}} \,||\, \mathit{NTA}_{\mathcal{L}'})$, the right hand side formula of this theorem captures exactly the assume-guarantee style responsiveness property of the LSC requirement, which is exactly what we require of $\mathcal{LS} \models \mathcal{L}'$. $\Box$

An LSC system $\mathcal{LS}$ satisfies a monitored existential chart $\mathcal{L}'$ iff one of the traces in $\mathcal{LS}$ is included in the traces of $\mathcal{L}'$.

Theorem 4. $\mathcal{LS} \models \mathcal{L}' \Leftrightarrow (\mathit{NTA}_{\mathcal{LS}} \,||\, \mathit{NTA}_{\mathcal{L}'}) \models \mathsf{E}\Diamond\ \mathit{Coord}_{\mathcal{L}'}.\mathit{Mch\_bot}.$

Proof. This theorem can be proved similarly to Theorem 3, except that an existential chart has no prechart. $\Box$

Paper C:
A Game-Theoretic Approach to Real-Time System Testing

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen
Center for Embedded Software Systems (CISS)
Department of Computer Science
Aalborg University, Denmark

Abstract

We present a game-theoretic approach to the testing of real-time embedded systems whose models may have output uncertainty and timing uncertainty of outputs. By modeling a system in question using timed game automata (TGA) and specifying the test purpose as an ACTL formula, we employ a recently developed timed game solver UPPAAL-TIGA to synthesize winning strategies, and then use these strategies to conduct black-box conformance testing of the system. The testing process is proved to be sound and complete with respect to the given test purpose. Case study and preliminary experimental results indicate that this is a viable approach to real-time embedded system testing.

Keywords: Real-Time Systems, Timed Game Automata (TGA), Test Purposes, Strategies, Conformance Testing

1 Introduction

Model-based conformance testing of real-time systems has attracted increasing research interests in recent years. A large proportion of these efforts employ timed automata (TA) [AD94] or its variants to model the systems in question. Among them some make the assumptions that the system TA model is *output-urgent* and has *isolated outputs* [SVD01, HLN$^+$03]. "Output-urgent" means that if the system can ever produce an output, it should produce the output immediately (i.e., with zero time delay). "Isolated output" means that at any moment in time if the system can produce an output, at that moment it should not be able to accept any input or produce any other output. These two assumptions on TA contribute to the *testability* (or *controllability* [1]) [SVD01] of timed automata. However, in many cases they appear to be unnecessarily strong.

In this paper we aim to cancel these two assumptions and present a test method for *uncontrollable* real-time systems, i.e., systems with *timing uncertainty of outputs* and *output uncertainty*. By "timing uncertainty of outputs" we mean that the system under test (SUT) can produce an output during a certain time interval rather than only at a certain time point. By "output uncertainty" (or "uncontrollable outputs") we mean that it is the SUT rather than the tester that determines whether an output will be produced, and if yes, which of the several possible outputs will be produced. The benefits of allowing timing uncertainty of outputs and output uncertainty in the system models include:

- It allows the implementors some freedom;
- It enables the testers to concern only with the high-level requirements rather than the implementation details; and
- It usually leads to more succinct and natural models.

A system $\mathcal{S}ys$ with output uncertainty and timing uncertainty of outputs may be modeled as a *timed game automaton* (TGA) [MPS95], which is a variant of TA with its actions partitioned into controllable ones (modeling input stimuli from the tester to $\mathcal{S}ys$) and uncontrollable ones (modeling output responses from $\mathcal{S}ys$ to the tester). The output uncertainty lies in the fact that from a certain TGA location, there could be both outgoing input action edges and (multiple) outgoing output action edges. The timing uncertainty of outputs comes with the clock invariants at those locations — they accommodate periods of rather than only zero delays of outputs.

[1]A timed automaton is said to be *controllable* [SVD01] if it is possible for an environment (tester) to drive the timed automaton through all of its transitions by offering appropriate test inputs. Controllability requires a timed automaton to be deterministic (i.e., the same source location and action will lead to the same target location), output-urgent, and have isolated outputs.

The interactions between $\mathcal{S}ys$ and the tester can be viewed as a game activity, where the tester is a game player and $\mathcal{S}ys$ is the game opponent. A play of the timed game between $\mathcal{S}ys$ and the tester is a run of the TGA towards a given test purpose (or winning objective), say, "location `SUT.Bright` can always be eventually reached" (Fig. 2(a)). A previously developed timed game solver Uppaal-Tiga [BCD+07] can check whether a specified ACTL[2] winning objective can be satisfied by a TGA, and if so, it can synthesize a winning strategy. Since a winning strategy provides step-by-step guidance to the tester towards the TGA states that satisfy the test purpose, it can be viewed as a test case. This opens up the possibility of game-based testing of uncontrollable timed systems.

1.1 Related work

Recent years have seen much work on model-based black-box conformance testing of real-time systems based on the timed automata or timed transition system models [ENDKE98, HNTC99, CO00, NS01a, SVD01, HLN+03, KJM03, LMN04, BB04, KT04]. For the sake of testability, some of them assume that the timed automata are controllable [SVD01, HLN+03]. This in turn requires that the timed automata are output-urgent and have isolated outputs. In this paper, both of these two requirements are cancelled.

Testing as a game problem for untimed systems has been proposed by Alur and colleagues [ACY95], and elaborated in [Yan04]. Dense-time control problem based on timed game automaton has been defined and solved by Maler and colleagues [MPS95], and later improved in [AMPS98, TA99, AT02]. More recently, based on [LS98] a truly on-the-fly algorithm that combines forward symbolic explorations and backward propagations of winning/losing information for more efficient timed game solving has been proposed and implemented in [CDF+05]. A further optimized re-implementation of the algorithm gives rise to the tool Uppaal-Tiga [BCD+07, BCD+08], which can synthesize winning strategies for given TGA models and given ACTL properties. The dramatic performance improvement over the first prototype [CDF+05] makes it possible for controller synthesis for non-trivial timed systems.

2 Test setup

2.1 The timed control problem

In a timed control problem, the control program (or "controller") actively offers inputs to and passively observes outputs from the system under control (the "plant") at appropriate time (or time periods). The inputs are controlled by the controller (Fig. 1, solid lines), and the outputs are controlled only by the

[2]ACTL is the universal fragment of the CTL logic where the path quantifier can only be A.

plant itself (Fig. 1, dashed lines). For a given control objective we can possibly synthesize a control strategy (or winning game strategy), guided by which the control program ensures that the control objective will be enforced no matter how the plant behaves.

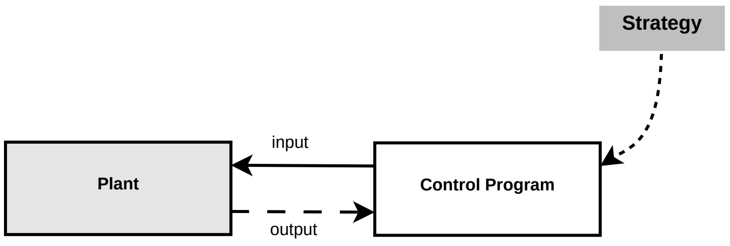


Figure 1: The timed control problem.

From a model-based testing point of view, the plant could be viewed as a system in question, and the controller could be viewed as the tester. In this way, a winning strategy for winning objective φ could be viewed as a test case for test purpose φ.

2.2 Timed I/O Game Automaton

Let X be a finite set of real-valued clocks, and $\mathcal{C}(X)$ be the set of constraints generated by the grammar:

$$\varphi ::= x \bowtie k \mid x - y \bowtie k \mid \varphi_1 \wedge \varphi_2,$$

where $k \in \mathbb{N}_{\geq 0},\ x, y \in X$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Definition 1 (timed automaton, TA [AD94]). *A* timed automaton *is a tuple* $\mathcal{S} = (L, l_0, Act, X, E, Inv)$, *where*

- L *is a finite set of locations;*
- $l_0 \in L$ *is the initial location;*
- Act *is a finite set of actions;*
- X *is a finite set of real-valued clocks;*
- $E \subseteq L \times \mathcal{C}(X) \times Act \times 2^X \times L$ *is a finite set of transitions; and*
- $Inv : L \to \mathcal{C}(X)$ *associates invariants to locations.* □

To characterize the uncontrollability of some actions, we adopt the notion of timed game automaton.

Definition 2 (timed game automaton, TGA [MPS95])**.** *A* timed game automaton *is a timed automaton with its set Act of actions partitioned into controllable actions* (Act_c) *and uncontrollable actions* (Act_u)*.* □

In this paper we further refine the above definition by assuming that all output actions Act_{out} are uncontrollable and all input actions Act_{in} are controllable.

Definition 3 (timed I/O game automaton, TIOGA)**.** *A* timed I/O game automaton *is a timed game automaton with its set of actions Act partitioned into input actions* Act_{in} *and output actions* Act_{out} *such that* $Act_{in} = Act_c$ *and* $Act_{out} = Act_u$*.* □

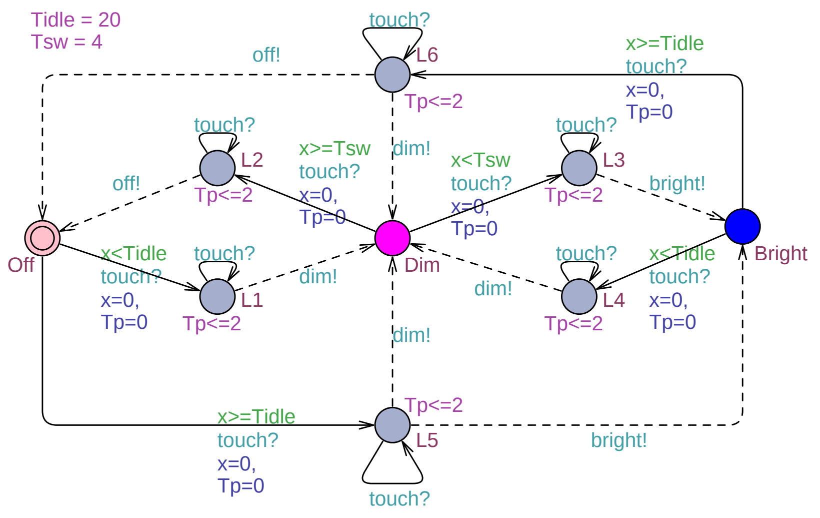
In a plant TIOGA, the controllable actions model the inputs from the controller (or the tester from a testing perspective) to the plant, and the uncontrollable actions model the outputs from the plant to the controller. A *run* of the system involves a sequence of controller-chosen input stimuli and plant-produced output reactions. Therefore, it can be viewed as a timed I/O game where the controller acts as a player and the plant acts as the opponent. Since sometimes the opponent may choose not to produce any output by just staying quiescent, the game run is not necessarily an alternating sequence of inputs and outputs.

This paper uses the simple Smart Lamp problem [HLN$^+$03] as an illustrating example. Fig. 2(a) is a TIOGA of the smart Lamp (the "plant") where solid lines represent transitions of controllable actions, and dashed lines represent transitions of uncontrollable actions[3]. Fig. 2(b) is the TIOGA of the user of the Lamp (the "controller"). There are three brightness levels for the Lamp: `Off`, `Dim` and `Bright`. The Lamp is initially in location `Off`. The user interacts with the Lamp by touching a touch-sensitive pad. The Lamp changes its brightness levels according to the timings between the touches. For example, if the Lamp has been in location `Off` for a long time ("$x \geq T_{idle}$" in Fig. 2(a)), then it is supposed to re-activate upon a *touch*? and go to location `L5`, and then either

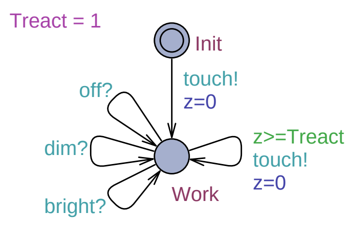
- to produce output *bright*! and go directly to location `Bright` within 2 time units; or
- to produce output *dim*! and go to location `Dim` within 2 time units; or even
- not to produce any output, and just remain in `L5` during that period.

The user does not know whether or which output will be produced. This is the so-called *output uncertainty* (or uncontrollable outputs). If an output is ever produced, the user cannot anticipate the exact time of the output. This is the so-called *timing uncertainty of outputs.*

[3]More precisely, only edges that correspond to plant-controlled message-**sending** events e! are drawn in dashed lines. All other edges are in solid lines. The reason is that controllability makes sense for message-sending events, but not for message-receiving events.



(a) the smart Lamp (`SUT`)



(b) the user

Clock variables:
x: the timing between touches;
T_p: the timing of pausing;
z: the timing between two successive touches;

Model parameters (constants):
T_{idle}: idling time;
T_{sw}: switching time;
T_{react}: reactive time (minimal delay between two successive touches).

Figure 2: TIOGAs for the smart lamp example.

We use timed I/O transition system (TIOTS) as the underlying semantic model of TIOGA.

Definition 4 (timed I/O transition system, TIOTS)**.** *A* timed I/O transition system *is a tuple* $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$*, where*

- *S is a set of states;*
- *$s_0 \in S$ is the initial state;*
- *Act_{in} and Act_{out} are sets of input and output actions, respectively; and*
- *$\rightarrow \in S \times (Act_{in} \cup Act_{out} \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the following sanity constraints:*
  - time determinism*:* $(s \xrightarrow{d} s') \wedge (s \xrightarrow{d} s'') \Rightarrow (s' = s'')$*;*
  - time additivity*:* $(s \xrightarrow{d_1} s') \wedge (s' \xrightarrow{d_2} s'') \Rightarrow (s \xrightarrow{d_1+d_2} s'')$*,*

where $\mathbb{R}_{\geq 0}$ *is the set of non-negative real numbers,* $s, s', s'' \in S$*, and* $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$. □

Let $s \in S$ and $\alpha \in (Act \cup \mathbb{R}_{\geq 0})$. We write $s \xrightarrow{\alpha}$ if $\exists s' \in S \,.\, s \xrightarrow{\alpha} s'$. Here α can be extended to strings of observable actions and time delays in the usual manner.

We define the following characteristics of TIOTS:

- A TIOTS has *isolated output* if $\forall s \in S \,.\, \forall \alpha \in Act_{out} \,.\, \forall \beta \in Act \,.\, (((s \xrightarrow{\alpha}) \wedge (s \xrightarrow{\beta})) \Rightarrow (\alpha = \beta))$; and
- A TIOTS is *output-urgent* if $\forall s \in S \,.\, \forall \alpha \in Act_{out} \,.\, ((s \xrightarrow{\alpha}) \Rightarrow \forall d \in \mathbb{R}_{>0} \,.\, (s \not\xrightarrow{d}))$.

Definition 5 (semantics of TIOGA)**.** *The* semantics *of a TIOGA* (L, l_0, Act, X, E, Inv) *is defined as a TIOTS* $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$*, where*

- $S \subseteq L \times \mathbb{R}^X$ *is the set of semantic states of location and clock vector;*
- $s_0 = (l_0, \overline{0}) \in S$ *is the initial state;*
- Act_{in} *and* Act_{out} *are sets of input and output actions, which partition Act; and*
- $\rightarrow \subseteq S \times (Act_{in} \cup Act_{out} \cup \mathbb{R}_{\geq 0}) \times S$ *satisfies the sanity constraints and consists of the following types of transitions:*
  - *time transition:* $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' \in [0, d] \,.\, ((u + d') \models Inv(l))$*;*
  - *action transition:* $(l, u) \xrightarrow{a} (l', u')$ *if* $\exists e = (l, a, g, r, l') \in E \,.\, ((u \models g) \wedge (u' = [r \rightarrow 0]u) \wedge (u' \models Inv(l')))$*.* □

A run of the TIOGA is characterized by a timed trace. An *observable timed trace* $\sigma \in (Act \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 a_2 \dots a_k d_{k+1}$. We define the set of observable timed traces of state s as:

$$\mathsf{TTr}(s) = \{\sigma \in (Act \cup \mathbb{R}_{\geq 0})^* \mid s \xrightarrow{\sigma}\}.$$

For a state s and a timed trace σ, $(s \ \mathsf{After}\ \sigma)$ is the set of states that can be reached after σ:

$$s \ \mathsf{After}\ \sigma = \{s' \mid s \xrightarrow{\sigma} s'\}.$$

The set of (immediately) observable outputs or delays at state s is defined as:

$$\mathsf{Out}(s) = \{a \in (Act_{out} \cup \mathbb{R}_{\geq 0}) \mid s \xrightarrow{a}\}.$$

The definitions of After and Out can both be extended to sets of states as usual.

A *run* of a TIOGA $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ is a timed trace (i.e., a sequence of alternating time and action transitions) in its TIOTS $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$.

We use $\mathsf{Runs}(s, \mathcal{S})$ to denote the set of all runs of $\mathcal{S}$ that start from state $s \in S$. If σ is a finite run, we use $last(\sigma)$ to denote the last state of σ.

In this paper, we only impose the following restrictions on the model of the plant:

- determinism; and
- strong input-enabledness.

In particular we do NOT require the plant model to be output-urgent (thus allowing timing uncertainty of outputs) or to have isolated outputs (thus allowing output uncertainty). Such a TIOGA is called an *uncontrollable* TIOGA. Likewise, its corresponding TIOTS is called an uncontrollable TIOTS.

Moreover, we can define the parallel composition of several TIOTS's in the usual manner.

2.3 Timed I/O conformance relation

To decide whether the behaviors of the implementation under test (IMP) conform to that of the system specification (SPEC), we use the timed input-output conformance (tioco) relation.

Definition 6 (timed input-output conformance relation, tioco [KT04])**.** *Let $i, s \in S$ be two states of a TIOTS. The* timed input-output conformance relation tioco *between i and s is defined as:*

$$i \ \mathsf{tioco}\ s \quad \textit{iff} \quad \forall \sigma \in \mathsf{TTr}(s).(\mathsf{Out}(i \ \mathsf{After}\ \sigma) \subseteq \mathsf{Out}(s \ \mathsf{After}\ \sigma)).$$

□

Assume that the behavior of the IMP can be modeled by a TIOTS $\mathcal{I}$. Assume that the initial state of $\mathcal{I}$ is i_0, and the initial semantic state of the specification TIOGA $\mathcal{S}$ is s_0. If i_0 tioco s_0, we say that $\mathcal{I}$ is a correct implementation of $\mathcal{S}$, denoted $\mathcal{I}$ tioco $\mathsf{TIOTS}(\mathcal{S})$.

Let $\mathcal{S}$ be input-enabled, and let i and s be two of its semantic states. The relation tioco can also be characterized in terms of timed trace inclusion:

$$i \ \mathsf{tioco}\ s \ \text{ iff } \ \mathsf{TTr}(i) \subseteq \mathsf{TTr}(s).$$

We can also use the environment-relativized timed input-output conformance relation rtioco [LMN04].

2.4 Test purpose

This paper aims to conduct targeted rather than comprehensive testing of whether an IMP conforms to a SPEC. This means that we should have in mind a test purpose. In this paper, we use a "control:"-prefixed ACTL formula to specify a test purpose for a control problem (or game problem). For example, "control: A◇ `SUT.Bright`" says that we can always manage to reach the goal location `SUT.Bright` by choosing to offer appropriate inputs or to delay at appropriate moments in time, no matter how the IMP behaves.

2.5 Test hypotheses

For the purpose of proving the soundness and completeness properties of our test method, we assume that the system implementation IMP can be modeled by a TIOTS and that it has the same sets of input actions Act_{in} and output actions Act_{out} as the SPEC. Furthermore, the IMP is assumed to be deterministic and controllable, i.e., it is output-urgent and has isolated outputs. This is reasonable since IMP is usually more deterministic than SPEC.

3 Testing with winning strategies

3.1 The testing framework

Fig. 3 is the framework of testing with winning strategies. The inputs to Uppaal-Tiga are the TIOGA models of the plant and the controller, and the test purpose which is specified as an ACTL formula. The output from Uppaal-Tiga is a winning strategy. Given the SPEC models, the black-box implementation IMP, with the winning strategy we can do conformance testing and issue a verdict of pass or fail.

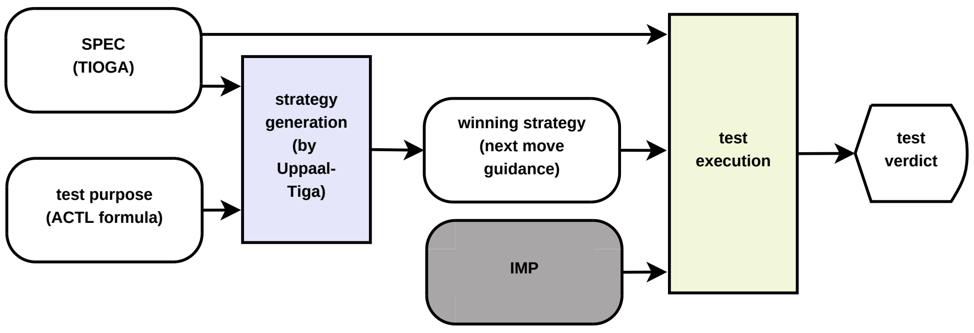

SPEC (TIOGA)
test purpose (ACTL formula)
strategy generation (by Uppaal-Tiga)
winning strategy (next move guidance)
IMP
test execution
test verdict

Figure 3: Testing with winning strategies.

3.2 Generating winning strategy

The key idea of our test method is to use a winning strategy as a test case. A reachability control problem is that given a TIOGA $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ and a set of goal states $K \subseteq L \times \mathbb{R}^X$ of its corresponding TIOTS, we should find a winning strategy f such that $\mathcal{S}$ supervised by f can reach some states in K. The given test purpose φ gives rise to K, and they are used to synthesize f.

We view the reachability control problem $(\mathcal{S}, K)$ as a game problem. A (finite or infinite) run of $\mathcal{S}$ $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_{n+1}$ is *winning* if $\exists k \geq 0 \,.\, (s_k \in K)$. The set of all winning runs in $\mathcal{S}$ that start from state s is denoted by $\mathsf{WinRuns}(s, \mathcal{S}, K)$. Winning runs in the underlying TIOTS are defined similarly.

A *strategy* f is a function that during the course of the timed game constantly gives information as to what the player should do in order to win the game [MPS95]. At a given state of the run, the player can be guided either to do a particular controllable action (i.e., to offer an input to the plant), or to do nothing at this moment in time and just wait (denoted by "λ").

Definition 7 (state-based strategy)**.** *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA, and let $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$ be the TIOTS of $\mathcal{S}$. A* state-based strategy *over $\mathcal{S}$ is a partial function:*

$$f : S \rightarrow (Act_c \cup \{\lambda\}).$$

□

Definition 8 (supervised run)**.** *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA, and f be a state-based strategy over $\mathcal{S}$. Let s be a state in the TIOTS of $\mathcal{S}$. The* f-supervised runs *of $\mathcal{S}$ from s is a subset* $\mathsf{SupRuns}(s, f) \subseteq \mathsf{Runs}(s, \mathcal{S})$ *defined inductively as:*

- $s \in \mathsf{SupRuns}(s, f)$;
- $\sigma' = (\sigma \xrightarrow{e} s') \in \mathsf{SupRuns}(s, f)$ *if,* $\sigma \in \mathsf{SupRuns}(s, f)$, $\sigma' \in \mathsf{Runs}(s, \mathcal{S})$ *and one of the following three conditions holds:*
  - $e \in Act_u$;
  - $e \in Act_c$ *and* $e = f(last(\sigma))$;
  - $e \in \mathbb{R}_{\geq 0}$ *and* $\forall e' \in [0, e) \,.\, \exists s'' \in S \,.\, ((last(\sigma) \xrightarrow{e'} s'') \wedge (f(s'') = \lambda))$;
- $\sigma \in \mathsf{SupRuns}(s, f)$ *if σ is an infinite run whose finite prefixes are all included in* $\mathsf{SupRuns}(s, f)$. □

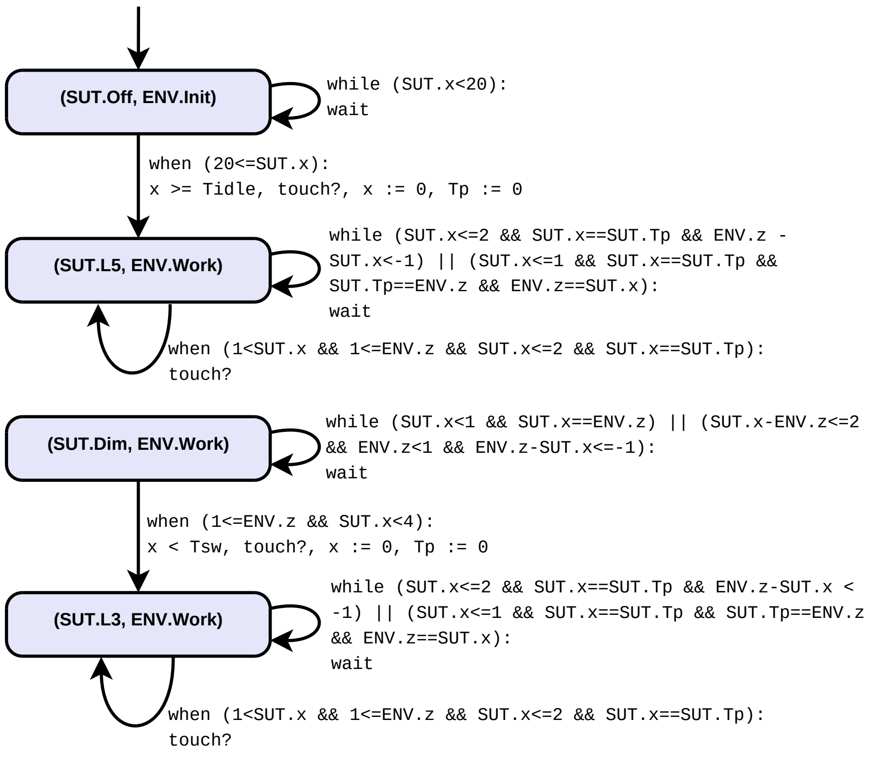
For a reachability game with $K \subseteq L \times \mathbb{R}^X$, a *maximal run* σ is either an infinite run, or a finite run such that either $last(\sigma) \in K$, or $(last(\sigma) \notin K) \wedge ((last(\sigma) \xrightarrow{\alpha}) \Rightarrow (\alpha = 0))$. We denote the set of all maximal runs from state s as $\mathsf{MaxRuns}(s)$.

Let $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_{n+1}$ be a run of TIOGA $\mathcal{S}$, and K be a set of goal states. Let $index(s_i) = i$ be the index of state s_i. If σ is a maximal run, then σ is *losing* if $\forall 0 \leq k \leq min\{index(last(\sigma)), \infty\} \,.\, (s_k \notin K)$.

