

Technical Report

August 15, 2006



Check It Out: On the Efficient Formal Verification of Live Sequence Charts^{*}

Jochen Klose³, Tobe Toben¹, Bernd Westphal¹, and Hartmut Wittke²

¹ Fak. II, Dept. für Informatik, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany, {toben,westphal}@informatik.uni-oldenburg.de

² OSC – ES AG, Industriestr. 11, 26121 Oldenburg, Germany, wittke@osc-es.de

³ Bombardier Transportation, Wolfenbüttler Str. 86/Obergstr. 5, 38102 Braunschweig, Germany, jochen.klose@de.transport.bombardier.com

Abstract. Live Sequence Charts (LSCs) are an established visual formalism for requirements in formal, model-based development, in particular aiming at formal verification of the model. The model-checking problem for LSCs is principally long solved as each LSC has an equivalent LTL formula, but even for moderate sized LSCs the formulae grow prohibitively large. In this paper we elaborate on practically relevant sub-classes of LSCs, namely bonded and time bounded, which don't require the full power of LTL model-checking. For bonded LSCs, a combination of observer automaton and fixed small liveness property and for additionally time bounded LSCs reachability checking is sufficient.

1 Introduction

Scenario-based approaches in general and Live Sequence Charts (LSCs) [1] in particular have shown adequate for the formal specification of inter-object requirements on distributed systems in formal, model-based development [2–8]. That is, requirements on a system under design are formally specified by LSCs before a model of the system is built. Model-checking can then automatically check whether the model satisfies the requirements to find errors early, before the actual implementation.

The model-checking problem of LSCs vs. Kripke structures is principally solved since universal LSCs translate to equivalent LTL formulae [9] and existential LSCs translate to observer automata, i.e. 1-acceptance [10], or CTL^{*} thus corresponding model-checkers can directly be employed. Accordingly there are proof-of-concept results for the formal verification of LSCs against StateMate [11, 8] and UML [12, 13] models. In particular with the industrial case study considered in [11] it turned out that the LTL formulae grow large even for LSCs of moderate size. That is, formal verification becomes expensive due to the size of the *requirements* (for details cf. [9, 14]). Our subject is *efficient* formal verification of LSCs. We identify two sub-classes of LSCs for which techniques that are faster but less powerful than LTL model-checking are sufficient, or help in finding errors fast.

^{*} Partly supported by DFG, grants SFB/TR 14 AVACS and DA 206/7-3, SPP 1064.

	mandatory/hot/universal	possible/cold/existential
chart	...each activating system run suffix adheres to scenario	... there is an activating and adhering system run suffix
location	progress enforced	progress not enforced
condition/ local invar.	system violates LSC if condition doesn't hold	system satisfies LSC if condition doesn't hold
message	reception has to be observed	reception needn't be observed

Table 1. Modalities of LSC elements and their semantics.

Related Work. Model-checking LSCs against system models has been first investigated by [6]. They manually derive a selection of small, local LTL properties from an LSC and check whether they hold for a model of a bus protocol. The limited size of the checked properties didn't raise the need to consider more efficient procedures. In [15, 16], model-checking is used as a technique to obtain satisfying paths for a set of LSCs in the context of playing-out LSCs [17]. Their representation of LSCs employs one automaton per instance-line. Similarly, [18] check a set of LSCs for consistency using a CSP semantics of LSCs, namely one CSP process per instance line, and the FDR model-checker. Both are particularly tailored for checking LSCs against each other and don't discuss the relation to general system model. Furthermore, both discuss only a limited subset of the dialect [11], in particular excluding time. We use the term "LSC verification" similar to, for instance, "LTL verification" which means checking a formula against a model. The observer based approach for LSC model-checking has been introduced in [19] and further studied in the context of Symbolic Timing Diagrams (STDs) in [20]. Our results slightly extend [20] since we have to discuss the case of non-deterministic automata which are needed for non-bonded LSCs while deterministic automata are sufficient for the scope of [20]. An observer approach has also been applied to testing in the domain of UML [21] and SystemC [22].

The remainder of the paper is structured as follows. In Sect. 2 we briefly recall LSCs. Section 3 introduces Timed Symbolic Automata (TSA), the semantical foundation of LSCs, together with the notions of determinism and time boundedness. It provides the basic strategy for efficient verification of TSAs. The application to LSC model-checking is discussed in Sect. 4. Section 5 supplies experimental results and Sect. 6 concludes.