Definition 9 (winning strategy). *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA, and f be a state-based strategy over $\mathcal{S}$ towards winning states K. Let s be a state in the TIOTS of $\mathcal{S}$. We say f is* winning *from state s if* $\mathsf{MaxRuns}(s) \cap \mathsf{SupRuns}(s, f) \subseteq \mathsf{WinRuns}(s, \mathcal{S}, K)$*. If f is winning from s_0, then f is a* winning strategy *for $\mathcal{S}$.* □

A strategy being winning means that if the controller acts strictly according to what the strategy suggests, then the test purpose will be enforced, no matter how the plant behaves.

Fig. 4 shows a state-based winning strategy for the user of the smart lamp example towards the test purpose "control: A◇ `SUT.Bright`". It is automatically generated by UPPAAL-TIGA.



(SUT.Off, ENV.Init)
while (SUT.x<20):
wait
when (20<=SUT.x):
x >= Tidle, touch?, x := 0, Tp := 0
(SUT.L5, ENV.Work)
while (SUT.x<=2 && SUT.x==SUT.Tp && ENV.z - SUT.x<-1) || (SUT.x<=1 && SUT.x==SUT.Tp && SUT.Tp==ENV.z && ENV.z==SUT.x):
wait
when (1<SUT.x && 1<=ENV.z && SUT.x<=2 && SUT.x==SUT.Tp):
touch?
(SUT.Dim, ENV.Work)
while (SUT.x<1 && SUT.x==ENV.z) || (SUT.x-ENV.z<=2 && ENV.z<1 && ENV.z-SUT.x<=-1):
wait
when (1<=ENV.z && SUT.x<4):
x < Tsw, touch?, x := 0, Tp := 0
(SUT.L3, ENV.Work)
while (SUT.x<=2 && SUT.x==SUT.Tp && ENV.z-SUT.x < -1) || (SUT.x<=1 && SUT.x==SUT.Tp && SUT.Tp==ENV.z && ENV.z==SUT.x):
wait
when (1<SUT.x && 1<=ENV.z && SUT.x<=2 && SUT.x==SUT.Tp):
touch?

Figure 4: An example winning strategy for the smart lamp user (Fig. 2(b)).

Note that there may exist more than one winning strategy for the same TIOGA model and test purpose. We use $\mathsf{Strategy}(\mathcal{S}, \varphi)$ to denote the set of all winning strategies for TIOGA $\mathcal{S}$ and test purpose φ.

3.3 Test execution

Definition 10 (test execution). *Let SPECS be the set of system specifications, F be the set of winning strategies, and IMPS be the set of system implementations.*

A test execution *is defined as a function:*

$$T : \mathit{SPECS} \times F \times \mathit{IMPS} \rightarrow \{\mathsf{pass}, \mathsf{fail}\}.$$

□

The basic idea of test execution towards reachability test purposes is to incrementally build a test run by constantly consulting the winning strategy and the SPEC model (see Algorithm 3.1). If an occurred output is not allowed by the specification model according to tioco, then we report fail, otherwise after reaching a goal state we report pass.

Since the winning strategy guarantees that a certain goal state must be reached, this indicates that the while-loop of Algorithm 3.1 must terminate.

Algorithm 3.1 TestExec_R($\mathcal{S}$, $\mathcal{I}$, K, f)

Input: TIOGA specification $\mathcal{S}$, system implementation $\mathcal{I}$, set K of goal states, and state-based winning strategy f;

Output: test verdict pass or fail, and test run σ;

Algorithm:

```
1: $\sigma := \langle\rangle$; /* the test run is initially an empty trace */
2: while ($\sigma \notin \mathsf{WinRuns}(s_0, \mathcal{S}, K)$) do   /* $s_0$ is the initial state */
3:    case $f(last(\sigma))$ of
4:       "input $i$":
5:          send $i$ to $\mathcal{I}$;
6:          $\sigma := \sigma \cdot i$;
7:       "delay $d$":
8:          if output $o$ occurs at $d' \leq d$ then
9:             $\sigma := \sigma \cdot d'$;
10:            if $o \notin \mathsf{Out}(s_0\ \mathsf{After}\ \sigma)$ then
11:               return("fail"); /* non-conforming behavior */
12:            else
13:               $\sigma := \sigma \cdot o$;
14:         else
15:            $\sigma := \sigma \cdot d$;
16:   esac
17: endwhile
18: return("pass"). /* good state reached */
```

Let $\mathcal{S}$ be a TIOGA specification, $\mathcal{I}$ be a TIOTS implementation, K be the set of goal states, and f be a winning strategy. We use $[\![\mathsf{TestExec\_R}(\mathcal{S}, \mathcal{I}, K, f)]\!]$ to denote the set of all (passing and failing) test runs under this configuration.

3.4 Soundness and completeness

In conformance testing, the soundness property says that if there exists a failing test run, then the implementation indeed does not comply with the specification.

Lemma 1. *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA specification with $Act = Act_{in} \cup Act_{out}$, $\mathsf{TIOTS}(\mathcal{S})$ be its corresponding TIOTS, $\mathcal{I} = (Z, z_0, Act_{in}, Act_{out}, \rightarrow)$ be a TIOTS implementation, and φ be a reachability test purpose for $\mathcal{S}$. Then $(\mathcal{I}\ \mathsf{tioco}\ \mathsf{TIOTS}(\mathcal{S})) \Rightarrow \forall f \in \mathsf{Strategy}(\mathcal{S}, \varphi)\,.\,\forall \sigma \in \mathsf{SupRuns}(z_0, f)\,.\,(\sigma\ is\ winning)$.*

Proof. (sketch). Let $\mathsf{TIOTS}(\mathcal{S}) = (S, s_0, Act_{in}, Act_{out}, \rightarrow)$. By $(\mathcal{I}\ \mathsf{tioco}\ \mathsf{TIOTS}(\mathcal{S}))$ we know that $z_0\ \mathsf{tioco}\ s_0$. It follows that $\mathsf{TTr}(z_0) \subseteq \mathsf{TTr}(s_0)$. Let f be an arbitrary winning strategy for φ, and let σ be an arbitrary f-supervised run starting from z_0. It is obvious that σ is also an f-supervised run starting from s_0. Because s_0 is the initial semantic state of $\mathcal{S}$ and f is a winning strategy, according to Algorithm 3.1, σ is a winning run. □

Theorem 1 (soundness)**.** *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA specification with $Act = Act_{in} \cup Act_{out}$, $\mathsf{TIOTS}(\mathcal{S})$ be its corresponding TIOTS, $\mathcal{I} = (I, i_0, Act_{in}, Act_{out}, \rightarrow)$ be a TIOTS implementation, and φ be a reachability test purpose. Then $\exists f \in \mathsf{Strategy}(\mathcal{S}, \varphi)\,.\,\exists \sigma \in \mathsf{SupRuns}(z_0, f)\,.\,(\sigma\ is\ failing) \Rightarrow (\mathcal{I}\ \cancel{\mathsf{tioco}}\ \mathsf{TIOTS}(\mathcal{S}))$.*

Proof. This theorem is the negation of Lemma 1. □

The completeness property says that if an implementation does not comply with a specification, then there must exist a failing test run. In this paper, we are conducting targeted testing with a test purpose. Therefore, given a test purpose φ that is satisfied by the specification, if the implementation does not conform to the specification w.r.t. φ, we will always be able to find a certain failing run. Hence the following theorem of (partial) completeness.

Theorem 2 (partial completeness)**.** *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA specification with $Act = Act_{in} \cup Act_{out}$, $\mathcal{I} = (I, i_0, Act_{in}, Act_{out}, \rightarrow)$ be a TIOTS implementation, φ be a test purpose such that $\mathcal{S} \models \varphi$, f be a strategy from $\mathsf{Strategy}(\mathcal{S}, \varphi)$, and $\mathcal{S}_f$ and $\mathcal{I}_f$ be the strategy-constrained behaviors of $\mathcal{S}$ and $\mathcal{I}$, respectively. Then $\forall f_1 \in \mathsf{Strategy}(\mathcal{S}, \varphi)\,.\,(\ (\mathcal{I}_{f_1}\ \cancel{\mathsf{tioco}}\ \mathcal{S}_{f_1}) \Rightarrow \exists f_2 \in \mathsf{Strategy}(\mathcal{S}, \varphi)\,.\,\exists \sigma \in [\![\mathsf{TestExec\_R}(\mathcal{S}, \mathcal{I}, K, f_2)]\!]\,.\,(\sigma\ is\ failing)\ )$.*

Proof. (sketch). By $\mathcal{I}_{f_1}\ \cancel{\mathsf{tioco}}\ \mathcal{S}_{f_1}$ we know that $z_0\ \cancel{\mathsf{tioco}}\ s_0$. By Definition 6, there exists a run $\sigma \in (Act_{in} \cup Act_{out} \cup \mathbb{R}_{\geq 0})^*$ such that $\sigma \in \mathsf{TTr}(z_0) \backslash \mathsf{TTr}(s_0)$. We generate a winning strategy f for φ. Now we define a winning strategy f' for $\mathcal{I}$ and φ such that f' has all the guidance as in f. In addition, we let f' have an extra supervised maximal run starting from z_0 such that it is exactly σ. Obviously, σ fails $\mathcal{S}$. □

3.5 Testing towards safety test purposes

The development so far (in Section 3) considers only reachability test purposes and reachability games. Given a safety test purpose (control objective) φ, we can induce a set K of “bad” states. Once a state in K is encountered during test execution, the run is declared to be losing. Similar to $\mathsf{WinRuns}(s, \mathcal{S}, K)$, we define the set of all losing runs in $\mathcal{S}$ that start from state s as $\mathsf{LoseRuns}(s, \mathcal{S}, K)$.

A winning strategy for a safety control problem will guide the controller to always avoid any of the bad states. For reactive real-time systems, it is no surprise that such guidance will lead to infinite executions due to the possible looping structures in the winning strategies.

Procedure 3.2 shows how to validate[4] an implementation against its specification model according to a winning strategy that is generated for a safety test purpose. The major differences from Algorithm 3.1 are shown in boldface font.

Procedure 3.2 TestExec_S($\mathcal{S}$, $\mathcal{I}$, K, f)

Input: TIOGA specification $\mathcal{S}$, system implementation $\mathcal{I}$, set K of “**bad**” states, and state-based winning strategy f;
Output: test verdict fail (if ever), and test run σ;
Procedure:

```
1:  $\sigma := \langle\rangle$; /* the test run is initially an empty trace */
2:  while ($\sigma \notin$ LoseRuns($s_0, \mathcal{S}, K$)) do   /*$s_0$: the init state*/
3:     case $f(last(\sigma))$ of
4:        “input $i$”:
5:           send $i$ to $\mathcal{I}$;
6:           $\sigma := \sigma \cdot i$;
7:        “delay $d$”:
8:           if output $o$ occurs at $d' \leq d$ then
9:              $\sigma := \sigma \cdot d'$;
10:             if $o \notin$ Out($s_0$ After $\sigma$) then
11:                return(“fail”); /* non-conforming behavior */
12:             else
13:                $\sigma := \sigma \cdot o$;
14:          else
15:             $\sigma := \sigma \cdot d$;
16:    esac
17: endwhile
18: return(“fail”). /* bad state encountered */
```

[4]Testing by definition is a finite experiment. Since a winning strategy for a safety game may lead to infinite execution, this validation activity in a strict sense is not testing. This validation also distinguishes itself from passive testing (a.k.a. run-time verification), because it “actively” offers input stimuli to the implementation based on the guidance of the strategy.

Similar to Section 3.4, we can prove the soundness and the (partial) completeness properties of testing towards safety test purposes.

4 Case study

We consider a simple leader election protocol (LEP) [Lam05b] (more details in the Appendix), which is essentially a distributed consensus algorithm with timing constraints. The idea is to elect the protocol node with the lowest network address as the "leader" by using message passing.

We model the problem as two parts: one TIOGA for the plant (or "SUT") which consists of an arbitrary node; and two TIOGAs for the controller which consists of a buffer with certain capacity and the simulated chaotic environment that consists of all the other nodes. The plant TIOGA has uncontrollable actions in the sense that in the plant node a *timeout*! event can be produced at any point of a time frame after the node has been waiting for a certain period of time without receiving any "useful" messages.

We defined the following test purposes:

- TP1: control: $\mathsf{A}\Diamond$ (`SUT.betterInfo` $==$ `1`) && `SUT.forward`;
- TP2: control: $\mathsf{A}\Diamond$ `forall(i : BufferId)(inUse[i] == 1)`; and
- TP3: control: $\mathsf{A}\Diamond$ `forall(i : BufferId)(inUse[i] == 1)` && `SUT.idle`,

where, e.g., test purpose TP1 intuitively means that given an arbitrary protocol node $node_i$, if we have full control over the other nodes and the buffer, then we can always manage to let $node_i$ reach a state where it is ready to forward "useful" messages, no matter how $node_i$ behaves (timeouts).

All the above three test purposes are checked to be true using Uppaal-Tiga. We carried out the strategy generation experiments on an application server with dual-core 2.4GHz CPU, 4096MB RAM, and Suse Linux Enterprise Desktop. Table 1 presents the performance results for these test purposes with different protocol parameter settings, where / means "out of memory". The time and memory columns represent the time overheads and the memory consumptions, respectively. Each sub-column corresponds to one parameter configuration, where n means that there are n nodes in the protocol, and there is a message buffer of size n, and the maximum distance between any two nodes is limited to $(n-1)$.

As can be seen from Table 1, winning strategy generation for the LEP protocol with up to 7 nodes takes less than 8 minutes, and the memory consumption is not well beyond out expectation considering the complexity of the problem.

Table 1: Strategy generation for LEP protocol.

	Time (s)						**Memory** (MB)					
	n=3	4	5	6	7	8	n=3	4	5	6	7	8
TP1	0.03	0.14	0.7	3.1	11.1	33.5	0.1	4	9	28	85	242
TP2	0.81	2.13	8.4	67.1	452.0	/	11.2	33	88	462	2977	/
TP3	0.89	2.79	25.9	73.2	453.8	/	11.9	40	289	578	3015	/

5 Conclusions

We examine the problem of black-box conformance testing of uncontrollable real-time systems using a game-theoretic approach. We model the systems in question using timed I/O game automata and specify the test purposes with ACTL formulas. With the help of a recently developed timed game solver, we can do testing based on winning strategies. Experimental results of a leader election protocol indicate that this approach is viable and computationally feasible. This opens up a new possibility for testing TA-modeled timed systems that have output uncertainty and timing uncertainty of outputs, which are previously thought of as somewhat under-specified.

Future work includes: (1) to generalize state-based strategy to history-based strategy; (2) to build a fully automated strategy-based testing environment, where an important issue is efficient strategy representation; (3) to evaluate strategy-based test effectiveness in terms of e.g. fault detecting capability; (4) if there does not exist a winning strategy, we hope to make a small "retreat" by doing cooperative testing; (5) strategy-based testing with partial observability.

Appendix: The leader election protocol model

We carried out case studies on a leader election protocol (LEP) [Lam05b, Lam05a]. As a small yet complex problem, it is illustrative of many aspects of complex software systems such as: concurrency, real-time (node timeouts and message transmission delays), non-determinism (timing uncertainties of node timeouts and message deliveries), communications (handshake synchronization and shared variable communication), and lossy media (loss of messages in transmission due to limited capacity media).

Protocol description

The leader election problem is abstracted from a class of consensus algorithms that are used in distributed environments such as mobile ad-hoc networks. The original version of LEP tries to construct a spanning tree for a network of nodes which are connected by some edges (direct network links), and after that the leader node maintains its leadership by periodically sending out the "I am leader" messages that are forwarded to all nodes in the network [Per85]. In this paper we study a simple version of it: for a network of nodes with fixed topology (i.e., without node failing or link breaking), how to achieve consensus on "which node should be elected as the leader" [Lam05b, Lam05a].

In the LEP protocol each node has an address (a natural number). The goal of the protocol is to identify the node with the lowest address in the network, and elect that node as the "leader". In order for the protocol to be correct, the electee should be with a unanimous approval, i.e., all connected nodes should have elected the same leader. According to the protocol, each node maintains information about which node it believes to be the leader and the number of hops (network links) from itself to the (believed) leader. Such information of node i is thus denoted by a pair $n_i = (\mathit{leader}, \mathit{hops})$.

The protocol transmits messages in the form $m = (\mathit{source}, \mathit{destination}, \mathit{leader}, \mathit{hops})$ among the nodes, where

- *source* is the address of the message-sending node;
- *destination* is the address of the message-receiving node;
- *leader* is the address of the node which the message-sending node believes to be the leader; and
- *hops* is the number of hops between the message-sending node and the (believed) leader.

Whenever a node i receives a message m, it compares $m.\mathit{leader}$ and $m.\mathit{hops}$ with the local information n_i that it is currently holding. If $m.\mathit{leader} < n_i.\mathit{leader}$ or ($m.\mathit{leader} = n_i.\mathit{leader}$ and $m.\mathit{hops} < n_i.\mathit{hops} - 1$), then n_i is updated according to m. Otherwise m is simply discarded by node i. Once n_i is updated, node i immediately sends a message $(i, j, n_i.leader, n_i.hops)$ to each of its neighbor nodes j, except for the source node of message m that triggers this update.

In the protocol the message transmissions are not instantaneous. Rather there is an upper time bound MSG_DELAY on the delivery of a message. The timing aspect also implies that messages might be reordered during transmissions.

The protocol has a timeout mechanism. Let *Timeout* be an integer variable. If for more than *Timeout* time units a node has not yet received any "good" messages (i.e., messages which contain "better" information than what that node already holds, or alternatively, messages which will not be discarded) since the last reception of a "good" message, then a timeout will happen within TIMEOUT_DELAY time units (i.e. TIMEOUT_DELAY represents a reaction delay). Similarly, a timeout will happen within TIMEOUT_DELAY time units if no "good" message is received after *Timeout* time units since the last timeout. This means that there is actually a time frame [*Timeout*, *Timeout*+TIMEOUT_DELAY], during which a timeout should happen. Initially, *Timeout* is assigned a constant value INIT_TIMEOUT. Upon receiving "good" messages, *Timeout* is set to (INIT_TIMEOUT + TIMEOUT_DELAY + $n_i.hops$ ∗ MSG_DELAY), which ensures the algorithm to achieve stability [Lam05b]. Once a timeout happens in a node, the node will immediately elect itself as the new leader and send an update message to each of its neighbors.

In this case study, we let INIT_TIMEOUT be 10 time units, TIMEOUT_DELAY be 5 time units, and M_DELAY be 3 time units.

System models

Fig. 5 shows the TIOGA model of one protocol node, and Fig. 6 shows the simulated environment for this chosen node, which consists of the other nodes (Fig. 6(a)[5]) and the message buffer (Fig. 6(b)). In Fig. 5, the output *timeout*! is uncontrollable (in dashed line). Furthermore, there is timing uncertainty of outputs in the TIOGA, because *timeout*! can occur at any point of time as specified by the transition guard. Therefore, the node TIOGA is uncontrollable.

The global declaration of the LEP system is shown in Listing 1, and the local declaration of the chosen `Node` is shown in Listing 2.

[5]In this sub-figure, *iutId* is declared as a parameter of the "ChaoticEnv" TA template, i.e., "const NodeId iutId". When the "ChaoticEnv" template is instantiated, *iutId* will take the value of the Id of the chosen `Node` (i.e., the node of Fig. 5).

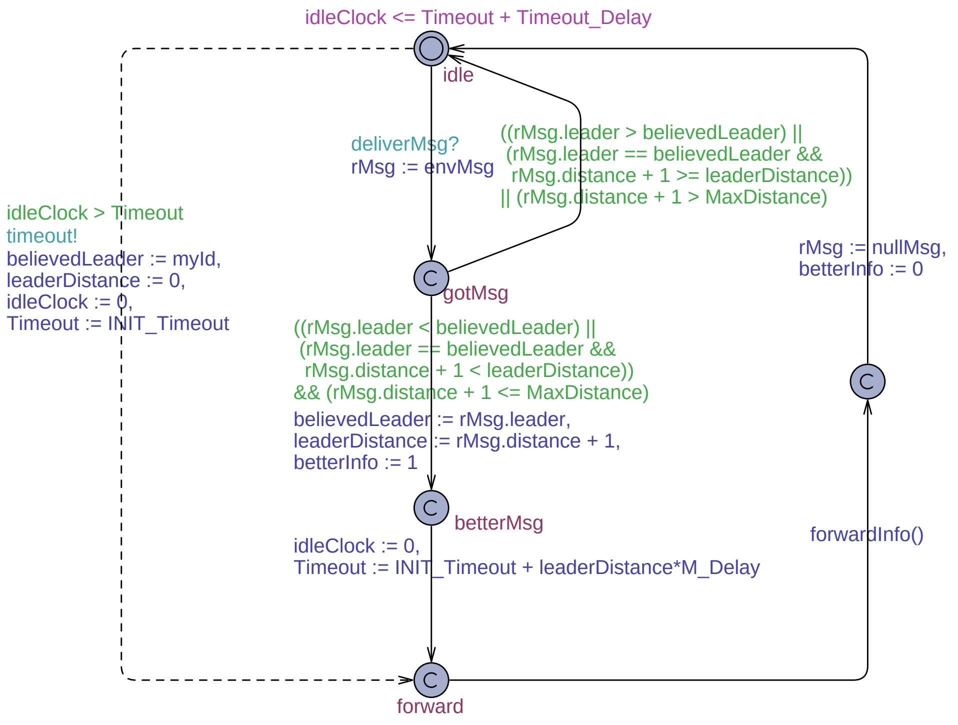
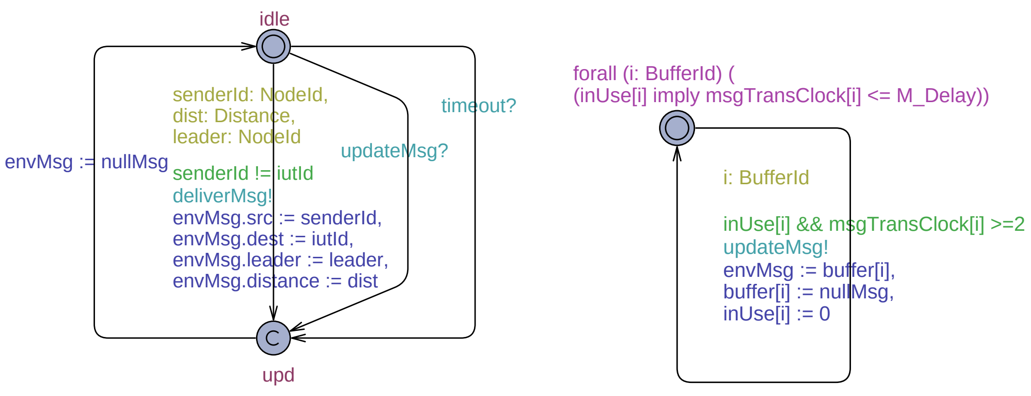

idleClock <= Timeout + Timeout_Delay
idle
deliverMsg?
rMsg := envMsg
((rMsg.leader > believedLeader) ||
(rMsg.leader == believedLeader &&
rMsg.distance + 1 >= leaderDistance))
|| (rMsg.distance + 1 > MaxDistance)
idleClock > Timeout
timeout!
believedLeader := myId,
leaderDistance := 0,
idleClock := 0,
Timeout := INIT_Timeout
rMsg := nullMsg,
betterInfo := 0
gotMsg
((rMsg.leader < believedLeader) ||
(rMsg.leader == believedLeader &&
rMsg.distance + 1 < leaderDistance))
&& (rMsg.distance + 1 <= MaxDistance)
believedLeader := rMsg.leader,
leaderDistance := rMsg.distance + 1,
betterInfo := 1
betterMsg
forwardInfo()
idleClock := 0,
Timeout := INIT_Timeout + leaderDistance*M_Delay
forward

Figure 5: The TIOGA for a chosen protocol Node.



(a) the ChaoticEnv component (b) the Buffer component

Figure 6: The models of the simulated environment.

Listing 1: UPPAAL global declarations for the LEP protocol.

```
const int  MaxNodeId = 3; // so the number of nodes is (MaxNodeId+1)
const int  MaxBufferId = 3; // so the  number of slots  is  (MaxBufferId+1)
const int  MaxDistance = 3;

typedef  int  [0, MaxNodeId] NodeId;
typedef  int  [0, MaxBufferId]  BufferId ;
typedef  int  [0, MaxDistance] Distance;

const int  INIT_Timeout = 10;
const int  Timeout_Delay = 5;
const int  M_Delay = 3;
int  Timeout = INIT_Timeout;

typedef  struct  {
  NodeId src;
  NodeId dest;
  NodeId leader;
  Distance  distance ;
} Message;

const Message nullMsg = {0,0,0,0};
Message envMsg; // Message from Environment

Message buffer[ MaxBufferId+1];
bool inUse[ MaxBufferId+1];

clock  msgTransClock[MaxBufferId+1]; // time that a msg. has been in  transmission

chan deliverMsg;  // ChaoticEnv -- >Node
chan updateMsg;  // Buffer  -- >ChaoticEnv
chan timeout;      // Node -- >ChaoticEnv
```

Listing 2: UPPAAL local declarations for the chosen protocol `Node`.

```
NodeId believedLeader = myId; // ''const NodeId myId'' is the parameter of the
                               // '' Node'' TA template
Distance leaderDistance = 0;
bool betterInfo = 0;
Message rMsg = nullMsg; // The received message
clock idleClock ; // How long since last reception of a '' good'' massage, or
                  // since a last '' timeout''

void forwardInfo (){
  int forwardee = 0;
  int slot = 0; // find an available message slot in the buffer
  for (forwardee = 0; forwardee <= MaxNodeId; forwardee++) {
    if ( (forwardee != myId) && (!(betterInfo && forwardee == rMsg.src)) ) {
    // do not send to itself ; and do not forward to the original sender
      for (slot = 0; slot <= MaxBufferId && inUse[slot]; slot++) {
        // do nothing;
      }
      if (slot <= MaxBufferId) { // slot is the available position in the buffer
        buffer [slot ]. src = myId;
        buffer [slot ]. dest = forwardee;
        buffer [slot ]. leader = believedLeader;
        buffer [slot ]. distance = leaderDistance;
        inUse[slot ] = 1;
        msgTransClock[slot] = 0; // now a message begins its transmission
      }
      else {
        // no available slot in the buffer ! simply drop the message !
      }
    }
  } // end '' for ''
}
```

Paper D:
Cooperative Testing of Timed Systems

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen
Center for Embedded Software Systems (CISS)
Department of Computer Science
Aalborg University, Denmark

Abstract

This paper deals with targeted testing of real-time embedded systems. The testing activity is viewed as a game between the tester and the system under test (SUT) towards a given test purpose (winning objective). The SUT is modeled using timed game automata (TGA) and the test purpose is specified as an ACTL formula. We employ a timed game solver UPPAAL-TIGA to check if the timed game is solvable, and if yes, to generate a winning strategy and use it for black-box conformance testing of the SUT.

Specifically, we show that in case the game solving yields a negative result, we can still possibly test the SUT against the test purpose. In this case, we use UPPAAL-TIGA to generate a *cooperative* winning strategy. The testing process will continue as long as the SUT reacts to the tester stimuli in a cooperative manner. In this way we can hopefully arrive at a certain state in the "surely winning" zone of the game state space, from which cooperation from SUT is no longer needed. We present an operational framework of cooperative winning strategy generation, test case derivation and test execution. The test method is proved to be sound and complete. Preliminary experimental results indicate that this approach is applicable to non-trivial timed systems.

Keywords: Real-Time Systems, Timed Game Automata (TGA), Test Purposes, Cooperative Winning Strategies, Cooperative Testing

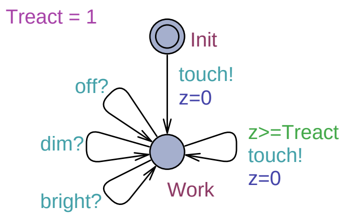
1 Introduction

In the field of model-based testing of real-time systems [CL95, ENDKE98, HNTC99, SVD01, KJM03, HLN+03, LMN04, BB04, KT04, NR05], a considerable proportion of efforts [ENDKE98, HNTC99, SVD01, KJM03, HLN+03, LMN04, BB04, KT04] employ timed automata (TA) [AD94] or timed transition systems (TTS) to model the systems in question. Among them some make the assumptions that the system TA model is *output-urgent* and has *isolated outputs* [SVD01, HLN+03]. "Output-urgent" means that if the system can produce an output, the output should be produced immediately. "Isolated output" means that anytime when the system can produce an output, it cannot accept inputs or produce a different output at that time. These two assumptions together with the *determinism* assumption contribute to the testability [SVD01] of timed automata by ensuring that given a timed input sequence fragment there is no more than one possible output emitted at a precise point in time. To put differently, they make it possible for an environment (tester) to "drive" a timed automaton through all of its possible transitions.

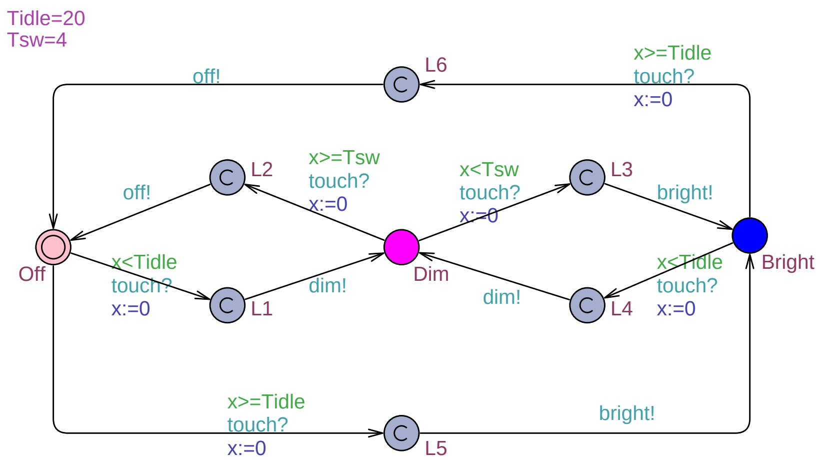
However, in many cases the assumptions of "output urgent" and "isolated outputs"are unnecessarily strong. For example in the simple Smart Lamp problem [HLN+03], in order to satisfy the requirements of output-urgency and isolated outputs (see Fig. 1(b)), it must be ensured that every output response of the Lamp is carefully designed (pre-programmed) and is thus perfectly predictable. Constructing this kind of TA models is too expensive in the sense that we should have one TA node exclusively for producing each output, and we should have strict timing (i.e., no tolerance) of each output.

In this paper we aim to cancel the assumptions of isolated outputs and output-urgency, and present a test method for *uncontrollable* timed system models, i.e., system models with *output uncertainty* and *timing uncertainty of outputs*. By "output uncertainty" (or uncontrollable outputs) we mean that during the course of system under test (SUT)/tester interactions, it is the SUT rather than the tester that determines whether an output will be produced, and if yes, which of the several possible outputs will be produced. By "timing uncertainty of outputs" we mean that the SUT can produce an output during a certain time interval rather than only at a fixed time point, or in other words, the exact timing of outputs is unpredictable by the tester. The benefits of permitting uncontrollable behavior in the system models include that: (1) it allows the implementors some freedom; (2) it provides the tester with high-level or abstract requirements; and (3) it usually leads to more natural and more succinct models.

Systems with output uncertainty and timing uncertainty of outputs can be modeled as *timed game automata* (TGAs) [MPS95], which is a variant of TA with their actions partitioned into controllable and uncontrollable ones. For example, Fig. 2(a) is a TGA of the Lamp, where solid lines carry controllable actions (input stimuli from user to lamp) and dashed lines uncontrollable actions (output



(a) The user.



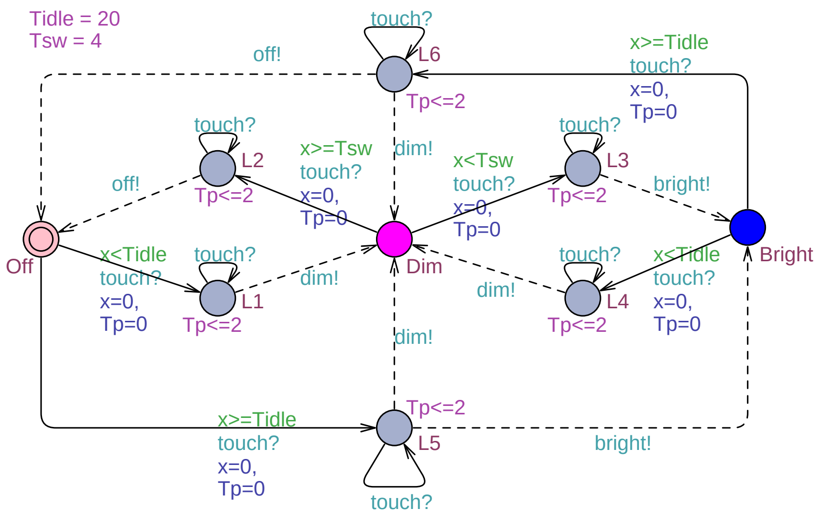
(b) "Controllable" TA of the Lamp.

Figure 1: A smart lamp example.

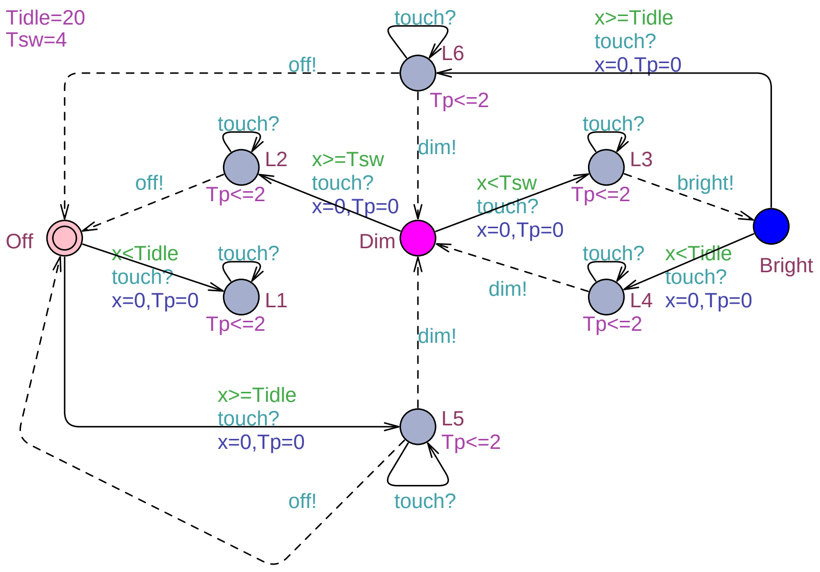
responses from lamp to user).

In a timed control problem, a control program (or "controller", e.g., Fig. 1(a)) actively offers inputs to and passively observes outputs from a system under control (or "plant", e.g., Fig. 2(a)). A *run* of the system involves a sequence of controller-chosen input stimuli and plant-chosen output reactions aiming to satisfy a given control objective (e.g., "location `Bright` can always be eventually reached"). Therefore it can be viewed as a timed game where the controller acts as a player, and the plant acts as the opponent (adversary). For a given control objective we can possibly synthesize a winning strategy, guided by which the control program ensures that the control objective will be enforced, no matter how the plant behaves.

The problem of dense-time controller synthesis has been solved using backward fix-point computation [MPS95]. As an improvement, a truly on-the-fly algorithm [CDF$^+$05] is proposed. This algorithm has been implemented in the timed game solver UPPAAL-TIGA [BCD$^+$07], which checks whether a user-specified winning objective (test purpose) can be enforced on a TGA, and if so, it efficiently synthesizes a winning strategy for that winning objective. Specifically, in this paper we address the problem that in case an affirmative winning objective is checked to be *unenforceable* on some system models (e.g., control: A$\Diamond$ `Bright`



(a) A normal smart lamp.



(b) A “problematic” smart lamp.

Figure 2: TIOGAs of the Lamp.

on Fig. 2(b)), we can make a "retreat" by relaxing the winning objective such that in order to win the game, the controller needs some cooperation from the plant, say, "location `Bright` can always be eventually reached *as long as* the lamp reacts to our moves in some desired manner". We use UPPAAL-TIGA to check whether the winning objective can be enforced in this relaxed sense, and if yes, to synthesize a *cooperative winning strategy.* Since a (cooperative) strategy provides step-by-step guidance to the controller towards the given winning objective, it can be viewed as a test and thus used for conformance testing [DLLN08b].

From a game point of view, testing of untimed systems has been discussed in [ACY95, Yan04, BGNV05], but to our knowledge no similar work for timed systems has been reported. Although strategy synthesis is inherently more expensive than some other approaches to timed testing, the idea and method proposed in this paper opens up the possibility of testing uncontrollable TA-modeled timed systems, especially when the user-specified test purpose happens not to be enforceable on the system model.

2 Test setup

In this paper we aim to test whether a black-box system implementation IMP complies with its specification model SPEC with respect to some given test purpose. As illustrated in Fig. 3, there are three steps in our testing framework: game strategy generation, test case generation and test execution.

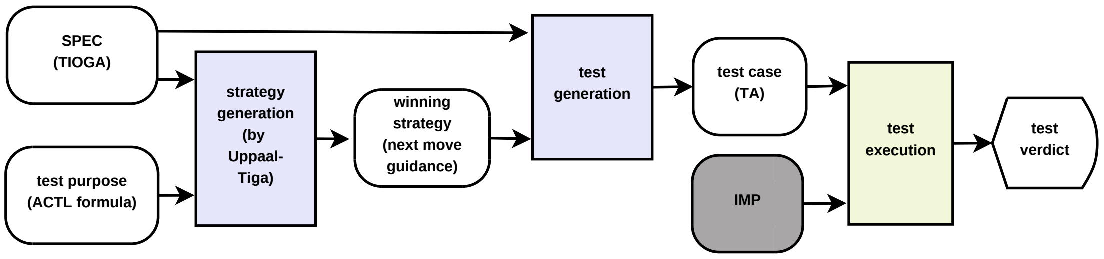

SPEC (TIOGA)
test purpose (ACTL formula)
strategy generation (by Uppaal-Tiga)
winning strategy (next move guidance)
test generation
test case (TA)
IMP
test execution
test verdict

Figure 3: The framework of strategy-based testing.

2.1 Timed I/O Game Automaton

Let X be a finite set of real-valued clocks, and $\mathcal{C}(X)$ be the set of constraints generated by grammar

$$\varphi ::= x \bowtie k \mid x - y \bowtie k \mid \varphi \wedge \varphi,$$

where $k \in \mathbb{N}_{\geq 0}$, $x, y \in X$, and $\bowtie \in \{<, \leq, =, \geq, >\}$.

A *timed automaton* (TA) [AD94] is a tuple $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ where:

- L is a finite set of locations;
- $l_0 \in L$ is the initial location;
- Act is a finite set of actions;
- X is a finite set of real-valued clocks;
- $E \subseteq L \times \mathcal{C}(X) \times Act \times 2^X \times L$ is a finite set of edges; and
- $Inv : L \rightarrow \mathcal{C}(X)$ associates invariants to locations. □

In a *timed game automaton* (TGA) [MPS95], the actions Act are partitioned into controllable ones and uncontrollable ones, i.e., $Act = Act_c \cup Act_u$ and $Act_c \cap Act_u = \emptyset$.

If we make a further assumption that all output actions (Act_{out}) are uncontrollable and all input actions (Act_{in}) are controllable, i.e., $Act_u = Act_{out}$ and $Act_c = Act_{in}$, then we have a *timed I/O game automaton* (TIOGA).

This paper uses the simple Smart Lamp problem [HLN+03] as an example. Fig. 1(a) is a TA of the user (the "controller"). Fig. 2(b) is a "problematic" TIOGA of the smart Lamp (the "plant"), where controllable actions (in solid lines) model the inputs from the controller to the plant, and uncontrollable actions (in dashed lines) model the outputs from the plant to the controller. The user interacts with the Lamp by touching a touch-sensitive pad. In Fig. 2(b), there are three brightness levels for the Lamp: `Off`, `Dim` and `Bright`. The Lamp is initially in location `Off`. There are uncontrollable behavior in `L2`, `L3`, `L4`, `L5` and `L6`.

The semantics of a TIOGA $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ is defined as a *timed I/O transition system* (TIOTS) $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$, where:

- $S \subseteq L \times \mathbb{R}^X$ is the set of semantic states of location and clock vectors;
- $s_0 = (l_0, \overline{0})$ is the initial state;
- Act_{in} and Act_{out} are the sets of input and output actions, respectively; and
- $\rightarrow \subseteq S \times (Act_{in} \cup Act_{out} \cup \mathbb{R}_{\geq 0}) \times S$ satisfies the sanity constraints of time determinism and time additivity. □

Let $s \in S$ and $\alpha \in (Act \cup \mathbb{R}_{\geq 0})$. If $\exists s' \in S \,.\, s \xrightarrow{\alpha} s'$, we write $s \xrightarrow{\alpha}$. Here α can be extended to strings of actions and time delays.

A *timed trace* $\sigma \in (Act \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 a_2 \ldots a_k d_{k+1}$. We define the set of timed traces of state s as:

$$\mathsf{TTr}(s) = \{\sigma \in (Act \cup \mathbb{R}_{\geq 0})^* \mid s \xrightarrow{\sigma}\}.$$

For a state s and a timed trace σ, we define the set of states that can be reached after σ as:

$$s \;\mathsf{After}\; \sigma = \{s' \mid s \xrightarrow{\sigma} s'\}.$$

If the above set is a singleton, then we denote it as a single target state s'.

The set of (immediately) observable outputs or delays at state s is defined as:

$$\mathsf{Out}(s) = \{a \in (Act_{out} \cup \mathbb{R}_{\geq 0}) \mid s \xrightarrow{a}\}.$$

The definitions of After and Out can be extended to sets of states as usual.

A *run* of a TIOGA $\mathcal{S}$ is a timed trace in its TIOTS. We use $\mathsf{Runs}(s, \mathcal{S})$ to denote the set of all runs of $\mathcal{S}$ that start from $s \in S$. Specifically, we denote $\mathsf{Runs}(s_0, \mathcal{S})$ as $\mathsf{Runs}(\mathcal{S})$. If σ is a finite run, then $last(\sigma)$ denotes the last semantic state of σ.

In this paper we only impose the "determinism" and "strong input-enabledness" restrictions on the TIOGA model of the plant. In particular we do not require the plant model to be output-urgent (thus allowing timing uncertainty of outputs), or have isolated outputs (thus allowing output uncertainty). Such a TIOGA is called an *uncontrollable* TIOGA, and its corresponding TIOTS is called an *uncontrollable* TIOTS.

The parallel compositions of several TIOGAs or several TIOTS's can be defined in the usual manner.

2.2 Timed Conformance Relation

Definition 1 (timed input-output conformance relation, tioco [KT04]). *Let $i, s \in S$ be two states of a TIOTS. The* timed input-output conformance relation tioco *between i and s is defined as:*

$$i \;\mathsf{tioco}\; s \quad \textit{iff} \quad \forall \sigma \in \mathsf{TTr}(s).(\mathsf{Out}(i \;\mathsf{After}\; \sigma) \subseteq \mathsf{Out}(s \;\mathsf{After}\; \sigma)).$$

□

As test hypothesis we assume that the behavior of the IMP can be modeled by a TIOTS $\mathcal{I}$, which has the same sets of input actions Act_{in} and output actions Act_{out} as the specification TIOGA $\mathcal{S}$. Let the initial state of $\mathcal{I}$ be i_0, and the initial semantic state of $\mathcal{S}$ be s_0. If i_0 tioco s_0, we say that $\mathcal{I}$ is a correct implementation of the specification, denoted $\mathcal{I}$ tioco $\mathsf{TIOTS}(\mathcal{S})$. Furthermore, $\mathcal{I}$ is assumed to be deterministic and controllable.

We can also use other timed versions [KJM03, LMN04, NR05] of the ioco conformance relation [Tre99].

2.3 Test Purpose

We aim to conduct targeted rather than comprehensive testing of whether an IMP conforms to a SPEC, thus we use a test purpose [JJ05]. In this paper, we use ACTL formulas to specify test purposes. For example, "control: A$\Diamond$ `Bright`" means that we can guarantee that the Lamp can reach the goal location `Bright` by choosing to offer appropriate inputs or just to delay at appropriate moments in time, no matter how the Lamp behaves in response.

To indicate that we would consider the timed game in the *relaxed* sense, we write "E$\Diamond$ control: A$\Diamond$ `Bright`". This property means that we can guarantee that the `Bright` location can always be reached eventually, *as long as* the Lamp is willing to cooperate with us (the controller) by producing outputs in some desired manner.

3 Cooperative winning strategy

A reachability control problem is that given a TIOGA $\mathcal{S}$ and a set K of goal states in its corresponding TIOTS, we should find a game strategy f such that $\mathcal{S}$ supervised by f can reach some states in K. If a state in K is reached, then the play of the game is said to be *winning.*

A strategy f is a function that during the course of a timed game constantly gives information as to what the player (the "controller") should do in order to win the game against the opponent (the "plant"). At a given state of the run, the player can be guided either to offer a particular input and bring it to a particular state, or to do nothing at this moment in time and just wait (denoted "λ").

When a winning objective (test purpose) φ is checked to be enforceable, then there must exist a winning strategy for φ. A strategy being winning means that if the controller acts strictly according to what the strategy suggests, then the winning objective can be enforced, no matter how the plant behaves.

When a winning objective φ is checked to be unenforceable, then there does not exist a (surely) winning strategy. For example in the TIOGAs of Fig. 1(a) and Fig. 2(b), there is no winning strategy for "control: A$\Diamond$ `Bright`". An obvious counterexample is that in location `L5` (Fig. 2(b)) we might be constantly brought back to `Off`. For this negative case, we can make a "retreat" by assuming that the opponent is not too "hostile". The basic idea is that in order to reach the goal states, we hope that the opponent will to some extent react in favor of us.

The principle of playing games with a *cooperative* winning strategy is illustrated in Fig. 4. The state space of a timed game is partitioned into three areas:

- the (surely) winning zone (i..e, "safe" zone);
- the possibly winning zone; and

- the losing zone (i.e., "no-hope" zone).

The relaxed-sense test purpose in Section 2.3 asks that if the opponent is willing to cooperate, can we possibly reach a certain state in the "safe" zone, from which the goal location `Bright` is always eventually (i.e. surely) reachable?

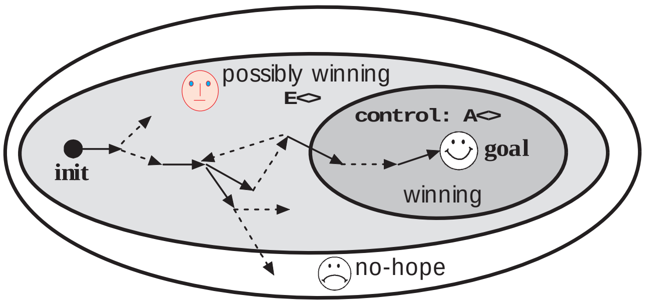


Figure 4: Playing games with cooperative winning strategies.

Definition 2 (cooperative strategy). *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA, $(S, s_0, Act_{in}, Act_{out}, \rightarrow)$ be its TIOTS, where $\rightarrow\ =\ (\rightarrow_{in} \cup \rightarrow_{out} \cup \rightarrow_d)$. A* cooperative strategy *f over $\mathcal{S}$ is defined as a partial function:*

$$f:\ S\ \rightarrow\ \{coop\}\times(\rightarrow_{in}\cup\rightarrow_{out}\cup\{\lambda\})\ \cup\ \{winning\}\times(\rightarrow_{in}\cup\{\lambda\}).$$

□

We use a projection function f_{stg} to indicate which stage ("coop"(erative) or "winning") f is currently in, and use f_{mov} to denote the suggested or desired move of f. For transition $t \in (\rightarrow \setminus \rightarrow_d)$, let $act(t)$ be the action and $tgt(t)$ be the target state. In the cooperative stage, if a strategy-desired output occurs as expected, then the game opponent is said to be *cooperating*, otherwise the strategy is violated.

Definition 3 (supervised run). *Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA, and f be a cooperative strategy over $\mathcal{S}$. Let s be a state in the TIOTS of $\mathcal{S}$. The* f-supervised runs *of $\mathcal{S}$ from s is a subset $\mathsf{SupRuns}(s, f) \subseteq \mathsf{Runs}(s, \mathcal{S})$ defined as:*

- *$s \in \mathsf{SupRuns}(s, f)$;*
- *$\sigma' = (\sigma \xrightarrow{e} s') \in \mathsf{SupRuns}(s, f)$ if, $\sigma \in \mathsf{SupRuns}(s, f)$, $\sigma' \in \mathsf{Runs}(s, \mathcal{S})$ and one of the following three conditions holds:*
  - *$e \in Act_u$ and either*
    - *$f_{stg}(last(\sigma)) = winning$; or*

  - $\cdot$ $f_{stg}(last(\sigma)) = coop$ *and* $e = act(f_{mov}(last(\sigma)))$;
- $-$ $e \in Act_c$ *and* $e = act(f_{mov}(last(\sigma)))$;
- $-$ $e \in \mathbb{R}_{\geq 0}$ *and* $\forall e' \in [0, e) \,.\, \exists s'' \in S \,.\, ((last(\sigma) \xrightarrow{e'} s'') \wedge (f_{mov}(s'') = \lambda))$;

- $\sigma \in \mathsf{SupRuns}(s, f)$ *if* σ *is an infinite run whose finite prefixes are all included in* $\mathsf{SupRuns}(s, f)$. □

Given a TIOGA $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ and a set of goal states $K \subseteq L \times \mathbb{R}^X$ of its corresponding TIOTS, let $(\mathcal{S}, K)$ be a reachability game. A *maximal run* σ is either an infinite run, or a finite run such that either $last(\sigma) \in K$, or $(last(\sigma) \notin K) \wedge ((last(\sigma) \xrightarrow{\alpha}) \Rightarrow (\alpha = 0))$. A (finite or infinite) run $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots s_n \xrightarrow{\alpha_n} \ldots$ is *winning* if $\exists k \geq 0 \,.\, (s_k \in K)$. A run σ is *losing* if σ is maximal and $\forall 0 \leq k \leq min\{index(last(\sigma)), \infty\} \,.\, (s_k \notin K)$. The set of all maximal runs that start from state s is denoted by $\mathsf{MaxRuns}(s)$, and the set of all winning runs that start from state s is denoted by $\mathsf{WinRuns}(s, \mathcal{S}, K)$.

Definition 4 (cooperative winning strategy). *Let* $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ *be a TIOGA,* f *be a cooperative strategy over* $\mathcal{S}$*, and* s *be a state in the TIOTS of* $\mathcal{S}$*. We say that* f *is* cooperatively winning *from state* s *if* $\mathsf{MaxRuns}(s) \cap \mathsf{SupRuns}(s, f) \subseteq \mathsf{WinRuns}(s, \mathcal{S}, K)$*. If* f *is cooperatively winning from* s_0*, then* f *is said to be a* cooperative winning strategy. □

For the TIOGA in Fig. 2(b) and the relaxed-sense test purpose "E◇ control: A◇ Bright", UPPAAL-TIGA automatically generates a cooperative winning strategy (Fig. 5) where the strategy-desired outputs are shown in dashed lines.

Usually there exists more than one cooperative winning strategy for the same TIOGA $\mathcal{S}$ and relaxed-sense test purpose φ. We use $\mathsf{Strategy}(\mathcal{S}, \varphi)$ to denote the set of all such cooperative winning strategies.

4 Test case generation

A test case for an uncontrollable reactive system cannot be just a sequentially preset I/O sequence. Rather it should be adaptive.

Furthermore, we hope to use cooperative winning strategies as test cases to drive the test executions and to issue test verdicts. To this end, we define a test case as a tree-like TIOTS, and we derive this TIOTS from the TIOGA model $\mathcal{S}$ and the strategy f. For better legibility, in the sequel we would present this TIOTS in the form of a timed automata and call it a timed automaton [1].

Given a TA location l, we use *outgoing*(l) to denote the set of output actions that originate from l. Given a state s in the TIOTS of a TA, we use *location*(s) to denote the corresponding location of s in the TA.

[1]Note that the derived test case is not really a timed automaton because there could exist non-reset clock assignments and non-integer constants in clock constraints.