2 Live Sequence Charts

The visual formalism LSCs has been introduced in [1] to overcome several shortcomings of the well-known Message Sequence Charts (MSC) wrt. a formal usage. It is a conservative extension of basic MSCs that gains increased expressive power by adding *modalities* to charts, locations, and elements (cf. Table 1).

The mode of a chart can be either *existential* or *universal*. An existential LSC is satisfied by a system if there is at least one system run adhering to the LSC. Conversely, a universal LSC is satisfied if all runs of the system adhere to it. A location's mode, either *hot* or *cold*, expresses liveness requirements. An

element following a hot location has to be observed *finally* in order to satisfy the LSC. A cold location doesn't enforce progress. Conditions are, in contrast to MSCs, semantically relevant in LSCs and have a mode. If a *mandatory* (or hot) condition isn't satisfied when supposed to according to the scenario, the chart is violated. If a *possible* (or cold) condition isn't satisfied, the whole chart is immediately considered satisfied. It is *legally exited*. This interpretation applies alike to *local invariants*. They have been introduced in [11] to state requirements on spans of time instead of only single points in time as with conditions. The mode of a message, either mandatory (hot) or possible (cold), denotes whether the message may get lost. The reception of a hot message has to be observed to satisfy the LSC, for a cold message it needn't be observed.

In addition to modes, LSCs add to MSCs means that characterise the situations to which the scenario applies, i.e. its *activation*. Activation in general is characterised by a prefix of the LSC called *pre-chart* meaning whenever the pre-chart is observed then the system shall adhere to the rest of the LSC, the *main-chart*. The *activation condition* is a shortcut for pre-charts with a single condition only. For formal verification, [11] adds the activation mode – one of *initial*, *invariant*, or *iterative* – to further restrict activation. Initial LSCs may only be activated in initial system states. Iterative LSCs disregard violations of reactivating LSCs, i.e. LSCs that comprise a sequence that adheres to the LSC's own pre-chart again. Furthermore, both [11, 17] have added the notion of *strict* vs. *tolerant* (or *weak*) interpretation. The strict interpretation requires that messages used in the chart don't occur at other points in time than the ones given by the LSC. The tolerant interpretation ignores additional messages. For example, a system sending *red_on* once again before expiration of the timer would not satisfy the LSC from Fig. 1 strictly.

Fig. 1(a) is a simplified requirement on a level crossing controller. It is activated when the crossing controller receives an asynchronous message '*secreq*' from the environment. The crossing controller shall finally, as indicated by the solid segment of its instance line, start the lights and barrier controllers by synchronous messages '*lights_on*' and '*barrier_down*'. The timing interval [5, 15] requires lowering the barrier to take between 5 and 15 units of time and the hot local invariant $\neg MvUp$ requires the barrier not to move up from '*barrier_down*' reception up to and including the point in time where '*barrier_ok*' is sent. If the traffic lights controller is not operational when receiving '*lights_on*', the LSC is legally exited at the cold condition '*Operational*'. Otherwise timer *t* is started. Timeout of *t* shall occur when sending '*lights_ok*', i.e. switching on the lights shall take exactly 7 units of time. The order of '*lights_ok*' and '*barrier_ok*' is explicitly relaxed by enclosing them in a *coregion* as indicated by the dotted line. The messages may occur in any order, even simultaneously. When both have been received, the crossing controller *may* send '*done*' back to the environment as no hot location *enforces* progress at this position.

We postpone recalling the formal semantics of LSC following [11] to Sect. 4, thus after the introduction of Timed Symbolic Automata in Sect. 3. Note that we discuss the LSC dialect of [11] which is tailored for the application domain of

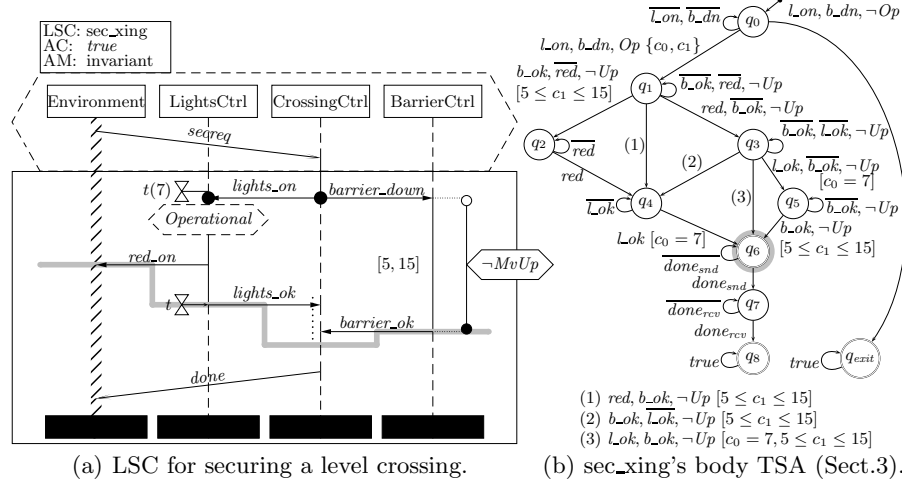


Fig. 1. For brevity, overlining denotes negation and comma denotes conjunction in Fig. 1(b). E.g. q_0 's loop fires if neither 'lights_on' nor 'barrier_down' are observed.