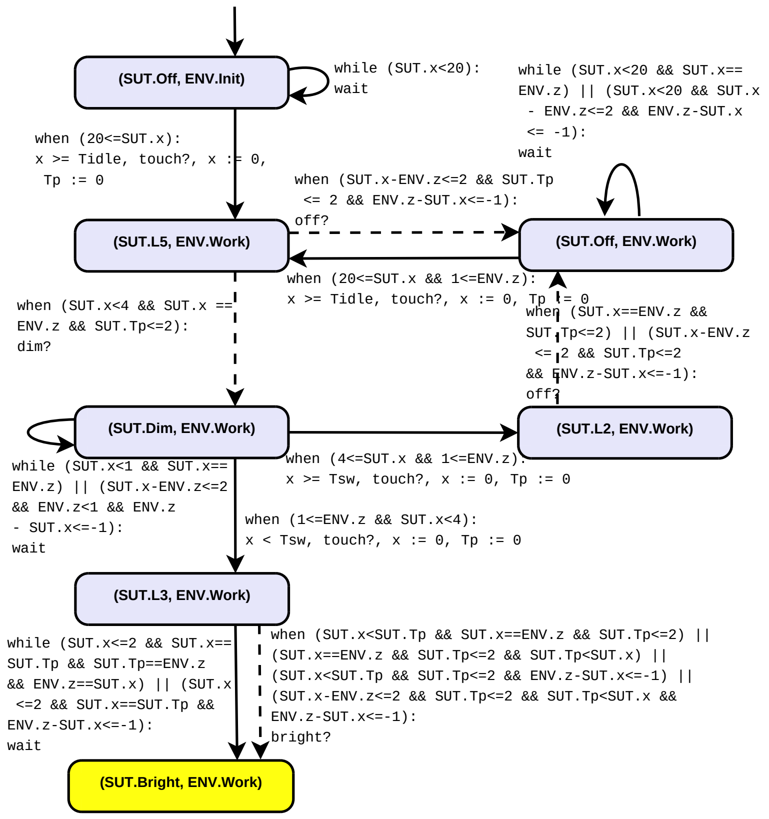

(SUT.Off, ENV.Init)
while (SUT.x<20): wait
while (SUT.x<20 && SUT.x== ENV.z) || (SUT.x<20 && SUT.x - ENV.z<=2 && ENV.z-SUT.x <= -1): wait
when (20<=SUT.x): x >= Tidle, touch?, x := 0, Tp := 0
when (SUT.x-ENV.z<=2 && SUT.Tp <= 2 && ENV.z-SUT.x<=-1): off?
(SUT.L5, ENV.Work)
(SUT.Off, ENV.Work)
when (20<=SUT.x && 1<=ENV.z): x >= Tidle, touch?, x := 0, Tp := 0
when (SUT.x<4 && SUT.x == ENV.z && SUT.Tp<=2): dim?
when (SUT.x==ENV.z && SUT.Tp<=2) || (SUT.x-ENV.z <= 2 && SUT.Tp<=2 && ENV.z-SUT.x<=-1): off?
(SUT.Dim, ENV.Work)
(SUT.L2, ENV.Work)
while (SUT.x<1 && SUT.x== ENV.z) || (SUT.x-ENV.z<=2 && ENV.z<1 && ENV.z - SUT.x<=-1): wait
when (4<=SUT.x && 1<=ENV.z): x >= Tsw, touch?, x := 0, Tp := 0
when (1<=ENV.z && SUT.x<4): x < Tsw, touch?, x := 0, Tp := 0
(SUT.L3, ENV.Work)
while (SUT.x<=2 && SUT.x== SUT.Tp && SUT.Tp==ENV.z && ENV.z==SUT.x) || (SUT.x <=2 && SUT.x==SUT.Tp && ENV.z-SUT.x<=-1): wait
when (SUT.x<SUT.Tp && SUT.x==ENV.z && SUT.Tp<=2) || (SUT.x==ENV.z && SUT.Tp<=2 && SUT.Tp<SUT.x) || (SUT.x<SUT.Tp && SUT.Tp<=2 && ENV.z-SUT.x<=-1) || (SUT.x-ENV.z<=2 && SUT.Tp<=2 && SUT.Tp<SUT.x && ENV.z-SUT.x<=-1): bright?
(SUT.Bright, ENV.Work)

Figure 5: An example cooperative winning strategy.

Definition 5 (test case)**.** *A* test case *is a timed automaton* $\mathcal{T} = (L_t, l_{0t}, Act, X_t, E_t, Inv_t)$*, where:*

- L_t *is a set of locations which include terminal nodes that are marked as* pass*,* fail *or* inconc*;*
- $l_{0t} \in L_t$ *is the initial location;*
- $Act = Act_{in} \cup Act_{out}$*;*
- X_t *is a set of clocks;*
- Inv_t *associates invariants to locations; and*
- E_t *is the transition relation such that:*
  - $\mathcal{T}$ *is deterministic;*
  - $\mathcal{T}$ *has bounded behavior, i.e.,* $\forall \sigma = \sigma_1\sigma_2\sigma_3 \ldots \in \mathsf{Runs}(\mathcal{T}) \,.\, (\#(\{i \mid \sigma_i \in Act_{in} \cup Act_{out}\}) < \infty \;\wedge\; \exists n > 0 \,.\, ((\Sigma_i\, \sigma_i) < n,\; \sigma_i \in \mathbb{R}_{\geq 0}))$*; and*

– $\forall l_t \in (L_t \setminus \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconc}\}) \,.\, \forall \alpha \in Act_{out} \,.\, (\alpha \in outgoing(l_t))$. $\square$

The basic idea of test case generation is to keep looking up the generated cooperative winning strategy and the SPEC model to decide when to make what move against the IMP in (forthcoming) test execution, and which decision (pass, fail, inconc, or to continue on by recursively building the test tree) to make upon every possibly observed output from IMP.

Let s be a semantic state of $\mathcal{S}$, and s_0 the initial state. We use $w = width(f_{mov}(s))$ to denote the width of the strategy-suggested observing (integer time delay) window. Let $x \in X_t$ be a unique clock variable for recording the timing constraints in f and for building invariants and transitions in the test case TA. We use another unique clock variable $h \in X_t$ to record the time when a strategy-desired output happens. A location l_t of the test case TA is called a *conditional branching location* if at this location the branching is based on the just-recorded occurrence time h of an output action. Thus l_t is the destination location of the transition of an observed output. Algorithm 4.1 sketches out the main idea of test case derivation.

A key point of our test generation algorithm is about the semantics of forced actions. We adopt the following semantics: if the upper bound of a location invariant of the TIOGA is hit, then we check whether there is any outgoing edge with enabled input action leading to other location, or some self-looping edge with enabled input action and clock resets. If there are no such edges, then we check whether there is some outgoing edge with output action leading to other location, or some self-looping edge with output action and clock resets. If there is also no such edge and the strategy still suggests "delay" when hitting the invariant, then we report fail.

Because the (relaxed-sense) test purpose is checked to be satisfiable, the synthesized (cooperative) winning strategy is of finite length and it guides us towards the goal states, Algorithm 4.1 will always terminate. The complexity of this recursive algorithm largely depends on the sizes of the strategy and the set *Act*.

5 Test execution

Definition 6 (test execution). *Let* $\mathsf{TIOTS}(\mathcal{T}) = (T, t_0, Act_{in}, Act_{out}, \rightarrow_t)$ *be the TIOTS of test case* $\mathcal{T}$*, and assume that the IMP may be modeled as another TIOTS* $\mathcal{I} = (I, i_0, Act_{in}, Act_{out}, \rightarrow_i)$*. The* execution *of* $\mathcal{I}$ *with* $\mathcal{T}$ *is modeled by the synchronous parallel execution* $\mathsf{TIOTS}(\mathcal{T}) \,||\, \mathcal{I}$ *which is defined by the rules:*

$$\frac{t \xrightarrow{g_1,\,\alpha}_t t',\ i \xrightarrow{g_2,\,\alpha}_i i'}{t||i \xrightarrow{g_1 \wedge g_2,\,\alpha} t'||i'}\ \alpha \in Act_{in}, \qquad \frac{t \xrightarrow{\alpha}_t t',\ i \xrightarrow{\alpha}_i i'}{t||i \xrightarrow{\alpha} t'||i'}\ \alpha \in Act_{out}, \qquad \frac{t \xrightarrow{d}_t t',\ i \xrightarrow{d}_i i'}{t||i \xrightarrow{d} t'||i'}\ d \in \mathbb{R}_{\geq 0}$$

where $t, t' \in T$, $i, i' \in I$, *and* $g_1, g_2 \in \mathcal{C}(X)$. $\square$

Algorithm 4.1 TestCase($\mathcal{S}$, f)

Input: TIOGA specification $\mathcal{S}$, cooperative winning strategy f;
Output: a test case TA $\mathcal{T}$;
Main:

```
1: $w := 0; x := 0; h := 0$; add_node($s_0$); //initialization
2: BuildTestCase($s_0$).
```

Procedure BuildTestCase(s): //s: a state in $\mathcal{S}$ and node in $\mathcal{T}$

```
1: if $s$ does not correspond to a conditional branching location then
2:    $w := width(f_{mov}(s))$;
3:    add invariant "$x \leq w$" for node $s$ in $\mathcal{T}$;
4:    foreach $o \in Act_{out}$ do
5:       if $o \in outgoing(location(s))$ then
6:          if the destination state of this transition is a goal state then
7:             add_edge("$s \xrightarrow{o!} \mathsf{pass}$"); //add a node pass and an edge in $\mathcal{T}$
8:          elseif $(f_{stg}(s) = coop) \wedge (o \neq act(f_{mov}(s)))$ then
9:             add_edge("$s \xrightarrow{o!} \mathsf{inconc}$");
10:         else // go recursively with a conditional branching location
11:            add_edge("$s \xrightarrow{o!,\, h:=x,\ x:=0} s'$");
12:            $s' := (s$ After $h)$ After $o$;
13:            BuildTestCase($s'$);
14:      else
15:         add_edge("$s \xrightarrow{o!} \mathsf{fail}$");
16:   endfor
17:   case $f_{mov}(s)$ of
18:      "$\lambda$":
19:         if $((s$ After $w)$ hits $location(s)) \wedge (f_{mov}(s$ After $w) = \lambda)$ then
20:            add_edge("$s \xrightarrow{x=w} \mathsf{fail}$"); // acc. to semantics of forced actions
21:         else
22:            add_edge("$s \xrightarrow{x=w,\, x:=0} s'$");
23:            $s' := s$ After $w$;
24:            BuildTestCase($s'$);
25:      "to offer input $i$":
26:         $s' := tgt(f_{mov}(s$ After $w))$;
27:         add_edge("$s \xrightarrow{x=w,\, i?} s'$");
28:         BuildTestCase($s'$);
29:      "to observe output $o$":
30:         if $(f_{stg}(s) = coop) \wedge (\{f_{mov}(s)\} \wedge \rightarrow_{out} \neq \emptyset) \wedge$ (no output occurs) then
31:            add_edge("$s \xrightarrow{x=w} \mathsf{inconc}$");
32:   esac
33: else // $s$ corresponds to a conditional branching location
34:   add invariant "$x = 0$" for $s$; // an "urgent" location in $\mathcal{T}$
35:   branching according to $f$ and the value of $h$;
36:   recursive calls of BuildTestCase().
```

End Procedure

A *test run* is a run of the product $\mathsf{TIOTS}(\mathcal{T})\,||\,\mathcal{I}$ that leads to a state whose left sub-state corresponds to a terminal node of $\mathcal{T}$. Formally, $\sigma \in \mathsf{Runs}(\mathsf{TIOTS}(\mathcal{T})\,||\,\mathcal{I})$ is a test run if, $\exists i' \in I, t' \in T \,.\, ((t_0||i_0 \xrightarrow{\sigma} t'||i') \cap (location(t') \in \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconc}\}))$. In the pass case, we say that $\mathcal{I}$ *passes* test run σ. In the fail case, we say that $\mathcal{I}$ *fails* σ. The inconc case indicates neither passing nor failing. It simply means that we do not get cooperation and are thus not assured of being able to reach the goal states.

Given $\mathcal{I}$ and $\mathcal{T}$, if there is a failing test run of $\mathsf{TIOTS}(\mathcal{T})\,||\,\mathcal{I}$, then $\mathcal{I}$ *fails* $\mathcal{T}$. If every test run of $\mathsf{TIOTS}(\mathcal{T})\,||\,\mathcal{I}$ is not failing, we say $\mathcal{I}$ *passes* $\mathcal{T}$.

6 Soundness and completeness

The soundness property of the test method says that if there exists a failing test run, then the system implementation indeed does not comply with the system specification. The (partial) completeness property of the test method says that if the system implementation does not comply with the system specification with respect to the specified relaxed-sense test purpose, then we can always find a failing test run.

Let $\mathcal{S} = (L, l_0, Act, X, E, Inv)$ be a TIOGA specification with $Act = Act_{in} \cup Act_{out}$, $\mathsf{TIOTS}(\mathcal{S})$ be its corresponding TIOTS, $\mathcal{I} = (I, i_0, Act_{in}, Act_{out}, \rightarrow_i)$ be a TIOTS implementation, $\varphi = \mathsf{E}\Diamond\ \mathsf{control}\colon \mathsf{A}\Diamond\,\phi$ be a relaxed-sense test purpose such that φ can be enforced on $\mathcal{S}$, and $\mathcal{S}_f$ and $\mathcal{I}_f$ be the strategy f-constrained behaviors of $\mathcal{S}$ and $\mathcal{I}$, respectively, then we have:

Theorem 1 (soundness). $\forall f \in \mathsf{Strategy}(\mathcal{S}, \varphi) \,.\, (\,(\mathcal{I}\ fails\ \mathsf{TestCase}(\mathcal{S}, f)) \;\Rightarrow\; \mathcal{I}\ \cancel{\mathsf{tioco}}\ \mathsf{TIOTS}(\mathcal{S})\,)$.

Proof. (sketch.) Let $\mathcal{T} = \mathsf{TestCase}(\mathcal{S}, f)$ and $\mathsf{TIOTS}(\mathcal{T}) = (T, t_0, Act_{in}, Act_{out}, \rightarrow_t)$. By $(\mathcal{I}\ fails\ \mathcal{T})$ we know that $\exists \sigma \in \mathsf{Runs}(\mathsf{TIOTS}(\mathcal{T})\,||\,\mathcal{I}) \,.\, \exists i' \in I \,.\, \exists t' \in T \,.\, (t_0||i_0 \xrightarrow{\sigma} t'||i') \cap (location(t') = \mathsf{fail})$. From Algorithm 4.1 we know that there are two cases of finishing with a fail verdict. The first case is that we observe an invalid output w.r.t. $\mathcal{S}$ (the first fail verdict in Alg. 4.1). According to Definition 6 and Definition 1, we conclude that $\mathcal{I}\ \cancel{\mathsf{tioco}}\ \mathsf{TIOTS}(\mathcal{S})$. The second case is when we are hitting the location invariant (the λ-case in Alg. 4.1). According to the forced semantics of controllable and uncontrollable actions in this circumstance, there should be a forced output. But unfortunately we have not observed it. Thus the conformance relation has been violated. Therefore $\mathcal{I}\ \cancel{\mathsf{tioco}}\ \mathsf{TIOTS}(\mathcal{S})$. □

Theorem 2 ((partial) completeness). $\forall f_1 \in \mathsf{Strategy}(\mathcal{S}, \varphi) \,.\, (\,(\mathcal{I}_{f_1}\ \cancel{\mathsf{tioco}}\ \mathcal{S}_{f_1}) \;\Rightarrow\; \exists f_2 \in \mathsf{Strategy}(\mathcal{S}, \varphi) \,.\, (\mathcal{I}\ fails\ \mathsf{TestCase}(\mathcal{S}, f_2))\,)$.

Proof. (sketch.) By $(\mathcal{I}_{f_1}\ \cancel{\mathsf{tioco}}\ \mathcal{S}_{f_1})$ we know that there exists a timed trace σ such that $\sigma \in \mathsf{TTr}(\mathcal{I}_{f_1}) \backslash \mathsf{TTr}(\mathcal{S}_{f_1})$ according to Definition 1. For simplicity, we suppose σ ends with the first violation w.r.t. $\mathcal{S}_{f_1}$. According to Algorithm

4.1 we know that this has two possible consequences. The first case is that σ has an output action which is disallowed in $\mathsf{TIOTS}(\mathcal{S})$. The second case is that σ has an observed quiescence when hitting a location invariant, but it is disallowed in $\mathsf{TIOTS}(\mathcal{S})$. Therefore we can build another timed trace σ' such that σ' has exactly the same prefix as σ, but σ' ends without a violation w.r.t. $\mathsf{TIOTS}(\mathcal{S})$. Therefore, we can generate some cooperative winning strategy f_2 from $\mathcal{S}$ and φ, and build a test case from $\mathcal{S}$ and f_2 such that σ' is not a failing run but σ is a failing run. According to Definition 6, we can conclude that $\exists f_2 \in \mathsf{Strategy}(\mathcal{S}, \varphi) \,.\, (\mathcal{I}\ fails\ \mathsf{TestCase}(\mathcal{S}, f_2))$. $\square$

7 Case study

We consider a simple leader election protocol (LEP) problem [Lam05b], where we have one TIOGA for an arbitrarily chosen protocol node (the "plant"), and two TIOGAs for simulating all other protocol nodes and simulating a buffer with certain capacity that is used for accepting, holding and forwarding messages (the "controller"). The plant has uncontrollable actions in the sense that a *timeout*! event might occur after waiting for a certain period of time without receiving "useful" messages, and an *ignore*! event might occur due to loss of messages. We defined the following test purposes (winning objectives):

- TP1: $\mathsf{E}\Diamond$ control: $\mathsf{A}\Diamond$ $\texttt{exists}(\texttt{i} : \texttt{BufferId})(\texttt{inUse}[\texttt{i}] == 1)$;
- TP2: $\mathsf{E}\Diamond$ control: $\mathsf{A}\Diamond$ $(\texttt{SUT.bufferInfo} == 1)$ && $\texttt{SUT.forward}$; and
- TP3: $\mathsf{E}\Diamond$ control: $\mathsf{A}\Diamond$ $\texttt{forall}(\texttt{i} : \texttt{BufferId})(\texttt{inUse}[\texttt{i}] == 1)$.

The original (stronger) versions of all of these test purposes (e.g., control: $\mathsf{A}\Diamond$ $\texttt{exists}(\texttt{i} : \texttt{BufferId})(\texttt{inUse}[\texttt{i}] == 1)$) are checked to be unenforceable using UPPAAL-TIGA. After they are relaxed (i.e., being prefixed with $\mathsf{E}\Diamond$, as shown above), they are all proved to be enforceable.

We carried out the strategy generation experiments on an application server. Table 1 presents the performance results of CPU time overheads and memory consumptions, where / means "out of memory". Each sub-column corresponds to one parameter configuration, where n means that there are n nodes in the protocol, and there is a message buffer of size n, and the maximum distance between any two nodes is limited to $(n-1)$. The table indicates that for some test purposes, cooperative winning strategy generation for the LEP protocol with up to 7 nodes takes less than 10 minutes. Memory consumption appears to be more of a problem. Furthermore, it seems that both time overhead and memory consumption are very sensitive to the test purposes. In summary, considering that strategy generation is the most computation intensive (and also the most inherently complex) step in our test framework, our testing method seems not to be only of theoretical value.

Table 1: Cooperative winning strategy generation for LEP with lossy channels.

	Time (s)						**Memory** (MB)					
	n=3	4	5	6	7	8	n=3	4	5	6	7	8
TP1	0.04	0.17	0.81	3.21	10.57	30.65	0.1	4.2	7.9	18.9	48.6	119.5
TP2	0.11	1.32	11.74	85.14	558.67	/	4.3	13.0	80.3	517.0	2959	/
TP3	3.22	75.56	/	/	/	/	24.3	493.5	/	/	/	/

Experiment platform: Sun Fire X4100 server, 2×2.4GHz CPU (Dual Core AMD Opteron 275), 4096MB RAM, Suse Linux Enterprise Desktop 10 - 64bit.

8 Conclusions

We examine black-box conformance testing based on uncontrollable timed system models using a cooperative game-based approach. We model the systems with timed I/O game automata and specify the test purposes as ACTL formulas. We generate cooperative winning strategies, derive test cases and execute them on the implementation. The test method is proved to be sound and complete w.r.t. the test purposes. Preliminary experimental results indicate that it is a viable approach. Compared with previous work [DLLN08b], the method in this paper will enable us to conduct a broader type of model-based conformance testing.

Sections 3 - 7 explain and demonstrate our cooperative approach mainly using reachability test purposes and reachability games. The underlying principles and the methodologies also apply to safety test purposes and safety games. In that case, the “surely winning” portion of a cooperative winning strategy for a safety game may also induce infinite executions. Accordingly, the test execution procedure might run forever without issuing a pass verdict.

Future work includes: (1) more case studies for performance evaluation, test effectiveness analysis and test method scalability improvement; (2) to generalize state-based strategy to history-based strategy; (3) to implement the test case generation and execution algorithms to build a fully automated strategy-based testing environment; (4) strategy-based testing with partial observability.

Paper E: Timed Testing under Partial Observability

Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen
Center for Embedded Software Systems (CISS)
Department of Computer Science
Aalborg University, Denmark

Abstract

This paper studies the problem of model-based conformance testing of partially observable timed systems. We model the system under test (SUT) using timed game automaton (TGA) that has internal actions, output uncertainty and timing uncertainty of outputs. We define the partial observability of SUT using a set of observable predicates over the TGA semantic state space, and specify the test purposes as ACTL logic formulas. A partially observable timed game solver UPPAAL-TIGA is used to generate winning strategies, which are then used as test cases. We propose a conformance testing framework for this particular setting, define a partial observation-based conformance relation, present the test execution algorithms, and prove the soundness and completeness of this test method. Experiments on some non-trivial examples show that this method yields encouraging results.

Keywords: Real-Time Systems, Timed Game Automata (TGA), Observable Predicates, Test Purposes, Observation-Based Stuttering-Invariant (OBSI) Strategies, Observation-Based Conformance

1 Introduction

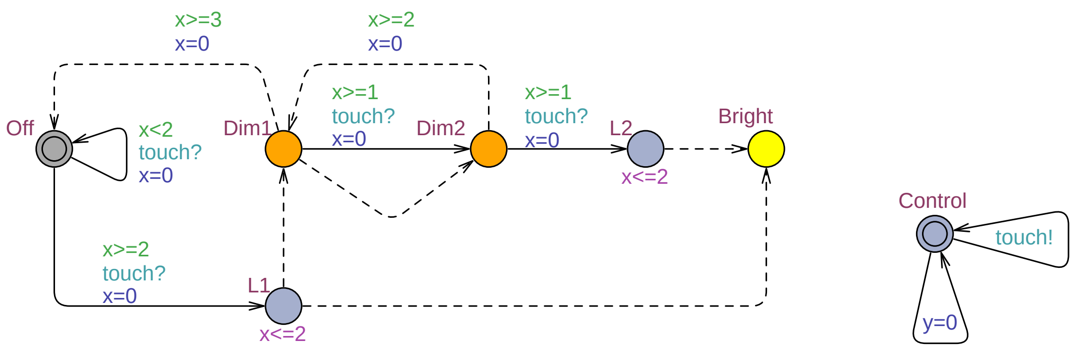
Timed automaton (TA) [AD94] has been widely used to model safety-/mission-/economic- critical real-time systems. A considerable proportion of existing efforts on real-time system testing [ENDKE98, SVD01, KJM03, HLN$^+$03, NS03, LMN04, BB04, KT04] are based on the TA model or its variants. To enable conformance testing, these methods build their implementation relations (a.k.a. conformance relations) on top of e.g. trace equivalence [ENDKE98, SVD01] or the ioco conformance relations [KJM03, HLN$^+$03, LMN04, BB04, KT04]. To steer the testing towards certain test purposes or test coverage criteria, some of these methods need (to be enhanced with) the assumption of *full observability* (a.k.a. *perfect information*), i.e., at any time, the tester knows precisely what state or configuration[1] the system under test (SUT) is in, or she can uniquely infer one such state or configuration by observing an externally observable timed input/output action sequence on the well-defined tester/SUT interface. The full observability assumption paves way to accurately drive, monitor the test executions and issue test verdicts. However, in practice this assumption is hardly realistic. On one hand, if the SUT consists of several interacting components, the tester might not be able to observe the internally coupling inputs/outputs or the internal state changes that are caused by those internal actions. On the other hand, if the tester has only limited-precision sensors to measure the SUT, she might not tell which exact state the SUT is in, or she might not precisely observe a timed input/output sequence. Environment noises and external interferences could also affect the observations.

In this paper we consider the problem of testing timed systems that are only *partially observable* (or, with *imperfect information*). One way to characterize partial observability is to assume that only a proper subset of those outputs from the SUT to the tester can be observed, and/or only a proper subset of the system clocks can be read by the tester [BDMP03]. In another way, partial observability is characterized in terms of a finite number of possible observations to be made on the SUT states (configurations) [CDL$^+$07]. This paper follows the latter approach which has mature algorithms and tool support.

We use the Smart Lamp problem [HLN$^+$03] as an illustrating example. Fig. 1(a) and 1(b) show the system specification (SPEC) models of the Lamp and its user, respectively. The user interacts with the Lamp by touching a touch-sensitive pad. In Fig. 1(a), there are four brightness levels (in ascending order) for the Lamp: `Off`, `Dim1`, `Dim2` and `Bright`. The Lamp model is initially in location `Off`. If the pad is touched at an appropriate time ($x \geq 2$), the Lamp will go to location `L1` where within 2 time units it will non-deterministically go to `Bright` or go to `Dim1`. At location `Dim1`, a *touch*? input at appropriate time can bring the TA to

[1]A *configuration* is a "snapshot" of the system state. For example, for a UPPAAL TA, a configuration consists of the information of the current residing location and the current values of all clock and data variables.

`Dim2`. The Lamp can automatically go to `Dim2` at any time. If clock x rises up to 3 while the Lamp is in `Dim1`, then it can automatically go `Off`. In Fig. 1(a), the edges that are not labeled by *touch*? are internal transitions, which need no synchronization with the user TA.



(a) Lamp (SUT) (b) user (tester)

Figure 1: A smart lamp example.

The dashed lines in Fig. 1(a) denote transitions that are controlled by the Lamp only. Note that in `Dim1`, `Dim2`, `L1` and `L2`:

(1) The internal transitions can autonomously occur when the conditions (if any) are satisfied, and their occurrences cannot be observed by the user;

(2) The Lamp itself decides whether or not to make an output or an internal transition, and if yes, which output or internal transition to make; and

(3) The Lamp itself decides when to make an output or internal transition.

These three characteristics are called *internal actions*, *uncontrollable actions* and *timing uncertainty of uncontrollable actions*, respectively. A TA with these characteristics is a liberal specification model that can be refined into a family of similar but different implementations. If the tester offers an input stimulus, different implementations might produce different responses.