formal verification in contrast to the play-engine dialect of [17]. This is not an exclusive choice as both share a large common sublanguage. LSC specifications may well be played in following [17] and then strengthened for formal verification using the features from [11]. Furthermore we assume *well-formed* LSCs [23], i.e. LSCs without internal contradictions.

3 Efficient TSA Model-Checking

3.1 Preliminaries

We use $Expr_{\mathcal{S}}$ to denote the propositional logic formulae over signature \mathcal{S} , and $\mathcal{I} \models \psi$ to denote that interpretation \mathcal{I} satisfies $\psi \in Expr_{\mathcal{S}}$ with fixed non-empty universe \mathcal{U} . Let C be a set of clocks. A *clock valuation* is a mapping $\tau : C \rightarrow \mathbb{N}_0$, and $T(C)$ denotes the set of all clock valuations. The update of a time valuation $\tau \in T(C)$ by a value $x \in \mathbb{N}_0$, written $\tau + x$, is pointwise defined as $(\tau + x)(c) := \tau(c) + x$ for all $c \in C$. The set of *clock constraints* $\Phi(C)$ is defined by the grammar $\phi ::= true \mid c \leq x \mid c \geq x, \varphi ::= \phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2, c \in C, x \in \mathbb{N}_0$. We write $\tau \models \varphi$ to denote that the clock valuation $\tau \in T(C)$ satisfies the clock constraint $\varphi \in \Phi(C)$. The definition of satisfaction is standard.

3.2 Timed Symbolic Automata

Timed Symbolic Automata (TSA) are a variant of timed Büchi automata where transitions are labelled by expressions from $Expr_{\mathcal{S}}$ instead of just an element of an alphabet (cf. [20] for references). The new notion of *default transitions* significantly eases the formal definition of the clock propagation introduced below.

Formally, a TSA over a signature \mathcal{S} is a tuple $\mathcal{A} = (Q, q_s, C, \rightsquigarrow, D, F)$ with a finite set of states Q , initial state $q_s \in Q$, a finite set of clocks C , transition relation $\rightsquigarrow \subseteq Q \times \text{Exprs}_{\mathcal{S}} \times \Phi(C) \times 2^C \times Q$, default transitions $D \subseteq Q \times Q$, and accepting states $F \subseteq Q$. We define $\rightarrow \subseteq Q \times Q$ as $\rightarrow := D \cup \{(q, q') \mid \exists (q, \psi, \varphi, \rho, q') \in \rightsquigarrow\}$. A TSA is called *Partially Ordered TSA* (POTSA) if the reflexive transitive closure of \rightarrow is a partial order, i.e. \rightarrow^* is anti-symmetric. Note that all loops in a POTSA are consequently self-loops.

Let \mathcal{S} be a signature and \mathcal{U} a fixed universe. A *timed interpretation sequence* is a sequence $r = ((\iota_i, t_i))_{i \in \mathbb{N}_0}$ with ι_i an interpretation of \mathcal{S} and $t_i \in \mathbb{N}_0$ a timestamp such that $t_i < t_{i+1}$, $i \in \mathbb{N}_0$. Let $((q_i, \tau_i))_{i \in \mathbb{N}_0}$ be a sequence with $q_i \in Q$ a state and $\tau_i \in T(C)$ a valuation of the clocks, $i \in \mathbb{N}_0$. It is called *timed run* of \mathcal{A} over r iff it starts in the initial state, i.e. $q_0 = q_s$, the clocks initially have value zero, i.e. $\tau_0(c) = 0$, $c \in C$, and states are \mathcal{A} -successors. That is, for $i \in \mathbb{N}_0$ either there is a transition $(q_i, \psi_i, \varphi_i, \rho_i, q_{i+1}) \in \rightsquigarrow$ such that the boolean and clock constraints hold, $\iota_i \models \psi_i$ and $(\tau_i + (t_{i+1} - t_i)) \models \varphi_i$, and the clock valuations are updated according to ρ_i , i.e. $\tau_{i+1}|_{\rho_i} = 0$ and $\tau_{i+1}|_{C \setminus \rho_i} = (\tau_i + (t_{i+1} - t_i))|_{C \setminus \rho_i}$, or there is a default transition $(q_i, q_{i+1}) \in D$ and $\tau_{i+1} = \tau_i + (t_{i+1} - t_i)$. A timed run $((q_i, \tau_i))_{i \in \mathbb{N}_0}$ is called *accepting* if $q_i \in F$ for infinitely many $i \in \mathbb{N}_0$. The *language accepted* by \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$, is the set of timed interpretation sequences for which an accepting run exists.