In [DLLN08b] we view testing of a TA-modeled timed system as playing a timed game under full observability, where the tester acts as a game player and the SUT acts as the game opponent. The test purposes are given in reachability or safety ACTL formulas. For example, "control: A◇ `Bright`" means that the tester can manage to guarantee that the `Bright` location can always be eventually reached, no matter how the SUT (i.e., the Lamp) behaves. We check the enforceability of the test purposes (winning objectives) using an existing time game solver UPPAAL-TIGA [CDF$^+$05, BCD$^+$07]. If the outcome is positive, the tool can synthesize a *state-based* winning strategy for the tester, which ensures enforcing the winning objectives by providing step-by-step guidance to the tester, e.g., "if the SUT is in states $\langle \texttt{Dim1},\ 1 \leq x < 3\rangle$, the tester should offer a *touch*? to the SUT; if in states $\langle \texttt{Dim1},\ x < 1\rangle$, the tester should just wait (stay quiet) there". In this way the tester will certainly win the game (by finally arriving at a "good" state, or by constantly avoiding the "bad" states). In an off-line testing manner, the generated strategies can be used as test cases to test a family of similar but different implementations.

If the SUT is only *partially observable*, say, in this Smart Lamp example the tester can observe [2]:

(1) whether or not the SUT is "off" (i.e., in `Off`); and

(2) whether or not the SUT is "dim" (i.e., in `Dim1` or `Dim2`, but not exactly in which one); and

(3) whether or not the SUT is "bright" (i.e., in `Bright`),

then the methods in [CDF+05] [DLLN08b] are no longer applicable. On one hand, synthesis of state-based strategies is based on the full observability assumption. On the other hand, the tester can not use (execute) a pre-computed strategy under *partial* observability. Consider that the tester feeds the SUT with a timed input preamble $2 \cdot$ *touch*? $\cdot\, 2$ at the initial state of Fig. 1(a). She may get the observation "dim" signaling that either location `Dim1` or location `Dim2` has been reached. But since she does not know the exact location, she has no idea which next move to take according to the strategy.

Now our problem is that *under partial observability, given a test purpose, is it possible to synthesize an observation-based winning strategy for the tester to ensure that the* `Bright` *location is always reachable, no matter how the Lamp behaves*? And if so, how can we *use this strategy to test whether a concrete implementation IMP also respects the test purpose*?

1.1 Related work

Model-based testing could be comprehensive testing [ENDKE98, SVD01, HLN+03, NS03, LMN04, BB04, KT04] or targeted testing [KJM03, HLN+03]. We follow the latter approach, i.e., we use test purposes to direct the testing towards some particular functional properties.

Many existing methods on real-time testing use timed automata to model the systems in question. For the sake of testability [SVD01], the timed automata are restricted to be *predictable*, i.e., they should be deterministic (or determinizable),

[2] In practice, the observable predicates of (1)-(3) can be implemented by probing/instrumenting the SUT with some Light sensors or software-defined location reporters; the assumption in predicate (2) is reasonable if the difference between these two brightness levels is too small to be discerned by the Light sensors.

output-urgent and have isolated outputs[3] [SVD01, HLN$^+$03]. This predictability leads to full observability, and thus favors well-steered state space exploration. But from a model-based development point of view, these TA models are too detailed and too restricted to be suitable for early-stage modeling. In contrast, a liberal TA model will allow the implementor more freedom, will enable the tester to capture the high-level design requirements rather than the less important implementation details, and a liberal TA model is usually more natural and more succinct than a detailed one.

By canceling the restrictions of isolated outputs and output-urgency, we obtain a kind of liberal TA model called timed game automaton (TGA) [MPS95]. The set of transitions of a TGA are partitioned into subsets of tester-controllable ones (drawn in solid lines) and tester-uncontrollable ones (in dashed lines), see Fig. 1(a). Given a network of TGAs and a test purpose in the form of reachability or safety property, there are efficient algorithm [CDF$^+$05] and tool [BCD$^+$07] to solve the timed game.

Game-theoretic approaches to untimed system testing have been discussed in [ACY95, Yan04, BGNV05]. In the timed case, winning strategies for a given test purpose have been used as tests for off-line black-box conformance testing [DLLN08b]. Specifically, if the test purpose is not satisfied by the SPEC model, we can still possibly synthesize cooperative winning strategies and use them to test the SUT against the test purpose as long as the SUT reacts to our test inputs in a desired manner [DLLN08a].

The methods in [DLLN08b, DLLN08a] require the full observability assumption. If we wish to abstract from component interactions inside the SUT model and wish to allow measurement inaccuracy, then we have to handle partial observability. More recently, it has been shown that by fixing the resources of the controller (i.e. a maximum number of clocks and a maximum allowed constant in guards), the timed control problems based on these TGAs are decidable [BDMP03], and it is possible to algorithmically synthesize observation-based stuttering-invariant strategies for the tester [CDL$^+$07]. This motivates us to investigate the possibility of using such strategies as tests for timed systems under partial observability.

[3]Given a TA (L, l_0, Act, X, E, Inv) where L is the location set, $l_0 \in L$ the initial location, $Act = (Act_I \cup Act_O)$ the set of input and output actions, and X, E, Inv the set of clocks, edges and invariants, respectively. Let $(S, s_0, Act, \rightarrow)$ be its underlying timed labeled transition system. We say that the TA is *deterministic* if $\forall s \in S \,.\, \forall \alpha \in Act \,.\, ((s \xrightarrow{\alpha} s') \wedge (s \xrightarrow{\alpha} s'') \Rightarrow (s' = s''))$. The TA is *output-urgent* if $\forall s \in S \,.\, \forall \alpha \in Act_O \,.\, ((s \xrightarrow{\alpha}) \Rightarrow \forall d \in \mathbb{R}_{>0} \,.\, (s \overset{d}{\not\rightarrow}))$. The TA has *isolated outputs* if $\forall s \in S \,.\, \forall \alpha \in Act_O \,.\, \forall \beta \in Act \,.\, (((s \xrightarrow{\alpha}) \wedge (s \xrightarrow{\beta})) \Rightarrow (\alpha = \beta))$.

1.2 Contributions

The main contributions of this paper include: (1) we apply game strategies to the context of model-based testing and propose a framework of conformance testing of timed systems based on partial observations; (2) we define an observation-based conformance relation between the SPEC and the IMP, propose test execution algorithms based on this relation, and prove their soundness and completeness; (3) we conduct preliminary case studies of test generation using a prototype tool and report the experimental results.

2 Timed control under partial observability

2.1 Partially observable time game

Given two TGAs that model a controller program (the "controller", or "Player 1") and the system under control (the "plant", or "Player 2"), and given a control objective which is formulated as e.g. a reachability or safety property, a *timed control problem* consists in finding a winning strategy for the controller such that the control objective will be enforced, no matter how the plant behaves.

For the time control problem to be decidable [BDMP03, CDL+07], we assume that all clock values in the TGA are bounded by a natural number, say $\mathsf{M} \in \mathbb{N}_{\geq 0}$.

Let X be a finite set of non-negative real-valued variables called *clocks*, then $\mathcal{C}(X, \mathsf{M})$ is the set of constraints generated by the grammar

$$\varphi ::= x \bowtie k \mid x - y \bowtie k \mid \varphi_1 \wedge \varphi_2,$$

where $k \leq \mathsf{M} \in \mathbb{N}_{\geq 0}$, $x, y \in X$, and $\bowtie \in \{<, \leq, =, \geq, >\}$; and $\mathcal{B}(X, \mathsf{M})$ is a subset of $\mathcal{C}(X, \mathsf{M})$ defined by

$$\varphi ::= \texttt{true} \mid k_1 \leq x < k_2 \mid \varphi_1 \wedge \varphi_2,$$

where $k_1 < k_2 \leq \mathsf{M} \in \mathbb{N}_{\geq 0}$, and $x \in X$.

Definition 1 (timed game automaton, TGA [MPS95]). *A* timed game automaton *is a tuple $A = (L, l_0, Act_c, Act_u, X, E, Inv)$ where*

- *L is a finite set of locations;*
- *$l_0 \in L$ is the initial location;*
- *Act_c and Act_u are disjoint finite sets of Player 1-controllable (input) actions and Player 2-controllable (output or internal) actions, respectively. Specifically, internal action $\tau \in Act_u$;*
- *X is a finite set of real-valued clocks;*

- *E is a finite set of edges partitioned into Player 1-controllable ones belonging to $L \times \mathcal{B}(X, \mathsf{M}) \times Act_c \times 2^X \times L$, and Player 2-controllable ones belonging to $L \times \mathcal{C}(X, \mathsf{M}) \times Act_u \times 2^X \times L$. Specifically, $(l, g, \tau, r, l') \in L \times \mathcal{C}(X, \mathsf{M}) \times Act_u \times 2^X \times L$ is an internal transition; and*

- *$Inv : L \to \mathcal{B}(X, \mathsf{M})$ associates invariants to locations.* □

For a network of interacting TGAs, we define their parallel composition in the usual manner.

The behavior of a TGA can be described using a timed labeled transition system (TLTS).

Definition 2 (2-player timed labeled transition system, 2-player TLTS)**.** *A* 2-player timed labeled transition system *is a tuple $(S, s_0, Act_c, Act_u, \to)$, where*

- *S is an (infinite) set of semantic states;*

- *$s_0 \in S$ the initial state;*

- *Act_c and Act_u the Player 1-controllable and Player 2-controllable actions, respectively; and*

- *$\to \subseteq S \times (Act_c \cup Act_u \cup \mathsf{R}_{>0}) \times S$ is the transition relation.* □

Given an M-bounded timed automaton, a *valuation v* of the clock variables is a mapping from X to an interval $[0, \mathsf{M}]$ of real numbers, i.e., $v : X \to [0, \mathsf{M}]$. The set of all such valuations is denoted $[X \to [0, \mathsf{M}]]$ or $[0, \mathsf{M}]^X$. For $r \subseteq X$, we denote by $[r \to 0]v$ the new valuation which is obtained from v by assigning 0 to any $x \in r$. Let $d \in \mathsf{R}_{\geq 0}$, v be an M-valuation, then $v + d$ is a valuation $(v + d)(x) = v(x) + d$, if for all $x \in X$, $v(x) + d \leq \mathsf{M}$. For $g \in \mathcal{C}(X, \mathsf{M})$ and $v \in [0, \mathsf{M}]^X$, we write $v \models g$ if v satisfies g.

Let $A = (L, l_0, Act_c, Act_u, X, E, Inv)$ be a TGA. let $K \subseteq L$ and $\varphi \in \mathcal{B}(X, \mathsf{M})$. We call (K, φ) an *observable predicate.* We use a finite set of observable predicates $P \subseteq 2^L \times \mathcal{B}(X, \mathsf{M})$ to observe a TGA. For instance, in Fig. 1(a) we can have $P = \{(\{\texttt{Off}\}, \texttt{true}), (\{\texttt{Dim1, Dim2}\}, \texttt{true}), (\{\texttt{Bright}\}, \texttt{true}), (L, 0 \leq y < 1)\}$ [4].

An observable predicate (K, φ) is true at a TGA semantic state $s = \langle l, v\rangle$ iff ($l \in K$ and $v \models \varphi$). A *state observation* (or *observation* for short) $o_{P,s}$ of the TGA with a set P of predicates at state s is a valuation of all the predicates in P at s:

$$o_{P,s} : P \to \{\texttt{true}, \texttt{false}\}.$$

For instance, in Fig. 1(a) at semantic state $\langle \texttt{Dim2}, (x = 2, y = 4)\rangle$ we have the observation $o_{P,s}$ such that

[4]The constraint $0 \leq y < 1$ means that the tester can test whether value of clock y falls within $[0, 1)$, but she cannot/need not read the exact value of y. In practice, this predicate can be implemented as a countdown timer whose timeout can be externally observed by the tester.

$o_{P,s}((\{\texttt{Off}\}, \texttt{true})) = \texttt{false}$,
$o_{P,s}((\{\texttt{Dim1}, \texttt{Dim2}\}, \texttt{true})) = \texttt{true}$,
$o_{P,s}((\{\texttt{Bright}\}, \texttt{true})) = \texttt{false}$, and
$o_{P,s}((L, 0 \leq y < 1)) = \texttt{false}$.
Let O_P be the set of all possible observations with P, then by definition we have $|O_P| \leq 2^{|P|}$. Each element in O_P corresponds to a set of TGA states that have the same truth value for each of the observable predicates, hence an equivalence class. Thus O_P defines a partition of the TGA state space.

For TGA A and a set P of observable predicates, we define a function $\gamma_P : O_P \to 2^{L\times[0,\mathsf{M}]^X}$ to map each observation o to its class of equivalent TGA states:

$$\gamma_P(o) = \Big\{\langle l, v\rangle \mid \bigwedge_{\{(K,\varphi)\,\mid\, o(K,\varphi)=\texttt{true}\}} (\langle l, v\rangle \vDash (K,\varphi)) \quad \wedge \bigwedge_{\{(K,\varphi)\,\mid\, o(K,\varphi)=\texttt{false}\}} (\langle l, v\rangle \nvDash (K,\varphi))\Big\}.$$

We define a function ξ_P to make observation of a TGA state:

$$\xi_P : (L \times [0,\mathsf{M}]^X) \to O_P.$$

Clearly, for $\langle l, v\rangle \in S$, if $\langle l, v\rangle \in \gamma_P(o)$, then we have $\xi_P(\langle l, v\rangle) = o$.

A TGA which is associated with a set of observable predicates is called a *partially observable* TGA (PO-TGA), whose semantics is defined as a 2-player TLTS.

Definition 3 (PO-TGA semantics)**.** *The* semantics *of a TGA $A = (L, l_0, Act_c, Act_u, X, E, Inv)$ that is associated with a set P of observable predicates is a 2-player TLTS $S_A = (S, s_0, Act_c, Act_u, \to)$ where:*

- $S = \{\langle l, v\rangle \mid (l \in L) \wedge (v \in (\mathsf{R}_{\geq 0} \cap [0,\mathsf{M}])^X) \wedge (v \models Inv(l))\}$*, and* $s_0 = \langle l_0, \bar{0}\rangle \in S$*;*
- *the transition relation $\to$ is composed of*
  - *discrete transitions: $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$ if, for $\langle l, v\rangle, \langle l', v'\rangle \in S$, and $a \in (Act_c \cup Act_u)$, there exists $e = (l, a, g, r, l') \in E \,.\, ((v \models g) \wedge (v' = [r \to 0]v) \wedge (v' \models Inv(l')))$; and*
  - *time transitions: $\langle l, v\rangle \xrightarrow{d} \langle l, v + d\rangle$ if, for $\langle l, v\rangle \in S$, $d \in \mathsf{R}_{>0}$, and ξ_P over A and P, $\forall d' \in [0, d] \,.\, (u+d' \models Inv(l))$ and $\forall d'' \in [0, d) \,.\, \xi_P(\langle l, v+ d''\rangle) = \xi_P(\langle l, v\rangle)$.* □

Definition 3 requires that during the period of a time transition the current observation remains unchanged. Therefore, a new observation occurs only as a consequence of a discrete transition, or at the rightmost point of a time delay.

Let $s \in S$, we define the set of possible actions or delays at s as $enable(s) = \{\sigma \in (Act_c \cup Act_u \cup \mathsf{R}_{>0}) \mid \exists s' \in S \,.\, ((s, \sigma, s') \in\to)\}$.

2.2 OBSI winning strategy

We consider *competing* rather than strictly turn-based two-player games between a plant and its controller (the tester).Thus if a controller move and a plant move are enabled at the same time, the former one could always be preempted by the latter one.

An observation-based stuttering-invariant (OBSI) strategy guides the controller to play the game in *sessions* (*macro* steps), during each of which the observation remains unchanged. A macro step consists of a number of consecutive *micro* steps, each of which is caused either by a controller move, or by a plant move:

- During each session of the controller/plant interaction, the controller sticks to either a controllable input $a_1 \in Act_c$ or *delay*, until a new observation occurs:
  - If the controller sticks to a_1, then the plant can choose either to delay (only if a_1 is not enabled) or to do an arbitrary uncontrollable action $a_2 \in Act_u$;
  - If the controller sticks to *delay*, then the plant can choose either to delay or to do an arbitrary uncontrollable action $a_2 \in Act_u$;
- Once a new observation occurs, a next session of controller/plant interaction begins.

By "stick to" an input action, we mean that the controller can arbitrarily repeat that move (and only that move) as long as it is enabled. By "stick to" *delay*, we mean that the controller keeps on delaying (and can only delay).

Definition 4 (play). *Let $\langle l_i, v_i \rangle$ be a TGA state, $c_i \in (Act_c \cup \{delay\})$ be a controller-chosen (or controller-desired) move, and $\sigma_i \in (Act_c \cup Act_u \cup \mathsf{R}_{>0})$ be the actually occurred transition, $i = 0, 1, \ldots, n$. A* play *is a sequence $\rho = \langle l_0, v_0 \rangle c_0 \sigma_0 \langle l_1, v_1 \rangle c_1 \sigma_1 \ldots \langle l_n, v_n \rangle c_n \sigma_n \ldots$, such that for all $i \geq 0$, $\langle l_i, v_i \rangle \xrightarrow{\sigma_i} \langle l_{i+1}, v_{i+1} \rangle$ and*

- *either ($\sigma_i = c_i \in Act_c$, or $\sigma_i \in Act_u$), or ($\sigma_i \in \mathsf{R}_{>0}$ if $\forall 0 \leq t < \sigma_i \,.\, c_i \notin enable(\langle l_i, v_i + t \rangle)$)* [5]*;*
- *if σ_i and σ_{i+1} are both in $\mathsf{R}_{>0}$, then $\xi_P(\langle l_i, v_i \rangle) \neq \xi_P(\langle l_{i+1}, v_{i+1} \rangle)$.* □

[5] If c_i is *delay*, because by definition for any state $\langle l, v \rangle$ we have $delay \notin enable(\langle l, v \rangle)$, then $\sigma_i \in \mathsf{R}_{>0}$ can be any time period that does not incur new observation; if $c_i \in Act_c$, then it means that time period $\sigma_i \in \mathsf{R}_{>0}$ can elapse only when during this period c_i is not enabled. In other words, if the controller prefers *delay*, or her preferred input is not enabled, then the plant can delay.

If a play ρ is a finite sequence, then it is called a *prefix*. The set of all prefixes of a TGA is denoted as Pref.

A *strategy* for a TGA is a function $\lambda : \mathsf{Pref} \to (Act_c \cup \{delay\})$. Thus it is a trace-based (history-based) strategy.

If a player plays the game always according to what a strategy λ suggests her to do, the resulting prefix is called a λ-*supervised prefix*.

Under a set P of observable predicates, an *observation history* is a function $\mathsf{Obs}_P : \mathsf{Pref} \to O_P^*$, which maps a prefix $\rho \in \mathsf{Pref}$ to the chronological sequence $\mathsf{Obs}_P(\rho)$ of non-stuttering observations along ρ.

Let λ be a strategy for the controller. Along any prefix, if the fact that the observation does not change at the next state implies the fact that λ does not suggest a new move at that state, then λ is called an *observation-based stuttering-invariant* (OBSI) strategy.

Given a reachability property $\varphi = \mathsf{control}\text{: } \mathsf{A}\Diamond\ \psi$ which asks "whether ψ can be guaranteed to be satisfied eventually, no matter how the plant behaves", an OBSI strategy λ for the controller is *winning* w.r.t. φ if there exists a λ-supervised prefix which has a trailing observation that satisfies ψ.

Given a network of bounded TGA models of the controller and the plant, a set of observable predicates, and a winning objective in terms of e.g. a reachability or safety property, we check whether the property can be enforced on the models using knowledge-based subset construction and on-the-fly partially observable reachability (OTFPOR) computation [CDL+07], which is based on a mixture of forward search and backward propagation timed game solving algorithm [CDF+05]. If the outcome is positive, a winning strategy for the controller will be extracted from the explored paths.

2.3 An example strategy

For a reachability property φ, a winning OBSI strategy can be represented as a directed acyclic graph which has an initial node corresponding to the initial state, and a number of leaf nodes corresponding to the states that satisfy φ.

For the Smart Lamp example in Fig. 1(a) and 1(b) and the reachability property "$\mathsf{control}$: $\mathsf{A}\Diamond$ `Bright`", Fig. 2 shows a winning OBSI strategy λ_R that is generated by our partially observable timed game solver. Each node in Fig. 2 corresponds to an observation history, and each of its outgoing edges correspond to a macro step. The symbolic state and observation for each strategy node are given in Table 1, where for simplicity instead of listing a complete observation for each node in the "Observations" column, we list only those observable predicates that evaluate to `true` under that observation.

In Fig. 1(a), if we consider the following prefix ρ:

$\langle \texttt{Off}, (x = 0, y = 0)\rangle \cdot delay \cdot 1 \cdot \langle \texttt{Off}, (x = 1, y = 1)\rangle \cdot resety \cdot resety \cdot \langle \texttt{Off}, (x = 1, y = 0)\rangle \cdot delay \cdot 1 \cdot \langle \texttt{Off}, (x = 2, y = 1)\rangle \cdot touch \cdot touch \cdot \langle \texttt{L1}, (x = 0, y = 1)\rangle \cdot delay \cdot 0 \cdot \langle \texttt{Dim1}, (x = 0, y = 1)\rangle \cdot touch \cdot 1 \cdot$

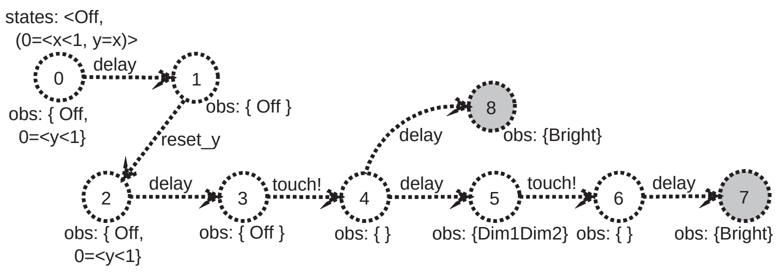


Figure 2: Winning strategy λ_R for reachability property control: A◇ Bright.

Table 1: The states and observations in strategy λ_R (Fig. 2).

Node	**TA symbolic states**	**Observations**
0	$\{\langle\texttt{Off}, (0 \le x < 1, y - x = 0)\rangle\}$	$\{\texttt{Off}, 0 \le y < 1\}$
1	$\{\langle\texttt{Off}, (x = 1, y - x = 0)\rangle\}$	$\{\texttt{Off}\}$
2	$\{\langle\texttt{Off}, (1 \le x < 2, y - x = -1)\rangle\}$	$\{\texttt{Off}, 0 \le y < 1\}$
3	$\{\langle\texttt{Off}, (x = 2, y - x = -1)\rangle\}$	$\{\texttt{Off}\}$
4	$\{\langle\texttt{L1}, (0 \le x \le 2, y - x = 1)\rangle\}$	$\{\}$
5	$\{\langle\texttt{Dim1}, (0 \le x < 3, y - x = 1)\rangle,$ $\langle\texttt{Dim2}, (0 \le x < 3, 2 \le y - x < 4)\rangle\}$	$\{\texttt{Dim1Dim2}\}$
6	$\{\langle\texttt{L2}, (0 \le x \le 2, 2 \le y - x < 5)\rangle\}$	$\{\ \}$
7	$\{\langle\texttt{Bright}, (0 \le x \le 2, 2 \le y - x < 5)\rangle\}$	$\{\texttt{Bright}\}$
8	$\{\langle\texttt{Bright}, (0 \le x \le 2, y - x = 1)\rangle\}$	$\{\texttt{Bright}\}$

Observable predicates: Off: ($\{\texttt{Off}\}$, true), $0 \le y < 1$: $(L, 0 \le y < 1)$, Dim1Dim2: ($\{\texttt{Dim1,Dim2}\}$, true), Bright: ($\{\texttt{Bright}\}$, true).

$\langle\texttt{Dim1}, (x = 1, y = 2)\rangle \cdot touch \cdot touch \cdot \langle\texttt{Dim2}, (x = 0, y = 2)\rangle \cdot touch \cdot 1 \cdot \langle\texttt{Dim2}, (x = 1, y = 3)\rangle \cdot touch \cdot touch \cdot \langle\texttt{L2}, (x = 0, y = 3)\rangle$,

then the observation history of ρ is:

$\{\texttt{Off}, 0 \le y < 1\} \cdot \{\texttt{Off}\} \cdot \{\texttt{Off}, 0 \le y < 1\} \cdot \{\texttt{Off}\} \cdot \{\} \cdot \{\texttt{Dim1Dim2}\} \cdot \{\}$.

The above history corresponds to the trace 0-1-2-3-4-5-6 in Fig. 2. To illustrate the idea of OBSI strategies, let us consider the macro step from node #5 to node #6 in Fig. 2. This macro step has been instantiated from a concrete state $\langle\texttt{Dim1}, (x = 0, y - x = 1)\rangle$ in Fig. 3. The macro step consists of 4 micro steps, during all of which the controller is advised to offer a *touch*!. Unfortunately, in states $\langle\texttt{Dim1}, (x = 0, y - x = 1)\rangle$ ("5a") and $\langle\texttt{Dim2}, (x = 0, y - x = 2)\rangle$ ("5c") the *touch* transitions are not enabled. Thus the controller fails to carry out her move and instead the uncontrollable plant moves (here the 1 time unit delays) actually take place.

Note that there may exist non-deterministic transitions in a strategy, i.e., for a given observation history and for the same controller move, there might exist more than one possible next observation. For instance, in Fig. 2, if the user sticks to the *delay* move at node #4, then there may be a new observation of either that in node #5 or that in node #8. This non-determinism is due to the reaction uncertainties in the TGA models. Considering this non-determinism,

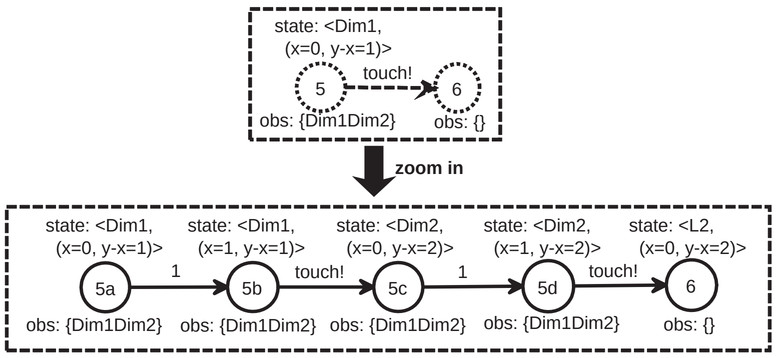

state: <Dim1, (x=0, y-x=1)>
touch!
5
6
obs: {Dim1Dim2}
obs: {}
zoom in
state: <Dim1, (x=0, y-x=1)>
state: <Dim1, (x=1, y-x=1)>
state: <Dim2, (x=0, y-x=2)>
state: <Dim2, (x=1, y-x=2)>
state: <L2, (x=0, y-x=2)>
5a
1
5b
touch!
5c
1
5d
touch!
6
obs: {Dim1Dim2}
obs: {Dim1Dim2}
obs: {Dim1Dim2}
obs: {Dim1Dim2}
obs: {}

Figure 3: Macro step vs. micro step in strategy λ_R.

we represent a strategy λ using a directed graph (N, E) where N is the set of nodes, and $E = N \times (Act_c \cup \{delay\}) \times N$ is the set of edges. An initial node n_0 corresponds to the initial state of the TGA. Each edge $e = (n, mov, n') \in E$ denotes that if at node n the user sticks to mov, then there will be a new observation at node n'.

3 Timed conformance testing

3.1 Test setup

If a winning OBSI strategy for the controller can be synthesized, then it implies that the user-specified property (winning objective) is satisfied by the parallel composition of the plant model and the strategy-constrained controller model. Now our problem is how to test whether the property (referred to as a *test purpose* in this context) is also satisfied by the various implementations IMP of the plant model.

In this paper we propose to use winning OBSI strategies as test cases on the plant implementations. In case a positive test verdict comes out, the IMP is said to satisfy the test purpose; otherwise, it does not satisfy it. Fig. 4 shows our testing framework where on-the-fly partially observable reachability (OTFPOR) computation was discussed earlier, and test execution will be discussed in this section. In Fig. 4, the dashed arrow means that the tester will use the observable predicates to “observe” the IMP, such that she can make a test verdict based on the current observation, or consult the strategy for a next guidance by using the current observation.

According to the conformance testing framework [BAL+90], for each testing method, a suitable implementation relation needs to be adopted. Existing timed conformance relations, e.g., those built on top of trace equivalence [ENDKE98, SVD01] or on top of observable behavior inclusion (e.g., ioco_{DTA} [KJM03], tioco [HLN+03, KT04], tioco_M [BB04], rtioco [LMN04]), are not directly based on the

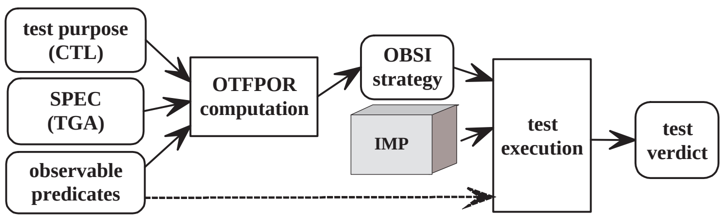


Figure 4: The testing framework.

partial observations that are made on the IMP. In this paper, we view the IMP as a “grey-box” in the sense that some of the IMP internals are “vaguely” observable by the tester (controller). We define a notion of conformance in terms of partial observations. Fig. 5 is a schematic view of observation-based conformance testing of *partially observable* timed systems.

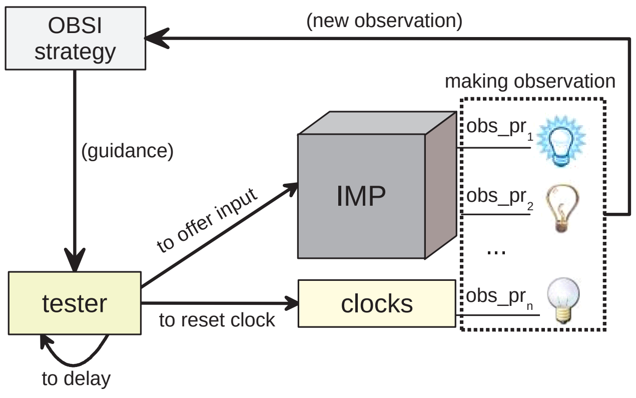


Figure 5: Observation-based conformance testing.

In Fig. 5, each observable predicate $\mathsf{obs\_pr}_i$ can be thought of as a “probe” into the IMP to detect whether the values of system variables and clocks are within some particular intervals, or a “location sensor” to report whether some particular locations have been reached. These could be realistic assumptions, because we are usually only able to make limited-precision measurements, and we usually have some guarding sensors which just monitor whether some valve values are reached or not, and do not care about the exact values at all. In order to keep track of the timing information of the system precisely enough such that the clock constraints in transition guards can be somehow satisfied, the tester might need some “private” clocks that she can read and reset, e.g. clock y in Fig. 1(b). The tester has three choices: to stick to a controllable input action, to reset the clocks, or to keep on delaying.

Given a deterministic TA that has no internal actions, we can prove that any externally observable timed I/O sequence uniquely corresponds to a history of TA state changes under full observability. The relaxation from full observability of the TA internals (i.e., current location and clock values) to partial observability of them enables us to cope with a variety of applications that are previously based on stricter assumptions. To some extent this also helps to realize the potential of design for testability by allowing us to test a family of different IMPs of the same SPEC.

In Fig. 5, the tester assumes the role of the user (controller) in the timed game, and IMP assumes the role of the plant. In the beginning of every tester/IMP interaction session, the tester consults the OBSI strategy using the observation history obtained so far, and in return gets an instruction on what move she is supposed to make during this session: either to stick to a particular controllable input action, or to stick to *delay*. By "stick to" we mean that the tester can arbitrarily repeat the same move (if it is a controllable action then it should be enabled) during the session until a new observation occurs, no matter how the IMP reacts. A new observation signifies the start of a next session of tester/IMP interactions. In Fig. 4, the tester makes test verdicts based on the IMP observations, the OBSI strategy and a conformance relation between the SPEC and the IMP.

3.2 Observation-based conformance

To decide whether the IMP is a correct implementation of the SPEC model, in this paper we define a partial observation-based conformance relation poco.

Similar to the definition of the input-output conformance relation ioco [Tre99], the idea of poco is that after the tester plays an arbitrary number of sessions with the IMP using the moves that are suggested by the strategy and enabled in the SPEC model, each possible next new observation that the IMP could exhibit should be allowed by the SPEC.

Let $\rho = \langle l_0, v_0\rangle c_0\sigma_0\langle l_1, v_1\rangle c_1\sigma_1 \ldots \langle l_n, v_n\rangle$ be a prefix, P be a set of observable predicates, and let $\mu \in (Act_c \cup \{delay\})$ be the move that the controller sticks to during one interaction session. We define the set of possible new prefixes where a new observation has just occurred as

$$\begin{aligned}\rho \ \mathsf{After}_P\ \mu \ &= \ \{\langle l_0, v_0\rangle c_0\sigma_0\langle l_1, v_1\rangle c_1\sigma_1 \ldots \langle l_n, v_n\rangle c_n\sigma_n\langle l_{n+1}, v_{n+1}\rangle c_{n+1}\sigma_{n+1} \\ &\quad\ \ldots \langle l_m, v_m\rangle\},\end{aligned}$$

where for all $n \le i \le m-1$, we have $c_i = \mu$, $\langle l_i, v_i\rangle \xrightarrow{\sigma_i} \langle l_{i+1}, v_{i+1}\rangle$, $\xi_P(\langle l_i, v_i\rangle) = \xi_P(\langle l_n, v_n\rangle)$, $\xi_P(\langle l_m, v_m\rangle) \ne \xi_P(\langle l_{m-1}, v_{m-1}\rangle)$, and the c's and σ's satisfy the constraints in Definition 4.

In the above definition, μ can be extended from a single move to a string of moves in $(Act_c \cup \{delay\})^*$. Specifically, when μ is ε (i.e., an empty string), we define $\rho\ \mathsf{After}_P\ \varepsilon = \{\rho\}$.

For instance, in the example of Fig. 1(a), we have:

$\langle \texttt{Off}, x = 0, y = 0\rangle$ After_P $\mathit{delay} = \{$
$\langle \texttt{Off}, x = 0, y = 0\rangle \cdot \mathit{delay} \cdot 1 \cdot \langle \texttt{Off}, x = 1, y = 1\rangle$,
$\langle \texttt{Off}, x = 0, y = 0\rangle \cdot \mathit{delay} \cdot 0.5 \cdot \langle \texttt{Off}, x = 0.5, y = 0.5\rangle \cdot \mathit{delay} \cdot 0.5 \cdot \langle \texttt{Off}, x = 1, y = 1\rangle$,
$\ldots\}$.

The definition of observation history in Section 2.2 can be extended from a single prefix to a set of prefixes, i.e., $\mathsf{Obs}_P : 2^{\mathsf{Pref}} \to 2^{O_P^*}$. Let $\mathsf{pref\_set} = \{\rho_1, \rho_2, \ldots, \rho_n\}$, we have

$$\mathsf{Obs}_P(\mathsf{pref\_set}) = \bigcup_{\rho_i \in \mathsf{pref\_set}} \mathsf{Obs}_P(\rho_i).$$

For instance, let the initial state in Fig. 1(a) be s_0, then we have

$\mathsf{Obs}_P(s_0$ After_P $\mathit{delay}) = \{\{\texttt{Off}, 0 \leq y < 1\} \cdot \{\texttt{Off}\}\}$; and

$\mathsf{Obs}_P(s_0$ After_P delay resety delay $\mathit{touch!}$ $\mathit{delay}) = \{$
$\{\texttt{Off}, 0 \leq y < 1\} \cdot \{\texttt{Off}\} \cdot \{\texttt{Off}, 0 \leq y < 1\} \cdot \{\texttt{Off}\} \cdot \{\} \cdot \{\texttt{Bright}\}$,
$\{\texttt{Off}, 0 \leq y < 1\} \cdot \{\texttt{Off}\} \cdot \{\texttt{Off}, 0 \leq y < 1\} \cdot \{\texttt{Off}\} \cdot \{\} \cdot \{\texttt{Dim1Dim2}\}$ $\}$.

Definition 5 (Partial Observation-based COnformance relation, poco)**.** *Let ρ, η be two prefixes of the timed game execution, and P be a set of observable predicates. The* partial observation-based conformance *relation* poco_P *between ρ and η is defined as:*

$$\rho \ \mathsf{poco}_P \ \eta \quad \mathit{iff}$$

$$\forall \mu \in (\mathit{Act}_c \cup \{\mathit{delay}\})^* \,.\, (\mathsf{Obs}_P(\rho \ \mathsf{After}_P \ \mu) \subseteq \mathsf{Obs}_P(\eta \ \mathsf{After}_P \ \mu)).$$

□

Let the TLTS of the SPEC model be $\mathcal{S}$ whose initial state is s_0 (note that s_0 is also a prefix); and assume that the behavior of the implementation IMP can be modeled by a TLTS $\mathcal{I}$, who can accept the same set of input actions and has the same set of observable predicates as $\mathcal{S}$; and assume that the initial state of $\mathcal{I}$ is i_0. If i_0 poco_P s_0, then we say $\mathcal{I}$ is a *correct implementation* of $\mathcal{S}$, denoted $\mathcal{I}$ poco_P $\mathcal{S}$.

While traditional conformance relations (such as tioco [HLN+03, KT04], tioco_M [BB04], rtioco [LMN04]) are based on sequences of I/O events, our poco is based on sequences of system state observations. While the former relations accommodate partial observability in terms of internal events only, poco in addition considers the inaccuracies of measurements. Consequently, even if there is no internal action in the system, our poco conformance relation could still be used.

3.3 Test execution algorithms

To execute a winning OBSI strategy as a test case on an IMP, the tester should stick to a specific move as suggested by the strategy. If the observation of the IMP does not change after a micro step, then the tester can offer the same input action to the IMP (if this input action is enabled in the SPEC) again, or delay again. Once the IMP observation changes, the tester checks whether this new observation is allowed by the SPEC model. If allowed, then a next session of tester/IMP interaction will start; otherwise, an error has been revealed. When computing the OBSI strategy, those sets of allowed next observations are included in the strategy, thus during test execution it is not necessary to consult the SPEC model.

Let $obsv(\mathsf{IMP})$ be an instantaneous observation (or a “snapshot”) of the IMP, $\lambda = (N, E)$ be a winning OBSI strategy for a reachability test purpose, and $n_0 \in N$ be the initial strategy node. Let $n \in N$, we define

- the observation at strategy node n as $obs(\lambda, n)$;
- the strategy-suggested move at node n as $move(\lambda, n) = mov \in (Act_c \cup \{delay\})$ such that $\exists n' \in N \,.\, (n, mov, n') \in E$; and
- the set of strategy-allowed next observations following node n as $suc\_obs(\lambda, n) = \{obs(\lambda, n') \mid \exists mov \in (Act_c \cup \{delay\}) \,.\, (n, mov, n') \in E\}$.

Furthermore, we define the following variables:

- *node*: to track the current position in a strategy;
- *imp_obs* and *imp_obs*$'$: to hold recent observations of the IMP; and
- *mov*: to record the strategy-suggested move.

If *imp_obs* satisfies the reachability test purpose, then we say *imp_obs* is a winning observation. For instance, the observations {`Bright`} and {`Bright`, $0 \leq y < 1$} are both winning w.r.t. control: A◇ `Bright`. Algorithm 3.1 describes the test execution towards a reachability test purpose φ. The algorithm takes the IMP and a winning OBSI strategy λ for φ as inputs.

Having said that a winning OBSI strategy for a reachability test purpose is a directed acyclic graph that ends with a number of observations that satisfy the test purpose. Clearly, the strategy provides only finite length guidance for the tester. This ensures that the while loop in Algorithm 3.1 eventually terminates.

In the repeat loop (Main procedure, lines 4-8), there are seemingly zenoness problems and racing problems between the tester and the SUT, i.e., sticking to an input action or clock resets without leading to a new observation might block the time, and sticking to *delay* might require the tester to make infinite frequent observations of the SUT. These, however, can both be avoided. In the former

Algorithm 3.1 **TestExec_Reachability**(λ, IMP)

Input: winning OBSI strategy λ, system implementation IMP;
Output: test verdict pass or fail;
Initialization:

```
 1: $node := n_0$;
 2: $imp\_obs := obsv(\mathsf{IMP})$;
 3: if $imp\_obs \neq obs(\lambda, node)$ then
 4:    return("fail");
```

Main:

```
 1: while ($imp\_obs$ is not winning) do
 2:    $imp\_obs' := imp\_obs$;
 3:    $mov := move(\lambda, node)$;
 4:    repeat
 5:       if $mov \in Act_c$ then
 6:          offer $mov$ to IMP;
 7:       $imp\_obs := obsv(\mathsf{IMP})$;
 8:    until $imp\_obs \neq imp\_obs'$;
 9:    if $imp\_obs \notin suc\_obs(\lambda, node)$ then
10:       return("fail'");
11:    else
12:       $node := n \in N$ such that $obs(\lambda, n) = imp\_obs$ and $(node, mov, n) \in E$; // the
    second $node$ takes the old value
13: endwhile
14: return("pass").
```

case, the generated strategy ensures that there will be no self-loop with any symbolic state; and in the latter case, we can implement the delaying by using the sleep and wake-up mechanisms.

The time complexity of Algorithm 3.1 depends on the length of the strategy, the shape of the strategy (i.e., the sizes of the sets *suc_obs*), and the lengths of the tester/IMP interaction sessions (i.e., how soon will there be a change of IMP observation after the provision of a test stimulus). Let p be the strategy length, $m = max\{\#(suc\_obs(\lambda, i)) \mid 0 \leq i \leq p-1\}$ be the maximal size of those observation sets $suc\_obs(\lambda, i)$. Recall that the clocks in the TGA models are bounded by a maximal constant M, this means that the cumulative delay time between any two adjacent observations in the SPEC TLTS could also be bounded by a constant, say $w = k \cdot \mathsf{M}$. Then the worst-case time complexity of Algorithm 3.1 is $O(p \cdot (w + m))$.

3.4 Soundness and completeness

An ideal test suite or test case should be both sound and complete (or "exhaustive"). A winning OBSI strategy executed as a test case is sound if there is no false positive (i.e, a detected error is indeed an error according to the conformance relation), and it is complete w.r.t. the given test purpose φ if there is no false negative (i.e., if there are non-conforming behaviors in IMP that are related to φ, then there must exist some particular test cases to detect these behaviors).

Let the TLTS of the SPEC TGA be $\mathcal{S}$ whose initial state is s_0, and assume that the behaviors of the implementation IMP can be modeled by a TLTS $\mathcal{I}$ which can accept the same set of input actions and has the same set of observable predicates as $\mathcal{S}$. We let the initial state of $\mathcal{I}$ be i_0.

Theorem 1 (soundness)**.** *If there is a* fail *verdict in test execution, then* $\mathcal{I}$ ~~poco~~ $\mathcal{S}$.

Let φ be a reachability test purpose that is satisfied by the SPEC model. Let $\mathcal{S}_\varphi$ and $\mathcal{I}_\varphi$ be the behaviors of $\mathcal{S}$ and $\mathcal{I}$ that are constrained by φ, i.e., the TLTS's that are obtained by removing those φ-violating runs from $\mathcal{S}$ and $\mathcal{I}$, respectively.

Theorem 2 ((partial) completeness)**.** *If* $\mathcal{I}_\varphi$ ~~poco~~ $\mathcal{S}_\varphi$*, then there exists a winning OBSI strategy (test case)* λ *for* φ *such that a* fail *verdict will occur when executing* λ *with the IMP.*

The proofs can be found in Appendix A.

4 Experimental results

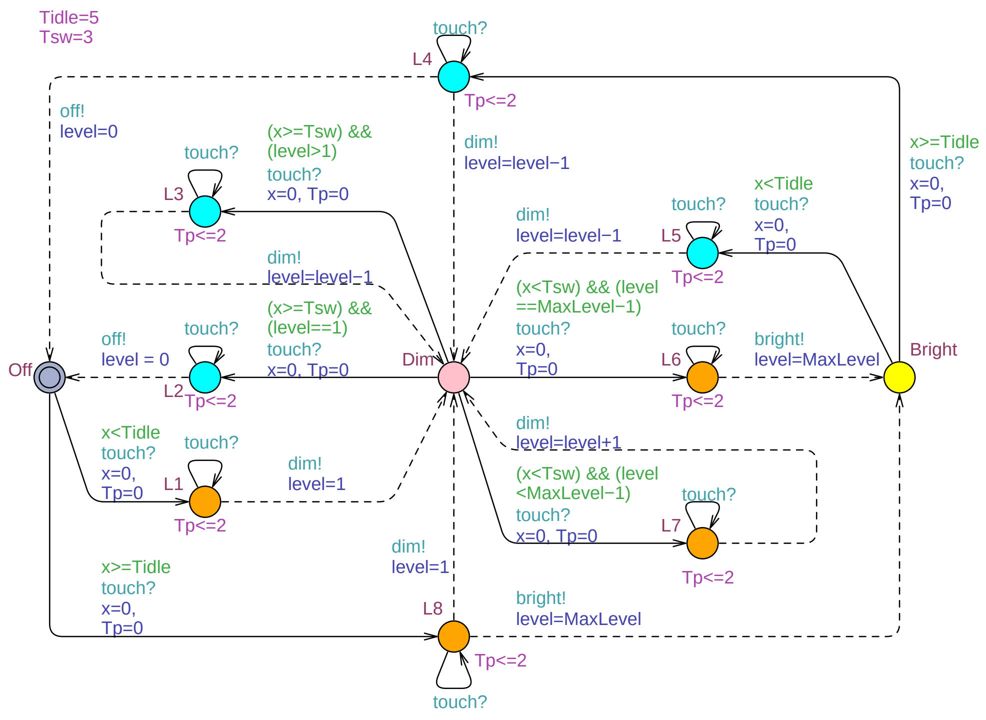
An in-house developed prototype timed game solver for partially observable systems has been implemented in [CDL+07]. It accepts a network of TGAs, a set of observable predicates and an ACTL test purpose as inputs. If the game is solvable, it can generate a winning OBSI strategy. More recently, the functionalities of the prototype tool has been incorporated (re-implemented and improved) into Uppaal-Tiga.

This section reports on some strategy (test case) generation experiments using the prototype tool and Uppaal-Tiga.

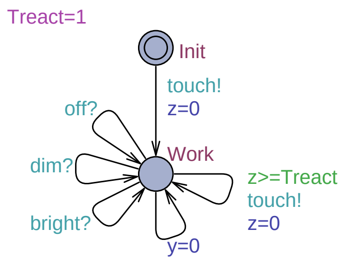
4.1 An extended smart lamp controller

In the example of Fig. 1(a) and 1(b), the Lamp has only four brightness levels: `Off`, `Dim1`, `Dim2` and `Bright`. Now we consider a finer granularity of the brightness levels, say, level 0 is `Off`, level MaxLevel is `Bright`, and there are (MaxLevel $-$ 1) stairs of `Dim` between `Off` and `Bright`. Furthermore, we add clock variables tp and z to the system for finer grained modeling of the timing constraints. Then we obtained the extended TGA models for the smart lamp (Fig. 6(a)) and the user (Fig. 6(b)).

Compared with Fig. 1(a) and 1(b), the extended version has larger state space. There are 4 clocks (x, tp, z and y) in the extended system, and an integer data variable $level \in [0, \mathsf{MaxLevel}]$ is used to represent the current brightness level.



(a) The lamp



(b) The user

Figure 6: A smart lamp system with multiple brightness levels.

Our definition of observable predicates in Section 2.1 can actually be extended with a further dimension of data variable constraints, e.g., $3 \leq var \leq 6$. This extended feature has been implemented in the prototype tool. This enables us to better describe the observations.

In this example, the set of observable predicates is $P = \{$
$(\{\texttt{Off}\}, \texttt{true})$,
$(\{\texttt{Bright}\}, \texttt{true})$,
$(\{\texttt{L1}, \texttt{L6}, \texttt{L7}, \texttt{L8}\}, \texttt{true})$,
$(\{\texttt{L2}, \texttt{L3}, \texttt{L4}, \texttt{L5}\}, \texttt{true})$,
$(\{\texttt{Dim}\}, \mathit{level} = \mathsf{MaxLevel}/2)$,
$(\{\texttt{Dim}\}, \mathsf{MaxLevel}/2 + 1 \leq \mathit{level} \leq \mathsf{MaxLevel} - 1)$,

$(\{\texttt{Dim}\}, 1 \leq \mathit{level} \leq \mathsf{MaxLevel}/2 - 1)$,
$(L, 0 \leq y < \mathsf{M})$ }
where M is the upper bound for clock y.

The test purposes can be formulated as reachability properties (TP_R) as follows:
TP_R control: A$\Diamond$ (`Dim` && ($\mathit{level} = \mathsf{MaxLevel}/2$)).

Here, TP_R asks whether we can guarantee that "Medium Dim" can be observed eventually, no matter how the Lamp behaves.

Table 2 presents the experimental results of test case generation for TP_R. They include the time overheads, memory consumptions and strategy sizes (how many nodes inside the strategy). The leftmost column specifies which test purpose and what MaxLevel values (10, 20, or 30) we are using. Time granularity for the clock-related observable predicate, say $y \in [0, 1)$, is also considered. In Table 2, "/" means running out of memory.

Table 2: Test case generation for extended smart lamp controller.

		$y \in [0, 100)$	[0, 10)	[0, 2)	[0, 1)	[0, 0.5)	[0, 0.1)	[0, 0.01)
		⟨**time** (s), **memory** (KB), **strategy size** (# of nodes) ⟩						
TP_R	10	0.67	0.66	0.87	1.26	1.44	1.38	1.39
		3828	3988	3924	4096	6220	6268	6272
		84	84	138	199	219	220	220
	20	2.30	3.55	3.80	8.26	14.30	18.98	18.86
		6380	7192	7456	11964	15600	17308	17404
		334	468	504	952	1500	1645	1645
	30	5.53	9.29	7.83	20.67	55.25	140.29	137.19
		7824	11468	11520	15556	30068	46992	46736
		709	996	840	1723	3514	5310	5270

Experiment platform: 2×2.00GHz CPU, 1024MB RAM; Ubuntu 8.04, Ruby 1.8.6, Ruby-BDD Binding 0.2, UPPAAL DBM Library 2.0.6, prototype partially observable timed game solver.

The results in Table 2 indicate that winning OBSI strategy generation for non-trivial TGA models are feasible in terms of both time overhead and memory consumption on an ordinary PC. We also find out that, in general, a coarser time granularity might lead to uncontrollability of the problem, whereas a finer granularity demands more resources to generate the test cases.

4.2 A leader election protocol

We also carried out experiments on test case generation for a leader election protocol (LEP) [Lam05b] for mobile ad hoc networks. The TGA models of this protocol have n nodes, and n buffer cells which model the capacity of the communication channels. Each of the nodes and buffers needs one clock. Moreover, there are a global clock for counting the desired election time and a clock specifically for resetting. So there are altogether $(2n + 2)$ clocks. In addition, there are a number of global/local data variables to store and compare the message information. We defined a collection of observable predicates.

The fully observable (TIOGA) and partially observable (PO-TIOGA) models of the LEP protocol and some test purposes are given in Appendix B.

We investigate how test generations based on TIOGA and PO-TIOGA models scale with different system sizes, how they perform under different levels of controllability, and how test generation based on PO-TIOGA models performs under different levels of observability. See Appendix C.

We also make comparison of these two approaches. For the same test purpose, Table 3 shows the results for (4 nodes, 4 buffers) and (5 nodes, 5 buffers), respectively. More details can be found in Appendix C.

Table 3: Full vs. partial observation-based test case generation.

	full observation			partial observation		
	time (s)	**memory** (MB)	**strategy** (# of nodes)	**time** (s)	**memory** (MB)	**strategy** (# of nodes)
$(4, 4)$	25.15	179.0	72300	18.33	168	56
$(5, 5)$	(out of memory)			137.34	761	81

Experiment platform: Sun Fire X4100, 2×2.4GHz CPU, 4096MB RAM;
Suse Linux Enterprise Desktop 10 (64bit), Uppaal-Tiga 0.13.

A remarkable finding is that the partial observation-based approach generates much smaller (shorter) test cases. This is in accordance with the rationale of our approach, because it in general makes a much coarser partitioning of the model state space than the full observation-based approach.

5 Conclusions and future work

We discuss the problem of model-based conformance testing of partially observable timed systems. The systems are modeled as a network of time game automata; the partial observability is characterized by using a set of predicates on the system states; and the test purposes are formulated as ACTL formulas that specify reachability properties. We use a previously-developed partially observable timed game solver to generate winning OBSI strategies for the test purposes, which in this paper are used as test cases for real-time conformance testing. We present a test framework, define a conformance relation poco, develop test execution algorithms, and prove the soundness and completeness properties. Experiments on test generations for some non-trivial examples show that this method yields encouraging results.