In the following we introduce two subclasses of TSAs, namely deterministic and time-bounded ones. In Sect. 3.4 we will see how membership in these classes determines the efficiency of the model-checking procedure.

We call a state $q \in Q$ *deterministic* if the constraints on all outgoing transitions are mutually disjoint, i.e. for each two transitions $(q, \psi_1, \varphi_1, \rho_1, q_1)$, $(q, \psi_2, \varphi_2, \rho_2, q_2) \in \rightsquigarrow$ with $q_1 \neq q_2$ we have $(\iota \models \psi_1 \wedge \tau \models \varphi_1) \rightarrow \neg(\iota \models \psi_2 \wedge \tau \models \varphi_2)$ for any ι and τ . It is called *reaching-deterministic* if all $q' \in Q$ with $q' \rightarrow^* q$ are deterministic. We call \mathcal{A} deterministic iff all its states are deterministic.

Given a set of clocks C , the set of upper bounded clock constraints $\Phi_{\leq}(C) \subseteq \Phi(C)$ is defined by the grammar $\phi ::= x_1 \leq c \wedge c \leq x_2$, $\varphi ::= \phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$, $c \in C$, $x_1, x_2 \in \mathbb{N}_0$. We call a state $q \in Q$ *time bounded* iff the clock constraints on all outgoing transitions are from $\Phi_{\leq}(C)$. We call \mathcal{A} time bounded iff all states from $Q \setminus F$ are time bounded. Now let \mathcal{A} be a POTSA and q a state of it s.t. all outgoing transitions impose a finite upper bound on clock c . Let q' be a state from which an accepting state is only reached by visiting q and let c not be reset along the path from q' to q . Then the boundedness of c transitively induces bounds on all transitions between q' and q , including self-loops [20]. For example, clock constraint $c_0 = 7$ at the transition from state q_4 to q_6 in Fig. 1(b) propagates to the transition from q_2 to q_4 and to the self-loops at q_4 and q_2 . Consequently, the number of time bounded states can be increased by propagating clock constraints backwards through the automaton, i.e. making the implicit constraints explicit. In [20] we give an algorithm which performs *clock propagation* on POTSA and yields a language-equivalent POTSA.

Another operation of interest is the *failure state completion* [20] of \mathcal{A} . For each reaching-deterministic state in Q , it adds a default transition to a designated fail-

ure state ‘ q_{fail} ’, i.e. it yields the language equivalent [20] TSA $(Q \dot{\cup} \{q_{fail}\}, q_s, C, \rightsquigarrow, D', F)$ with $D' := D \cup \{(q, q_{fail}) \mid q \text{ is reaching-deterministic}\}$. Then reaching ‘ q_{fail} ’ with sequence r is a sufficient criterion for $r \notin \mathcal{L}(\mathcal{A})$.

3.3 Timed Symbolic Automata as Specification

A *Kripke Structure* is a tuple $K = (AP, S, S_0, R, L)$ with atomic propositions AP , states S , initial states $S_0 \subseteq S$, transition relation $R \subseteq S \times S$, and labelling function $L : S \rightarrow 2^{AP}$, AP and S finite. As TSAs are defined using interpretation sequences, we assume that each subset $\ell \subseteq AP$ defines an interpretation ι_ℓ of S . The TSA and the model are then called *compatible*. A run s_0, s_1, \dots of K induces the timed interpretation sequence $((\iota_i, t_i))_{i \in \mathbb{N}_0}$ with $\iota_i = L(s_i)$, $t_i = i$. The set of all timed interpretation sequences induced by the runs of K is denoted by $R(K)$.

Using a TSA \mathcal{A} as a specification on K means relating $R(K)$ to $\mathcal{L}(\mathcal{A})$. To increase precision of the specification, here we always consider an *activation condition* $\nu \in Expr_S$ and an activation mode of *initial*, *invariant*, and *iterative* together with \mathcal{A} . We say K satisfies \mathcal{A} initially wrt. ν , denoted by $K \models_{\nu, init} \mathcal{A}$, iff all $((\iota_i, t_i))_{i \in \mathbb{N}_0} \in R(K)$ with $\iota_0 \models \nu$ are in $\mathcal{L}(\mathcal{A})$. It satisfies \mathcal{A} invariantly wrt. ν , denoted by $K \models_{\nu, inv} \mathcal{A}$, iff $\iota_i \models \nu$ implies that the suffix $(\vec{t}_{i+1}, \vec{t}_{i+1})(\vec{t}_{i+n}, \vec{t}_{i+n}) \dots$ is in $\mathcal{L}(\mathcal{A})$. Iterative satisfaction, denoted by $K \models_{\nu, iter} \mathcal{A}$, is special to POTSA. It is similar to invariant but excludes overlapping activations of \mathcal{A} . The motivation to introduce this mode was to ease the understanding of counter-examples. It is easier to uniquely identify where activation takes place if there are no overlapping activations. But this mode has the serious drawback that if it is used for a TSA that actually *has* an overlapping activation, a violation may be shadowed. Checking whether a TSA doesn’t have an overlapping activation (it is then called *non-reactivating*) or not is an additional non-trivial task.

Note that by the definition above we choose model steps as time units. In general, other notions of time have to be supported, for instance, the supersteps of Statemate models. In [20], the approach presented here has been extended to support two notions of time by assuming that the passing of time is observable on the model, for instance by a special signal of the model which is raised whenever a superstep is completed. To keep the discussion focused and to adhere to space restrictions, we only consider step time. Adding the more general treatment of time following [20] is actually straightforward.

3.4 Efficient POTSA Model-Checking

Each non-iterative POTSA with step clocks has an equivalent LTL formula [24] in negative normal form, thus LTL model-checking can be applied to decide whether K satisfies \mathcal{A} . For all POTSA, there is an additional approach based on composing in parallel to the model a number of time counters, one for each clock, and a kind of transition system view of the TSA, i.e. dismissing the Büchi criterion. This parallel composition is then checked for reachability of certain states or for a small fixed liveness formula. In this section we introduce the

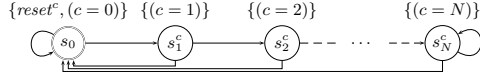


Fig. 2. Kripke structure of timer c with upper bound $N + 1$.

parallel composition of K with observers. Section 3.4.3 discusses when to apply which approach.

3.4.1 Timer Extension. In model-checking timed TSA properties, we can't directly employ the sequences from $R(K)$ as the time stamps are unbounded. By [20] it is sufficient for the observer approach to consider finite time counters for each clock of the (not necessarily time bounded) TSA and these bounds are effectively computable. These time counters are transition systems that count model steps until their finite upper bound is reached and that may reset themselves any time and then set a reset flag (cf. Fig. 2). They are composed in parallel to the model. More formally, let c be a clock with upper bound $N + 1$, and $S_c = \{s_0^c, \dots, s_N^c\}$ a set of fresh states wrt. S . The *timer extension* of K for c is $K^c = (AP^c, S^c, S_0^c, R^c, L^c)$ with

- $AP^c = AP \dot{\cup} \{\text{reset}^c, (c=0), \dots, (c=N)\}$, $S^c = S \times S_c$, $S_0^c = S_0 \times \{s_0^c\}$,
- $R^c = \{((s, s_i^c), (s', s_0^c)), ((s, s_N^c), (s', s_N^c)) \mid (s, s') \in R\}$
 $\cup \{((s, s_i^c), (s', s_i^c + 1)) \mid (s, s') \in R, 0 \leq i < N\}$
- $L^c((s, s_i^c)) = L(s) \cup \{\text{reset}^c \mid i = 0\} \cup \{(c = i)\}$.

The timer extension K^c is by the same procedure extended to $K^{c,c'}$ for a second clock. We use $K^C := K^{c_1, \dots, c_n}$ to denote the timer extension of K for all clocks from C . A state $s \in K^C$ canonically defines a clock valuation τ_s as $\tau_s(c) := i$ if $(c = i) \in L(s)$. The clocks reset in s are $\rho_s := \{c \mid \text{reset}^c \in L(s)\}$.