This method has some necessary ingredients to be used in software product line engineering. By modeling the variabilities of the product lines using the reaction uncertainties of our TGA models, we can generate test cases for testing a family of similar but different software products against their (common) high-level requirements.

Compared with full observation-based conformance testing, the approach in this paper requires weaker assumptions and seems to generate smaller (shorter) test cases. These suggest that partial observation-based monitoring and testing has better prospects of being practically useful and being industrially adopted.

Future work includes: (1) More and larger case studies on test generation. This might involve improving the game solver towards better performance (especially memory-wise performance), better scalability and further optimized strategies; (2) To provide guidelines for creating a sufficient/correct/consistent set of observable predicates; (3) Testing (or validating) towards safety test purposes, whose corresponding OBSI strategies may provide infinite guidance to the tester.

Appendix A: Proofs in Section 3.4

To prove the theorems, we need some definitions and (re-)formalization of some already-mentioned concepts.

For simplicity, we assume that a set P of observable predicates have been fixed. Therefore, we do not put P as the subscript of After, Out, Obs, etc., if it clear from the context.

Given (a prefix of) a play ρ, we define $\rho(n)$ to be the prefix up to $\langle l_n, v_n\rangle$, and define $last(\rho(n)) = \langle l_n, v_n\rangle$.

A state of a prefix where the observation changes is called a *choice point* [CDL$^+$07]. The set of all choice points on prefix ρ is denoted $\mathsf{ChoicePoint}(\rho) = \{0\} \cup \{i \mid \gamma^{-1}(\langle l_i, v_i\rangle) \neq \gamma^{-1}(\langle l_{i-1}, v_{i-1}\rangle), i \geq 1\}$. We define $max(\mathsf{ChoicePoint}(\rho))$ to be the trailing choice point.

An *observation history* is a function $\mathsf{Obs} : \mathsf{Pref} \to O^*$, which maps a prefix ρ to the chronological sequence $\mathsf{Obs}(\rho)$ of observations on the choice points of ρ.

Let λ be a strategy for the controller. For all $\rho_1, \rho_2 \in \mathsf{Pref}$, if $\mathsf{Obs}(\rho_1(max(\mathsf{ChoicePoint}(\rho_1)))) = \mathsf{Obs}(\rho_2(max(\mathsf{ChoicePoint}(\rho_2)))) \Rightarrow \lambda(\rho_1) = \lambda(\rho_2)$, then λ is said to be an *OBSI strategy.* In other words, if the observation does not change at the next state, then the strategy does not suggest a new move at that state.

Definition 5 decides conformance by checking the inclusion of sets of observation histories, which contain all the observation information from the very beginning. To facilitate run-time conformance checking, we can refine the definition of the poco relation using the inclusion of the sets of the "latest" observations that are obtained by sticking to an arbitrary string of moves:

$$\rho \ \mathsf{poco}\ \eta \quad \text{iff} \quad \forall \mu \in (Act_c \cup \{\mathsf{delay}\})^* \,.\, \{\gamma^{-1}(last(\rho'(max(\mathsf{ChoicePoint}(\rho'))))) \mid \rho' \in (\rho \ \mathsf{After}\ \mu)\} \subseteq \{\gamma^{-1}(last(\eta'(max(\mathsf{ChoicePoint}(\eta'))))) \mid \eta' \in (\eta \ \mathsf{After}\ \mu)\}.$$

Given a system specification $\mathcal{S}$ and a test purpose (winning objective) φ, we use $\mathsf{Strategy}(\mathcal{S}, \varphi)$ to denote the set of all winning OBSI strategies.

Let the TLTS of the SPEC TGA be $\mathcal{S}$, whose initial state is s_0, and assume that the behavior of the implementation IMP can be modeled by a TLTS $\mathcal{I}$, which can accept the same set of input actions and has the same set of observable predicates as $\mathcal{S}$. We let the initial state of $\mathcal{I}$ be i_0.

Theorem 1 (soundness). If there is a fail verdict in test execution, then $\mathcal{I} \ \cancel{\mathsf{poco}}\ \mathcal{S}$.

Proof. (sketch.) If the test purpose φ is a reachability property, then there are two possibilities for a fail verdict in Algorithm 3.1. The first case is that the initial observation o_{i_0} of the IMP is different from the first observation o_{s_0} in the strategy, which is also the initial observation of the SPEC model. Since $o_{i_0} \neq o_{s_0}$, we get $\{o_{i_0}\} \nsubseteq \{o_{s_0}\}$, thus $\mathsf{Obs}(i_0 \ \mathsf{After}\ \varepsilon) \nsubseteq \mathsf{Obs}(s_0 \ \mathsf{After}\ \varepsilon)$. The second case is that after the tester sticks to a number of moves $\mu \in (Act_c \cup \{\mathsf{delay}\})^*$, a new observation o of the IMP occurs but it is disallowed by the strategy, i.e., $\exists \mu \in (Act_c \cup \{\mathsf{delay}\})^* \,.\, \exists \rho \in (i_0 \text{ After } \mu) \,.\, \forall \eta \in (s_0 \text{ After } \mu) \,.\, (\gamma^{-1}(last(\rho(max(\mathsf{ChoicePoint}(\rho)))))$ $\neq \gamma^{-1}(last(\eta(max(\mathsf{ChoicePoint}(\eta))))))$. By (the refined) Definition 5, i_0 ~~poco~~ s_0. In both cases, we conclude that the IMP does not comply with the SPEC. $\square$

Let φ be a reachability test purpose that is satisfied by the SPEC model. Let $\mathcal{S}_\varphi$ and $\mathcal{I}_\varphi$ be the behaviors of $\mathcal{S}$ and $\mathcal{I}$ that are constrained by φ, i.e., the TLTS's that are obtained by removing those φ-violating runs from $\mathcal{S}$ and $\mathcal{I}$, respectively.

Theorem 2 ((partial) completeness). If $\mathcal{I}_\varphi$ ~~poco~~ $\mathcal{S}_\varphi$, then there exists a winning OBSI strategy (test case) λ for φ such that a fail verdict will occur when executing λ with the IMP.

Proof. (sketch.) By $\mathcal{I}_\varphi$ ~~poco~~ $\mathcal{S}_\varphi$, we know that $\forall \lambda \in \mathsf{Strategy}(\mathcal{S}, \varphi) \,.\, \exists \mu \in (Act_c \cup \{\mathsf{delay}\})^* \,.\, \{\gamma^{-1}(last(\rho(max(\mathsf{ChoicePoint}(\rho))))) \mid \rho \in (i_0 \text{ After } \mu)\} \nsubseteq \{\gamma^{-1}(last(\eta(max(\mathsf{ChoicePoint}(\eta))))) \mid \eta \in (s_0 \text{ After } \mu) \cap \mathsf{Sup}^\lambda(s_0, \mathcal{S})\}$. This means that among those prefixes obtained after applying a string μ of moves on the IMP, there is a sub-class of (infinitely many) prefixes whose last stuttering observations are disallowed by the constrained SPEC model. Pick an arbitrary one from them, say ρ. According to Algorithm 3.1, ρ will end up with a fail verdict. $\square$

Appendix B: Fully and partially observable models of the leader election protocol

We will build fully observable and partially observable models of the leader election protocol (LEP) [Lam05b, Lam05a].

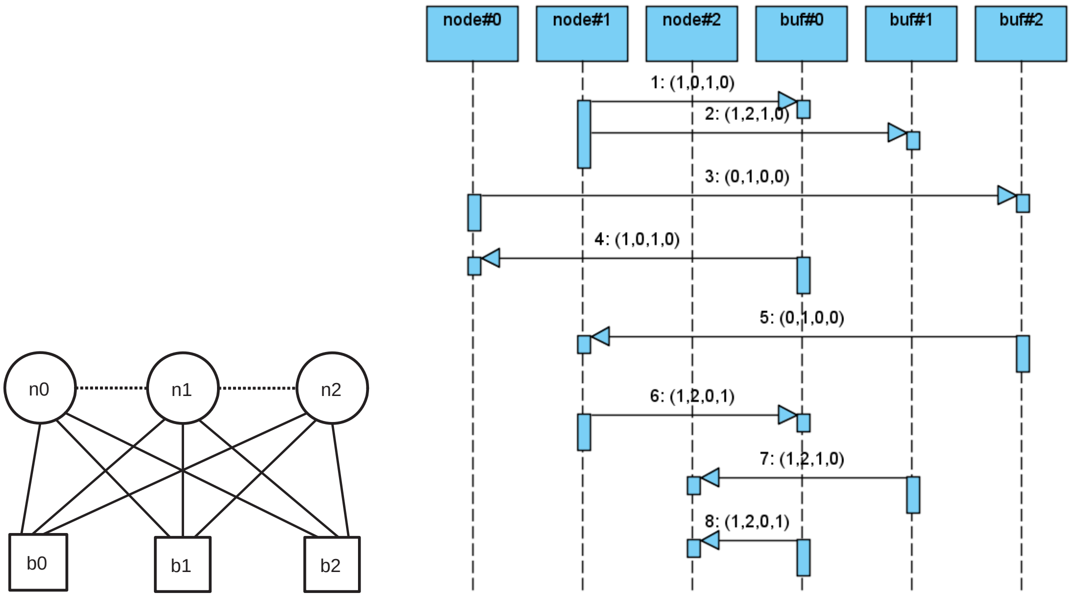
The problem descriptions of the LEP protocol can be found in the Appendix of Paper C, where fully observable models for LEP have already been constructed. In this paper we make an alternative modeling of the problem. Both the nodes and the buffers will be *explicitly* modeled.

System overview

The LEP protocol entities include a number of protocol nodes. A node may send out a message, which might be further forwarded to its neighboring nodes. A data link between two neighboring nodes is modeled as a buffer, which accepts an incoming message and ensures that it will be properly delivered to the destination node within the deadline. To model the different actual transmission delays and thus the possible re-orderings of the messages, we model the media as a "bag" rather than a queue (or a stack) of buffer cells. Since the media is of limited capacity, we assume that the number of buffers is bounded by a constant B. In other words, at any time there could be no more than B messages simultaneously in transmission.

Based on the above design decisions, nodes and buffers are identified as the protocol entities, whereas messages are viewed as datagrams that are transmitted among them.

Fig. 7(a) is an example of an LEP system that consists of 3 nodes (linearly connected) and 3 buffers. If there is a dashed line connecting two nodes, then it means that there is a network link between them. A solid line between a node and a buffer means that the buffer can accept a message from and can deliver a message to the node. Fig. 7(b) is a possible scenario of leader election where node#0 is finally elected as the leader.



(a) system structure (b) a possible scenario of leader election

Figure 7: A leader election protocol.

Fully observable (TIOGA) model

In UPPAAL we use two templates: Node for those protocol nodes and Buffer for those messages in transmission. These two templates can both be instantiated to represent the concrete protocol entities.

Since each node has a timeout mechanism, it should have a clock, say *idleClock*, which denotes how long this node has been waiting for a "good" message. Since each message has a maximal transmission delay, each buffer should also have a clock, say *msgTransClock*, which denotes how long this message has been in transmission. For verification purpose we need an auxiliary clock *globalClock*, which is never reset, to record how much time has elapsed since the beginning.

If a node is controllable (i.e., belonging to the tester part), then all the output transitions are in solid lines, otherwise they are in dashed lines.

For example, the UPPAAL global declarations of an LEP system with 3 nodes (linearly connected) and 3 buffers are shown in Listing 1.

Listing 1: UPPAAL global declarations for the LEP system.

```
const int  MaxNodeId = 2; // the number of nodes is (MaxNodeId+1)
const int  MaxBufferId = 2; // the number of buffers  is  (MaxBufferId+1)
const int  MaxDistance = MaxNodeId;

typedef  int  [0,  MaxNodeId]  NodeId;
typedef  int  [0,  MaxBufferId] BufferId ;  // the  "capacity"  of  media
typedef  int  [0,  MaxDistance] Distance;  // the  hops between two nodes.

// timing  parameters
const int  INIT_TIMEOUT = 10;
const int  TIMEOUT_DELAY = 5;
const int  MSG_DELAY = 3;

typedef  struct  {
   NodeId src;
   NodeId dest;
   NodeId leader;
   Distance distance ;
} Message;

const Message nullMsg = {0, 0, 0,  0};
Message envMsg;

meta bool inUse[MaxBufferId+1]; // whether a buffer  is  occupied.
BufferId  nextBuffer ;  // to  store  a  message into  this  buffer .

chan deliverMsg;  // Buffer  −− >Node
chan sendMsg; // Node −− >Buffer

clock  globalClock ;  // for  verification    purpose

// network topology  (here  linear  )
meta bool topology[MaxNodeId + 1][MaxNodeId + 1] = { {1, 1, 0},
                                                      {1,  1,  1},
                                                      {0,  1,  1} };
```

The UPPAAL template of a node is Node(const NodeId *myId*), where the parameter *myId* can be instantiated to $0, 1, 2, \ldots$. The UPPAAL local declarations for a node are shown in Listing 2.

Listing 2: Uppaal local declarations for a node.

```
NodeId believedLeader = myId; //Initially , each node sets himself as the leader.
Distance distToBlvLeader = 0;

int [0, INIT_TIMEOUT + TIMEOUT_DELAY +
        MaxDistance*MSG_DELAY] Timeout = INIT_TIMEOUT;

Message rMsg = nullMsg; //The received message

//How long since last reception of a "good" massage, or since last "timeout"
clock idleClock;

bool betterInfo = false;
bool lossOfMsg = false;

int [0, MaxNodeId + 1] forwardee = 0; // to iterate over all the nodes
int [0, MaxBufferId + 1] slot = 0; // to find an available buffer
```

As can be seen from Listing 2, each node maintains its own clock *idleClock* which records how much time has elapsed since the last timeout of this node, or since the last reception of a "good" message by this node.

Fig. 8 and Fig. 9 are the TIOGAs of a controllable and an uncontrollable Node template, respectively.

The Uppaal template of a buffer is Buffer(const BufferId *myId*), where the parameter *myId* can be instantiated to 0, 1, 2, ... The Uppaal local declarations for a buffer are shown in Listing 3.

Listing 3: Uppaal local declarations for a buffer.

```
Message bufferMsg = nullMsg;

// to record the time that a message has been in transmission
clock msgTransClock;
```

Fig. 10(a) and Fig. 10(b) are the TIOGAs of a controllable and an uncontrollable Buffer template, respectively, where message transmission is bounded by a delay of MSG_DELAY time units.

Since the number of buffers is fixed, if a node wants to send a message to its neighbors but finds out that currently there is no available buffer, then the message will be discarded (i.e., get lost).

Note that in TIOGA models there is no internal action. In order not to clutter up the figures, we omit a *toDeliver*? action at each location of Node, and omit each of the following actions at each location of Buffer: *timeout*?,

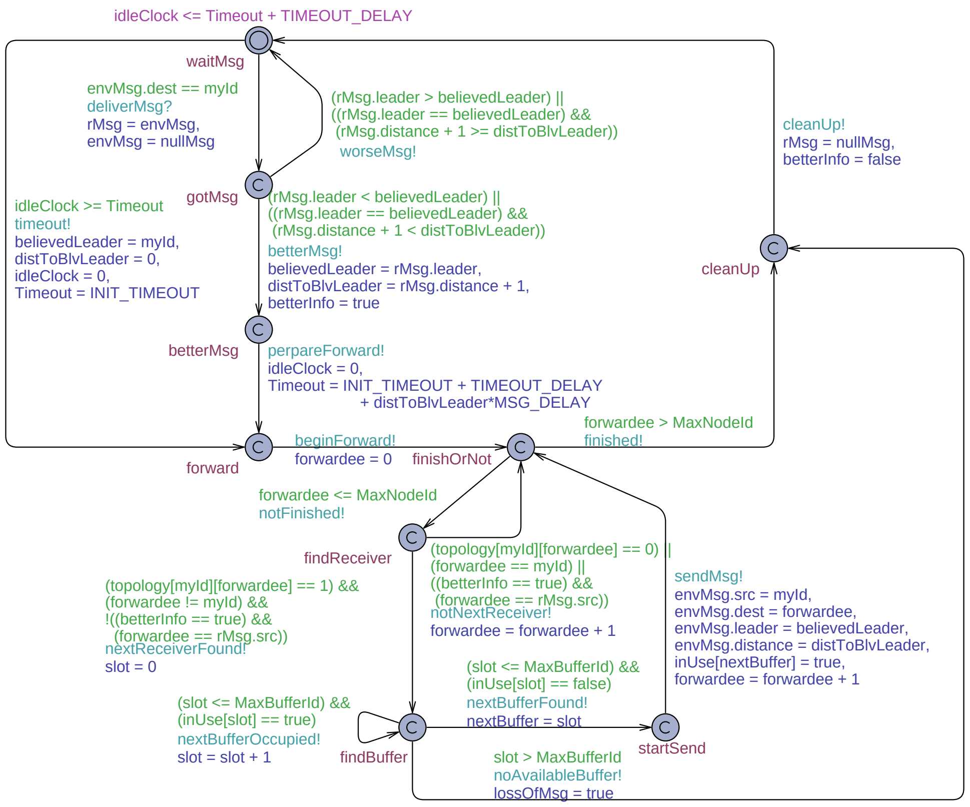


Figure 8: TIOGA of a controllable node

worseMsg?, *betterMsg*?, *prepareForward*?, *beginForward*?, *finished*?, *notFinished*?, *notNextReceiver*?, *nextReceiverFound*?, *nextBufferOcupied*?, *nextBufferFound*?, *noAvailableBuffer*? and *cleanUp*?.

Partially observable (PO-TIOGA) model

The PO-TIOGA model is the TIOGA model that are relaxed with internal transitions and associated with a set of observable predicates that are declared on the model state space, e.g.,

- whether an SUT node is in some particular sets of possible locations; and/or
- whether the value of some clock falls in some particular intervals; and/or
- whether the value of some data variable falls in some particular intervals.

In addition to the clocks for each node and buffer and the *globalClock*, we may need yet another clock to control the timing in the OBSI strategy. For instance, we can have the following set P1 of observable predicates, where y is the newly introduced auxiliary clock for PO-TIOGA:

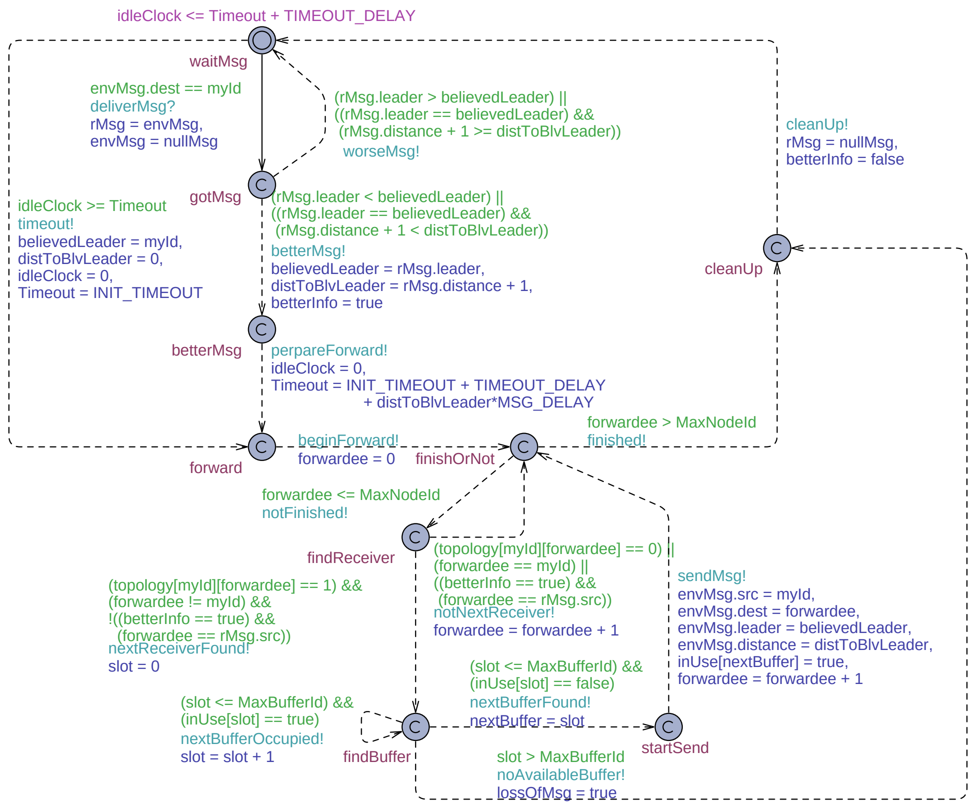


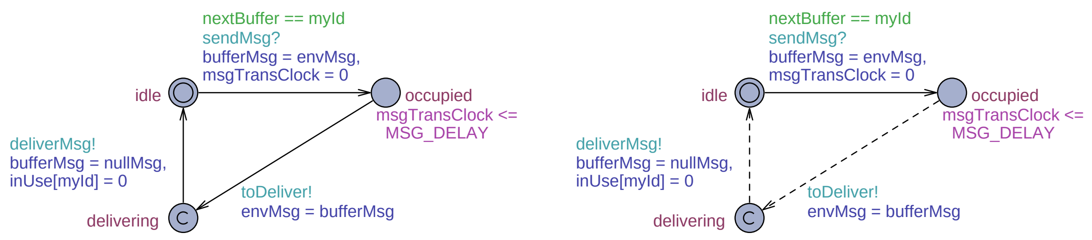
Figure 9: TIOGA of an uncontrollable node

$$
\begin{aligned}
\mathsf{P1} \;=\; & \{Node0.waitMsg, Node0.finishOrNot, Node0.findReceiver, Node0.startSend,\\
& Node1.waitMsg, Node1.betterMsg, Node1.finishOrNot, Node1.findReceiver,\\
& Node1.startSend, (y >= 0)\ \&\&\ (y < 1), (globalClock < \mathsf{INIT\_TIMEOUT} +\\
& \mathsf{TIMEOUT\_DELAY} + \mathsf{MaxDistance} * \mathsf{MSG\_DELAY} + 1)\ \&\&\\
& (Node0.believedLeader == 0)\ \&\&\ (Node1.believedLeader == 0)\ \&\&\\
& (Node2.believedLeader == 0)\},
\end{aligned}
$$

where MaxDistance is the maximal number of hops from one node to another node in the network.

In the TIOGA modeling of the LEP system, the nodes and the buffer actually communicate only through the channels *sendMsg* (from node to buffer) and *deliverMsg* (from buffer to node). Since internal transitions are allowed in PO-TIOGA models, we replace all transitions except for those of *deliverMsg* and *sendMsg* with the corresponding internal transitions both in Node and in Buffer.

After associated with a set of observable predicates, we get the PO-TIOGA model of the LEP system.



(a) controllable buffer (b) uncontrollable buffer

Figure 10: TIOGAs of a buffer.

Test purpose specification

We can define test purposes as reachability or safety properties. We are most interested in the property "leader can always be elected within a period of time". Suppose we have 3 nodes. For TIOGA, this property can be formulated as:

TP1 = control: A◇ (*globalClock* <= INIT_TIMEOUT + TIMEOUT_DELAY + MaxDistance $*$ MSG_DELAY) && $(Node0.believedLeader == 0)$ && $(Node1.believedLeader == 0)$ && $(Node2.believedLeader == 0)$.

The functional and timing properties that we could be interested in the LEP protocol include:

- Prop1 ("leader always eventually elected"): The system is supposed to eventually arrive at a state where the believed leader of each node is the lowest numbered one, despite all possible uncontrollable behaviors;
- Prop2 ("leader always elected within a period of time"): It is always the case that the correct leader is known at each node i after (INIT_TIMEOUT + TIMEOUT_DELAY + MaxDistance * MSG_DELAY) time units, where MaxDistance is the maximal number of hops (direct links) from node i to the leader node;
- Prop3 ("messages never lost"): There are always sufficiently many buffers for use, and therefore messages will not be dropped;
- Prop4 ("always eventually loss of message"): It is always possible that some node in the future will drop messages because of the unavailability of empty buffers; and
- Prop5 ("buffers always eventually used out"): All buffers will eventually be occupied.

For PO-TIOGA it is a little bit different: the *globalClock* cannot have a strict upper bound in the predicate:

TP2 = control: $A\Diamond$ $(globalClock <$ INIT_TIMEOUT + TIMEOUT_DELAY + MaxDistance $*$ MSG_DELAY $+ 1)$ && $(Node0.believedLeader == 0)$ && $(Node1.believedLeader == 0)$ && $(Node2.believedLeader == 0)$.

Appendix C: Quantitative evaluation of test generation with the leader election protocol

The performance of test generation is jointly determined by many factors, such as the topology and layout of the network, the timing parameters, the means of communication (handshaking or shared variables), system size (the numbers of nodes and buffers), the degree of controllability and the degree of observability. This section examines how the latter three factors influence the performance.

Scalability

For the TIOGA model and the test purpose TP1 in Appendix B, we consider linear network topology, i.e., nodes $0, 1, \ldots, n$ are linearly connected. The timing parameters INIT_TIMEOUT, TIMEOUT_DELAY and MSG_DELAY are chosen to be 10, 5 and 3, respectively. The SUT consists of only one node (i.e., the largest numbered node). Since it is a TIOGA model, it is fully observable. Tables 4 reports the performance results of test generation for different system sizes. In the table / means "out of memory".

Table 4: Test generation from TIOGA models for different system sizes.

#bufs	3 nodes (linear)			4 nodes (linear)			5 nodes (linear)		
	size	time(s)	mem(KB)	size	time(s)	mem(KB)	size	time(s)	mem(KB)
2	897	0.09	120	9155	2.63	23420	98134	135.75	506508
3	1739	0.16	5140	33873	7.90	62072	179095	380.39	1847460
4	2383	0.23	5860	72300	25.15	183252	/	/	/
5	2383	0.25	6088	94435	49.07	337132	/	/	/

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); UPPAAL-TIGA 0.13.

As can be seen from Table 4, the models with more nodes and buffers are more resource-demanding. This is reasonable because each node and buffer should be associated with a clock and a number of data variables.

For the PO-TIOGA model and the test purpose TP2 in Appendix B, we use the same system configuration as for TIOGA. But this time the PO-TIOGA model is only partially observable. Table 5 reports the experimental results of test generation for different system sizes.

Table 5: Test generation from PO-TIOGA models for different system sizes.