3.4.2 Observer Extension. Let $\mathcal{A} = (Q, q_s, C, \rightsquigarrow, D, F)$ be a TSA compatible with K and K^C the timer extension of K . Using s_C to denote the components of states that are introduced by the timer extension, the observer extension of K for activation expression ν and initial activation \mathcal{A} is $K^{\mathcal{A}}/init = (AP^{\mathcal{A}}, S^{\mathcal{A}}, S_0^{\mathcal{A}}, R^{\mathcal{A}}, L^{\mathcal{A}})$ with

- $AP^{\mathcal{A}} = (AP^C \dot{\cup} \{\text{fair}, \text{fail}\})$, $S^{\mathcal{A}} = S^C \times (Q \dot{\cup} \{q_{idle}\})$,
- $S_0^{\mathcal{A}} = \{(s, s_C, q_s) \mid (s, s_C) \in S_0^C, \iota_{L(s)} \models \nu\}$
 $\cup \{(s, s_C, q_{idle}) \mid (s, s_C) \in S_0^C, \iota_{L(s)} \not\models \nu\}$,
- $((s, s_C, q), (s', s'_C, q')) \in R^{\mathcal{A}}$ iff $((s, s_C), (s', s'_C)) \in R^C$ and $q = q' = q_{idle}$ or
 - either there is a regular transition $(q, \psi, \varphi, \rho, q') \in \rightsquigarrow$ with $\iota_{L(s)} \models \psi$, $\tau(s', s'_C) \models \varphi$, and $\rho_s = \rho$
 - or there is a default transition $(q, q') \in D$ and $\rho_s = \emptyset$,
- $L^{\mathcal{A}}((s, s_C, q)) = L^C((s, s_C)) \cup \{\text{fair} \mid q \in F\} \cup \{\text{fail} \mid q = q_{fail}\}$.

For invariant activation, the observer K^A/inv is activated non-deterministically thus additionally all (s, s_C, q_{idle}) are in S_0^A independent from satisfaction of ν . Furthermore there is a transition from $q = q_{idle}$ to $q' = q_s$ whenever $\iota_{L(s)} \models \nu$. For iterative activation, the initial states of the observer extension are as in $K^A/init$. $K^A/iter$ remains in q_{idle} only if ν is not satisfied and takes the transition to q_s whenever possible. In addition there are transitions back to q_{idle} and q_s from each accepting state with only a self-loop. The transition is to q_s if ν is satisfied and to q_{idle} otherwise. Thus $K^A/iter$ is slightly smaller than K^A/inv . Note that K^A/m , m a mode, restricts the behaviour of the clocks, but not the behaviour of the model. In K^A/m , clocks are only reset if there is a transition in the TSA with a corresponding reset set. K^A/m sets the flag ‘fail’ iff the failure state of the TSA is reached and ‘fair’ iff the TSA is in an accepting state. We call a state $s^A \in S^A$ a *failure state* iff $fail \in L^A(s^A)$ and *fair state* iff $fair \in L^A(s^A)$.

3.4.3 Putting It All Together. Now we can devise a strategy for deciding whether a given Kripke structure satisfies a failure state completed POTSA using four different standard model-checking procedures of different worst-case complexity, namely reachability checking with safety observer, ACTL model-checking with and without observer, and LTL model-checking. The following Lemma states that the less powerful techniques are helpful for finding violations early and that they are sufficient for deterministic (time bounded) POTSA.

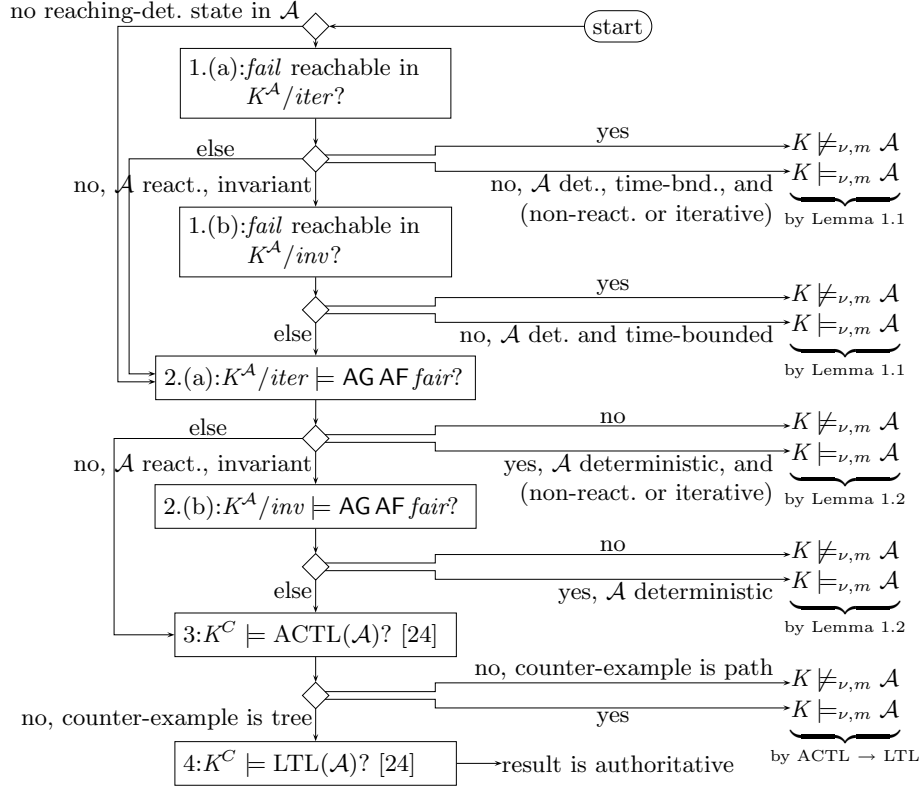
Lemma 1. *Let \mathcal{A} be a POTSA, K a Kripke structure, ν a condition, m a mode.*

1. [20] *If a failure state of K^A/m is reachable, then $K \not\models_{\nu, m} \mathcal{A}$. If \mathcal{A} is deterministic and time bounded then $K \not\models_{\nu, m} \mathcal{A}$ implies reachability of a failure state.*
2. [20] *If $K^A/m \models \text{AG AF fair}$, then $K \models_{\nu, m} \mathcal{A}$. If \mathcal{A} is deterministic then $K \models_{\nu, m} \mathcal{A}$ implies $K^A/m \models \text{AG AF fair}$.*
3. *If \mathcal{A} is non-deterministic, then there is no LTL formula λ using only atomic proposition ‘fair’ s.t. $K^A/m \not\models_{\nu, m} \lambda$ implies $K \not\models_{\nu, m} \mathcal{A}$. \diamond*

Proof. (1.3) Assuming such a formula λ , exploit non-determinism in observer to construct a Kripke structure K s.t. K^A/m doesn’t satisfy λ but $K \models_{\nu, m} \mathcal{A}$. \square

Note that Lemma 1.3 implies that there is no known procedure to decide satisfaction of iterative non-deterministic TSAs because within an LTL formula we cannot, as with observer extensions, refer to the own state of activation.

Using that the ACTL formula obtained from an LTL formula in negative normal form by prefixing all modal operators by ‘A’ implies the LTL formula, we devise the strategy depicted in the following control flow diagram for the verification of POTSA specifications. The idea is to apply the procedure with the best worst-case complexity first to find contradictions early. Only if no errors are unveiled *and* the procedure is too weak for the POTSA, the next expensive procedure is applied. The increasing prevalence of multi-processor or multi-core hosts allows to apply the procedures in parallel and stop all other procedures once a significant result is obtained.



The refinement of steps 1 and 2 into (a) and (b) is a minor optimisation using the expectation that checking \mathcal{A} iteratively is less expensive (cf. 3.4.2). An initial TSA is treated like an interactive one, as the time of activation is then uniquely determined to be an initial state.

Note that the time and space resource consumption of certain observer based tasks can be reduced by applying a POTSA version of the decomposition algorithm presented in [14] and conducting the decomposed, smaller tasks in parallel.

4 Efficient LSC Model-Checking

The following transfer of the results from the previous section to the domain of LSCs is effective as most practically used LSCs yield deterministic POTSA. Furthermore, unbounded liveness requirements typically only occur in early and rather abstract parts of a system's LSC specification and are restricted by bounds in later, more detailed versions thus most LSCs yield time-bounded TSAs.

Section 4.1 briefly recalls the LSC semantics in terms of TSA and relates the TSA classes identified in Sect. 3 to LSCs. For a complete definition of the LSC language, its abstract syntax, and the semantics-giving unwinding procedure the reader is referred to [11]. The discussion of LSC verification starts in Sect. 4.2 with the simplest case, namely invariant universal LSCs without pre-chart or

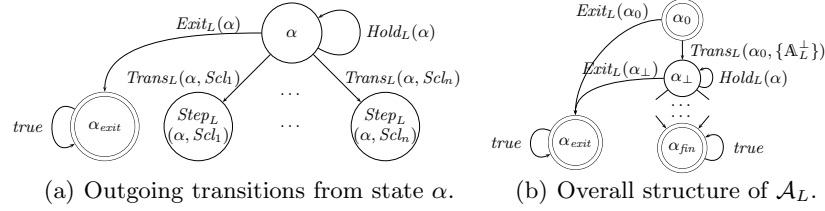


Fig. 3. Structure of the LSC body automaton. Double lined states are in F .

assumptions. The additional features of LSCs are then discussed in isolation in the subsequent sections.