#bufs	3 nodes (linear)			4 nodes (linear)			5 nodes (linear)		
	size	time(s)	mem(KB)	size	time(s)	mem(KB)	size	time(s)	mem(KB)
2	67	0.99	27400	56	25.43	112976	81	85.41	416872
3	67	1.11	33924	56	20.56	138328	81	98.83	522872
4	67	1.28	41632	56	18.33	172224	81	113.73	646512
5	67	1.50	49944	56	19.41	203180	81	137.34	779124

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); UPPAAL-TIGA 0.13.

As can be seen from Table 5, when the number of nodes increases, time overhead and memory consumption increases rapidly, but strategy size does not increase accordingly. This is because that strategy size mainly depends on the number of observable predicates and their Cartesian space, rather than on the TIOGA state space, therefore it is not so sensitive to the number of clocks in the system.

It seems abnormal that the test cases for 3-nodes are larger than for 4 nodes. The reason is that in our experiments, it is a branching strategy for the 3 nodes case, whereas a linear strategy for the 4 nodes case.

With the increase of the number of buffers, the time overhead and memory consumption do not increase rapidly. This is because that in this LEP case study, most of the observable predicates are about the nodes and there is no predicate about the buffers.

Degree of controllability

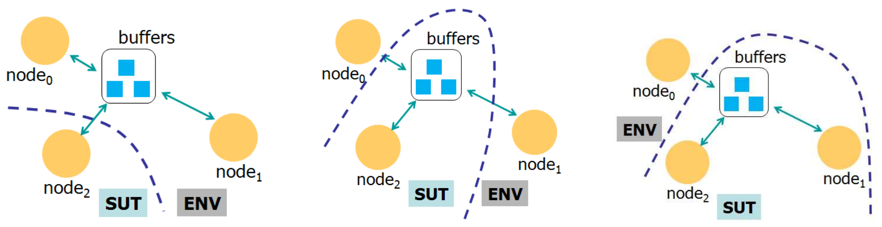
By *degree of controllability* we mean how many components in the LEP system are controllable by the tester, or how many LEP components constitute the tester (ENV) part. This concerns the tester/SUT partitioning of the LEP system. We consider the following partitioning schemes:

- scheme (a): only one node as the SUT, and all other nodes and all buffers at the tester side. See Fig. 11(a). In the subsection of "Scalability", we have the results for this scheme;

- scheme (b): one node and all buffers as the SUT, and all other nodes at the tester side. See Fig. 11(b); and

- scheme (c): two nodes and all buffers as the SUT, and all other nodes at the tester side. See Fig. 11(c).

Clearly, in scheme (a) the SUT is most controllable, and in scheme (c) it is least controllable.



(a) 1 node as the SUT (b) 1 node + all buffers (c) 2 nodes + all buffers

Figure 11: The partitioning of nodes and buffers into SUT and its environment (ENV).

Table 6 presents the results of test generation for TIOGA models with the different schemes of SUT/ENV partitioning. The protocol nodes are linearly connected and the timing parameters are 10, 5 and 3.

Table 6: Test generation from TIOGA models for different degrees of controllability.

controll-	3 nodes+3 bufs(linear)			4 nodes+4 bufs(linear)			5 nodes+5 bufs(linear)		
ability	size	time(s)	mem(KB)	size	time(s)	mem(KB)	size	time(s)	mem(KB)
(a)	1739	0.16	5140	72300	25.15	183252	/	/	/
(b)	2102	0.30	6640	439269	703.08	3464788	/	/	/
(c)	3350	7.64	64084	/	/	/	/	/	/

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); Uppaal-Tiga 0.13.

As can be seen from Tables 6, the more uncontrollable the SUT is, the more resource-demanding the test generation is. Controllability is a crucial factor that affects the performance of test generation.

Similarly, Table 7 presents the results for PO-TIOGA models. As can be seen from Table 7, when SUT's degree of uncontrollability increases, the time overhead and the memory consumption increase rapidly. However, the strategy size has no such tendency. This is because the number of observable predicates is not increasing accordingly.

Table 7: Test generation from PO-TIOGA models for different degrees of controllability.

controll-	3 nodes+3 bufs(linear)			4 nodes+4 bufs(linear)			5 nodes+5 bufs(linear)		
ability	size	time(s)	mem(KB)	size	time(s)	mem(KB)	size	time(s)	mem(KB)
(a)	67	1.11	33924	56	18.33	172224	81	137.34	779124
(b)	31	4.96	95060	51	216.28	805864	/	/	/
(c)	/	/	/	/	/	/	/	/	/

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); Uppaal-Tiga 0.13.

Degree of observability

By *degree of observability* we mean how many Points of Observation we can have on the SUT. This concerns the level of detailedness that observations can be made on the LEP system.

Let us consider an LEP system with 3 nodes and 3 buffers, where Node2 and all buffers constitute the SUT (i.e., scheme (b) in Fig. 11(b)). We define different degrees of observability:

- P-: (using none of the predicates in P1)
- P1: (i.e., the P1 in Appendix B)
- P2: (P1 + *Node0.betterMsg*)
- P3: (P2 + *Node2.waitMsg*)
- P4: (P3 + *Node2.betterMsg* + *Node2.finishOrNot* + *Node2.findReceiver* + *Node2.startSend*)
- P5: (P4 + *Buffer0.occupied* + *Buffer1.occupied* + *Buffer2.occupied*)

It is obvious that the LEP system is most observable with P5 and least observable with P-. Table 8 presents the results for these different degrees of observability.

Table 8: Test generation from PO-TIOGA models for different degrees of observability.

observability	3 nodes + 3 buffers (linear)		
	size	time(s)	mem(KB)
P-	(game not solvable)		
P1	31	4.26	86892
P2	31	4.96	94892
P3	34	12.04	107248
P4	44	20.15	127164
P5	155	48.28	220560

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); UPPAAL-TIGA 0.13.

As can be seen from Table 8, with the increase of the degree of observability, test generation has an increasing demand for resources.

From Table 8 we also learn that: if too few predicates are defined (like the case of P-), then the game may be not solvable. If too many predicates are defined (like the case of P5, where the strategy branches several times), then the test generation may be too resource-demanding and the strategy may be a little bit too large. Therefore, it is important to define a sufficiently but not excessively large set of predicates.

Full vs. partial observability

The full observability of TIOGA can be viewed as an extreme case of the partial observability of PO-TIOGA by assuming that we have a most detailed set of observable predicates, i.e., we have:

- a predicate (`in` $C_i.l_j$) for each TA location l_j of each LEP component C_i;
- a predicate $x \in [k, k+1)$ for each possible value k of each clock x; and
- a predicate $v == m$ for each possible integer/boolean value m of each data variable v.

We consider the LEP system with nodes linearly connected, with only one node as the SUT and with timing parameters of 10, 5 and 3. We carried out comparative studies of test generation based on TIOGA and PO-TIOGA models. Table 9 presents the results.

Table 9: Test generation from TIOGA vs. from PO-TIOGA models.

model	3 nodes+3 bufs(linear)			4 nodes+4 bufs(linear)			5 nodes+5 bufs(linear)		
	size	time(s)	mem(KB)	size	time(s)	mem(KB)	size	time(s)	mem(KB)
TIOGA	1739	0.16	5140	72300	25.15	183252	/	/	/
PO-TIOGA	67	1.11	33924	56	18.33	172224	81	137.34	779124

Experiment platform: Sun Fire X4100, 2x2.4GHz CPU, 4096MB RAM; Suse Linux Enterprise Desktop 10 (64bit); UPPAAL-TIGA 0.13.

As can be seen from Table 9, with the increase of the numbers of nodes and buffers, PO-TIOGA-based test generation scales better than TIOGA-based method. Furthermore, PO-TIOGA-based method generates much smaller test cases. The reason is that these two methods use different game solving algorithms, and the former one generates strategies based on much smaller state space.

Bibliography

[AAG$^+$07] Yasmina Abdeddaïm, Eugene Asarin, Matthieu Gallien, Félix Ingrand, Charles Lesire, and Mihaela Sighireanu. Planning robust temporal plans: A comparison between cbtp and tga approaches. In *Proc. 17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pages 2–9, 2007.

[AB99] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proc. 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE'99)*, pages 239–248, 1999.

[ABL98] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In *Proc. 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, pages 263–280, 1998.

[ABM98] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, 1998.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.

[ACY95] Rajeev Alur, Costas Courcoubetis, and Mihalis Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proc. 27th Annual ACM Symposium on Theory of Computing (STOC'95)*, pages 363–372, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AFH94] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. A determinizable class of timed automata. In *Proc. 6th International Conference on Computer Aided Verification (CAV'94)*, pages 1–13, 1994.

[AH97] Rajeev Alur and Thomas A. Henzinger. Real-time system = discrete system + clock variables. *Software Tools for Technology Transfer (STTT)*, 1(1-2):86–109, 1997.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[Alu99] Rajeev Alur. Timed automata. In *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, pages 8–22, 1999.

[AM04] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In Bernardo and Corradini [BC04], pages 1–24.

[AMB$^+$04] Alain Abran, James W. Moore, Pierre Bourque, Robert Dupuis, and Leonard L. Tripp. *Guide to the Software Engineering Body of Knowledge (SWEBOK).* IEEE Press, Piscataway, NJ, USA, 2004.

[AMPS98] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proc. 5th IFAC Conference on System Structure and Control (SSC'98)*, pages 469–474. Elsevier Science, July 1998.

[AR04] George S. Avrunin and Gregg Rothermel, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. ACM, 2004.

[AT02] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In *Proc. 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, july 2002.

[Aut] Automotix. Lincoln continental problems - safety recalls and defects. `http://www.automotix.net/autorepair/recalls/lincoln-continental/`, accessed August 2009.

[BAL$^+$90] Ed Brinksma, Rudie Alderden, Rom Langerak, Jeroen van de Lagemaat, and Jan Tretmans. A formal approach to conformance testing. In *Proc. 2nd Int. Workshop on Protocol Test Systems*, pages 349–363, 1990.

[Bal09] Sandie Balaguer. Extending uppaal with scenario-oriennted verification. Master's thesis, École Centrale de Nantes, Nantes, France, September 2009.

[BB04] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In Grabowski and Nielsen [GN05], pages 64–78.

[BB05] Henrik C. Bohnenkamp and Axel Belinfante. Timed testing with torx. In Fitzgerald et al. [FHT05], pages 173–188.

[BC04] Marco Bernardo and Flavio Corradini, editors. *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science.* Springer, 2004.

[BCD+07] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In *Proc. 19th International Conference on Computer Aided Verification (CAV'07)*, pages 121–125, 2007.

[BCD+08] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-TIGA *User Manual.* Aalborg University, Aalborg, Denmark, 0.12 edition, August 2008.

[BCDL09] Peter Bulychev, Thomas Chatain, Alexandre David, and Kim Guldstrand Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *Proc. 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'09)*, pages 73–87, 2009.

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Bernardo and Corradini [BC04], pages 200–236.

[BDMP03] Patricia Bouyer, Deepak D'Souza, P. Madhusudan, and Antoine Petit. Timed control with partial observability. In *Proc. 15th International Conference on Computer Aided Verification (CAV'03)*, pages 180–192, 2003.

[BGNV05] Andreas Blass, Yuri Gurevich, Lev Nachmanson, and Margus Veanes. Play to test. In Grieskamp and Weise [GW06], pages 32–46.

[BGS05] Annette Bunker, Ganesh Gopalakrishnan, and Konrad Slind. Live sequence charts applied to hardware requirements specification and verification. *Software Tools for Technology Transfer (STTT)*, 7(4):341–350, 2005.

[BGT04] Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans. Summary - perspectives of model-based testing. In *Dagstuhl Seminar Proc. on Perspectives of Model-Based Testing*, 2004.

[BHJP04] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Grabowski and Nielsen [GN05], pages 125–139.

[BJK$^+$05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[Bon05] Yves Bontemps. *Relating Inter-Agent and Intra-Agent Specifications - The Case of Live Sequence Charts*. PhD thesis, University of Namur, Namur, Belgium, 2005.

[BS07] Yves Bontemps and Pierre-Yves Schobbens. The computational complexity of scenario-based agent verification and design. *J. Applied Logic*, 5(2):252–276, 2007.

[BSL04] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundam. Inform.*, 62(2):139–169, 2004.

[BT00] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Proc. 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP'00)*, pages 187–195, 2000.

[CDF$^+$05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proc. 16th International Conference on Concurrency Theory (CONCUR'05)*, pages 66–80, 2005.

[CDHR06] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Algorithms for omega-regular games with imperfect information. In *Proc. 20th International Workshop on Computer Science Logic (CSL'06)*, pages 287–302, 2006.

[CDL$^+$07] Franck Cassez, Alexandre David, Kim Guldstrand Larsen, Didier Lime, and Jean-François Raskin. Timed control with observation based and stuttering invariant strategies. In *Proc. 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, pages 192–206, 2007.

[CDL09] Thomas Chatain, Alexandre David, and Kim Gulstrand Larsen. Playing games with timed games. In *Proc. 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'09)*, 2009.

[Cer92] Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *Proc. Fourth International Workshop on Computer Aided Verification (CAV'92)*, pages 302–315, 1992.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[CHK08] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Software and System Modeling*, 7(2):157–175, 2008.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[CJL$^+$09] Franck Cassez, Jan Jakob Jessen, Kim Guldstrand Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Automatic synthesis of robust and optimal controllers - an industrial case study. In *Proc. 12th International Conference on Hybrid Systems: Computation and Control (HSCC'09)*, pages 90–104, 2009.

[CKL98] Richard Castanet, Ousmane Koné, and Patrice Laurençot. On the fly test generation for real time protocols. In *Proc. International Conference On Computer Communications and Networks (ICCCN'98)*, pages 378–387, 1998.

[CL95] Duncan Clarke and Insup Lee. Testing real-time constraints in a process algebraic setting. In *Proc. 17th International Conference on Software Engineering (ICSE'95)*, pages 51–60, 1995.

[CO00] Rachel Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Asp. Comput.*, 12(5):350–371, 2000.

[CSE96] John R. Callahan, Francis Schneide, and Steve M. Easterbrook. Specification-based testing using model checking. In *Proc. 2nd Workshop on the SPIN Verification System (SPIN'96)*, 1996.

[DEF$^+$96] L. Doldi, V. Encontre, J. Fernandez, T. Jéron, S.L. Bricquir, N. Texier, and M. Phalippou. *Testing of communicating systems*, chapter Assessment of automatic generation methods of conformance test suites in an industrial context, pages 347–361. Chapman & Hall, 1996.

[DGG09] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.* Addison-Wesley Professional, 2009.

[DH99] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. In *Proc. IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, 1999.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.

[DK01] Werner Damm and Jochen Klose. Verification of a radio-based signaling system using the statemate verification environment. *Formal Methods in System Design*, 19(2):121–141, 2001.

[DLL$^+$10] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: A complete specification theory for real-time systems. In *Proc. 13th International Conference on Hybrid Systems: Computation and Control (HSCC'10)*, 2010.

[DLLN08a] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of timed systems. In *Proc. 4th Workshop on Model-Based Testing (MBT'08)*, 2008. ENTCS, 220(1):79-92.

[DLLN08b] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *Proc. 11th Conference on Design, Automation and Test in Europe (DATE'08)*, pages 486–491, 2008.

[DTW06] Werner Damm, Tobe Toben, and Bernd Westphal. On the expressive power of live sequence charts. In *Program Analysis and*

Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday, pages 225–246, 2006.

[dVT00] René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *Software Tools for Technology Transfer (STTT)*, 2(4):382–393, 2000.

[DWDMR08] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robust safety of timed automata. *Formal Methods in System Design*, 33(1-3):45–84, 2008.

[EFM97] André Engels, Loe M. G. Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In *Proc. 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'97)*, pages 384–398, 1997.

[ENDKE98] Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, 1998.

[FHD$^+$99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, and Ursula Goltz. Timed sequence diagrams and tool-based analysis - a case study. In *Proc. 2nd International Conference on the Unified Modeling Language (UML'99)*, pages 645–660, 1999.

[FHT05] John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*. Springer, 2005.

[GH99] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'99)*, pages 146–162, 1999.

[GH02] Jens Grabowski and Dieter Hogrefe. Sdl- and msc-based specification and automated test case generation for inap. *Telecommunication Systems*, 20(3-4):265–290, 2002.

[GHJ97] Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In *Proc. International Workshop on Hybrid and Real-Time Systems (HART'97)*, pages 331–345, 1997.

[GMMP04] Blaise Genest, Marius Minea, Anca Muscholl, and Doron Peled. Specifying and verifying partial order properties using template mscs. In *Proc. 7th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'04)*, pages 195–210, 2004.

[GN05] Jens Grabowski and Brian Nielsen, editors. *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*. Springer, 2005.

[GRR03] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using spin to generate testsfrom asm specifications. In *Proc. 10th International Workshop on Abstract State Machines (ASM'03)*, pages 263–277, 2003.

[GW06] Wolfgang Grieskamp and Carsten Weise, editors. *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*. Springer, 2006.

[Har87] David Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HCL$^+$03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proc. 25th International Conference on Software Engineering (ICSE'03)*, pages 232–243, 2003.

[Hes07] Anders Hessel. *Model-Based Test Case Generation for Real-Time Systems*. PhD thesis, Uppsala University, Uppsala, Sweden, 2007.

[HK00] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. In *Proc. 5th International Conference on Implementation and Application of Automata (CIAA'00)*, pages 1–33, 2000.

[HK02] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. *Int. J. Found. Comput. Sci.*, 13(1):5–51, 2002.

[HKMP02] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart play-out of behavioral requirements. In *Proc. 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, pages 378–398, 2002.

[HKP04] David Harel, Hillel Kugler, and Amir Pnueli. Smart play-out extended: Time and forbidden elements. In *Proc. 4th International Conference on Quality Software (QSIC'04)*, pages 2–10, 2004.

[HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Proc. Formal Methods in Software and Systems Modeling*, pages 309–324, 2005.

[HLN$^+$03] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. In Petrenko and Ulrich [PU04], pages 114–130.

[HLS99] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In *Proc. 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, pages 277–298, 1999.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 327–341, 2002.

[HM02] David Harel and Rami Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *Proc. 10th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, pages 193–202, 2002.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[HN04] A. Hartman and K. Nagin. The agedis tools for model based testing. In Avrunin and Rothermel [AR04], pages 129–132.

[HNTC99] Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating test cases for a timed i/o automaton model. In *Proc. IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS'99)*, pages 197–214, 1999.

[HP06] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: An industrial case-study. In *Proc. 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'06) and 5th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'06)*, pages 116–131, 2006.

[HS06] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *Proc. 14th International Symposium on Formal Methods (FM'06)*, pages 1–15, 2006.

[HT03] David Harel and P.S. Thiagarajan. *UML for real: design of embedded real-time systems*, chapter Message Sequence Charts, pages 77–105. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[IT96] ITU-T. Message sequence charts (msc), itu-t recommendation z.120, 1996.

[IT99] ITU-T. Message sequence charts – msc-2000, itu-t recommendation z.120, 1999.

[JJ05] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms. *Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[JRLD07] Jan Jakob Jessen, Jacob Illum Rasmussen, Kim Guldstrand Larsen, and Alexandre David. Guided controller synthesis for climate controller using uppaal tiga. In *Proc. 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'07)*, pages 227–240, 2007.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[Kho02] Ahmed Khoumsi. A method for testing the conformance of real time systems. In *Proc. 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'02)*, pages 331–354, 2002.

[KHP$^+$05] Hillel Kugler, David Harel, Amir Pnueli, Yuan Lu, and Yves Bontemps. Temporal logic for scenario-based specifications. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 445–460, 2005.

[KJM03] Ahmed Khoumsi, Thierry Jéron, and Hervé Marchand. Test cases generation for nondeterministic real-time systems. In Petrenko and Ulrich [PU04], pages 131–146.

[Klo03] Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior.* PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003.

[KPP09] Hillel Kugler, Cory Plock, and Amir Pnueli. Controller synthesis from lsc requirements. In *Proc. 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, pages 79–93, 2009.

[KS09] Hillel Kugler and Itai Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Proc. 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, pages 77–91, 2009.

[KSH07] Hillel Kugler, Michael J. Stern, and E. Jane Albert Hubbard. Testing scenario-based models. In *Proc. 10th International Conference on Fundamental Approaches to Software Engineering (FASE'07)*, pages 306–320, 2007.

[KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Proc. 11th International SPIN Workshop on Model Checking Software (SPIN'04)*, pages 109–126, 2004.

[KT06] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation. In *Proc. 3rd International Colloquium on Theoretical Aspects of Computing (ICTAC'06)*, pages 317–331, 2006.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[KTWW06] Jochen Klose, Tobe Toben, Bernd Westphal, and Hartmut Wittke. Check it out: On the efficient formal verification of live sequence charts. In *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, pages 219–233, 2006.

[KV97] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *Proc. 2nd International Conference on Temporal Logic (ICTL'97)*, pages 91–106, Manchester, July 1997.

[KW01] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 512–527, 2001.

[Lah08] Jussi Lahtinen. Model checking timed safety instrumented systems. Master's thesis, Helsinki University of Technology, Espoo, Finland, June 2008. Research Report TKK-ICS-R3.

[Lam05a] Leslie Lamport. Real-time is really simple. TechReport MSR-TR-2005-30, Microsoft Research, March 2005.

[Lam05b] Leslie Lamport. Real-time model checking is really simple. In *Proc. 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, pages 162–175, 2005.

[Lio96] Jacques-Louis Lions. Ariane 5 flight 501 failure: Report of the inquiry board. Paris, 1996.

[LK01] Marc Lettrari and Jochen Klose. Scenario-based monitoring and testing of real-time uml models. In *Proc. 4th International Conference on the Unified Modeling Language (UML'01)*, pages 317–328, 2001.

[LMM02] Martin Leucker, P. Madhusudan, and Supratik Mukhopadhyay. Dynamic message sequence charts. In *Proc. 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'02)*, pages 253–264, 2002.

[LMN04] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In Grabowski and Nielsen [GN05], pages 79–94.

[LMNS05] Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *Proc. 5th ACM International Conference On Embedded Software (EMSOFT'05)*, pages 299–306, 2005.

[LPWY99] Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nord. J. Comput.*, 6(3):271–298, 1999.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[LRRA98] Peter Liggesmeyer, Martin Rothfelder, Michael Rettelbach, and Thomas Ackermann. Qualitätssicherung software-basierter technischer systeme - problembereiche und lösungsansätze. *Informatik Spektrum*, 21(5):249–258, 1998.

[LS98] Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract). In *Proc. 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, pages 53–66, 1998.

[LY97] Kim Guldstrand Larsen and Wang Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Inf. Comput.*, 134(2):75–101, 1997.

[MLN04] Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Proc. 19nd IEEE/ACM International Conference on Automated Software Engineering (ASE'04)*, pages 396–397, 2004.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, pages 229–242, 1995.

[NAS99] NASA. Report of the mars climate orbiter mishap, 1999.

[NAS00] NASA. The jpl special review board report of the loss of the mars polar lander and deep space 2 missions, 2000.

[Ng93] Meng-Siew Ng. Reasoning with timing constraints in message sequence charts. Master thesis, University of Stirling, Scotland, U.K., August 1993.

[NR05] Manuel Núñez and Ismael Rodríguez. Conformance testing relations for timed systems. In Grieskamp and Weise [GW06], pages 103–117.

[NS01a] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 343–357, 2001.

[NS01b] Brian Nielsen and Arne Skou. Test generation for time critical systems: Tool and case study. In *Proc. 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, pages 155–162, 2001.

[NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *Software Tools for Technology Transfer (STTT)*, 5(1):59–77, 2003.

[NVS+04] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. In Avrunin and Rothermel [AR04], pages 55–64.

[Org05] Object Management Organization. Uml 2.0 superstructure specification. `http://www.omg.org/spec/UML/2.0`, 2005.

[Per85] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Comput. Commun. Rev.*, 15(4):44–53, 1985.

[Pra95] K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995.

[PU04] Alexandre Petrenko and Andreas Ulrich, editors. *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*. Springer, 2004.

[Pus10] Saulius Pusinskas. *Capturing and Testing Behavioral Requirements by Means of Live Sequence Charts*. PhD thesis, Aalborg University, Aalborg, Denmark, 2010.

[RAJGJ04] J.G Rye-Andersen, M.W. Jensen, R. Goettler, and M. Jakobsen. Peel: Property extraction engine for lscs. Master's thesis, Aalborg University, Aalborg, Denmark, 2004.

[RH01] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th IEEE International Conference on Engineering of Computer-Based Systems (ECBS'01)*, pages 83–, 2001.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[SC02] Bikram Sengupta and Rance Cleaveland. Triggered message sequence charts. In *Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT FSE'02)*, pages 167–176, 2002.

[SD05a] Jun Sun and Jin Song Dong. Model checking live sequence charts. In *Proc. 10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 529–538, 2005.

[SD05b] Jun Sun and Jin Song Dong. Synthesis of distributed processes from scenario-based specifications. In Fitzgerald et al. [FHT05], pages 415–431.

[ST08] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *Proc. 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08)*, pages 250–264, 2008.

[STMW04] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The rhapsody uml verification environment. In *Proc. 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 174–183, 2004.

[SVD01] Jan Springintveld, Frits W. Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theor. Comput. Sci.*, 254(1-2):225–257, 2001.

[TA99] Stavros Tripakis and Karine Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *Proc. World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, pages 233–252, 1999.

[TB03] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. In *Proc. 1st European Conference on Model-Driven Software Engineering (ECMDSE'03)*, pages 31–43, 2003.

[Tre96a] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *Proc. 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, pages 127–146, 1996.

[Tre96b] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] Jan Tretmans. Testing concurrent systems: A formal approach. In *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, pages 46–65, 1999.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[VCST05] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'05)*, pages 273–282, 2005.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *Proc. 22nd International Conference on on Software Engineering (ICSE'00), Future of Software Engineering Track*, pages 147–159, 2000.

[VRC06] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In *Proc. 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV'06)*, pages 240–253, 2006.

[VRKE07] Jüri Vain, Kullo Raiend, Andres Kull, and Juhan P. Ernits. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 363–372, 2007.

[WQSD07] Hai H. Wang, Shengchao Qin, Jun Sun, and Jin Song Dong. Realizing live sequence charts in systemverilog. In *Proc. First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 379–388, 2007.

[WRYC04] Tao Wang, Abhik Roychoudhury, Roland H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *Proc. 6th International on Practical Aspects of Declarative Languages (PADL'04)*, pages 178–192, 2004.

[Yan04] Mihalis Yannakakis. Testing, optimizaton, and games. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, pages 28–45, 2004.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer (STTT)*, 1(1-2):123–133, 1997.