4.1 LSC Semantics and TSA Properties

The central concept of the LSC semantics [11] is the notion of the *cut*, i.e. a set of at least one location per instance line (more than one for coregions). The gray line in Fig. 1(a) indicates a cut. Each TSA state corresponds to one cut of the LSC, e.g. the example cut in Fig. 1(a) corresponds to state q_6 in Fig. 1(b). Intuitively, a system satisfies the LSC if for each system run suffix with a prefix in the language of the pre-chart TSA, the rest of the run is in the language of the main-chart TSA. The algorithm of [11] translates the pre- and main-chart of each commitment and assumption LSC of a requirement into separate TSAs. Fig. 3(a) schematically shows a state of the TSA obtained by the unwinding procedure [11]. It has a self-loop awaiting the subsequent LSC elements and one transition for each combination of occurrence and non-occurrence of awaited elements. Violations of cold conditions lead to the accepting state α_{exit} . Fig. 3(b) shows the overall structure of the TSA. The state α_\perp corresponds to the cut with all instance heads and α_{fin} to complete traversal. Fig. 1(b) gives the TSA of the main-chart from Fig. 1(a). (The initial TSA state α_0 is omitted for brevity.)

Following [9], each TSA of an LSC is a POTSA. The timer propagation algorithm of [20] applies directly. In [9], we have introduced a sufficient criterion on LSCs that implies determinism of the corresponding TSAs. The TSAs are deterministic if all conditions and local invariants occur *bonded* in the LSCs. That is, if they are in a simultaneous region with at least one message, timeout, or instance head. Then the evaluation time for the condition is well-defined. This criterion is easily [23] checkable on the abstract syntax of the LSC. The observer automaton construction of [20] as introduced in Sect. 3 applies directly as LSCs share activation modes initial, invariant, and iterative with TSAs.

4.2 LSC Model-Checking

Note that an LSC's strict or tolerant interpretation is mostly orthogonal to the issues discussed below. In the TSA, strictness is expressed by strengthening the transition annotations. Each expression is additionally conjoined with a term that excludes all messages that are not referred to by the expression. Thus

the strict TSA doesn't introduce new non-determinism and as it only considers messages, non-determinism is not resolved except for few pathological cases. Consequently we needn't treat the interpretation explicitly below.

4.2.1 Universal, no Pre-chart or Assumption. In the easiest case, we only have to consider the TSA \mathcal{A} of the main-chart and an activation condition. Then the algorithm from Sect. 3.4.3 applies directly.

4.2.2 Pre-Chart. For the pre-chart of an LSC, a separate TSA is constructed. The slightly different algorithm adjusts transitions to the special interpretation of pre-charts which don't have a notion of violation. Using the ACTL/LTL way there are two options for checking an LSC with pre-chart. Let φ_{pc} and φ_L be the LTL formulae corresponding to the pre-chart and the *whole* LSC, i.e. pre- and main-chart together. Following [25], we can check $\mathbf{G}(\varphi_{pc} \rightarrow \varphi_L)$. This formula tends to grow large since the pre-chart part occurs twice, even if the more compact formulae also presented in [25] is usable.

As the semantics of the pre-chart is indication of complete traversal of its scenario, the Büchi criterion is actually not needed. Finite automaton acceptance is sufficient. This can be exploited by composing in parallel with the observer extension of a model a (non-deterministically activating) observer for the pre-chart that drives an additional proposition ' pc ' which holds iff the pre-chart has just been observed. By changing the main-chart observer such that its activation expression is ' pc ' the algorithm from Sect. 3.4.3 still applies directly.

If the observer way is too weak, then $\mathbf{G}(pc \rightarrow \varphi_{mc})$ (or the ACTL correspondence) is checked in steps 3 and 4; φ_{mc} being the main-chart's LTL formula.

4.2.3 Assumptions. The semantics of LSC assumptions is standard. A system satisfies a universal LSC with assumptions iff all runs satisfying all assumptions also satisfy the commitment. Thus, iff

$$(\mathbf{G}(\varphi_{a_1} \wedge \dots \wedge \varphi_{a_m})) \rightarrow (\mathbf{G}(pc \rightarrow \varphi_{mc})), \quad (1)$$

with $\varphi_{a_1}, \dots, \varphi_{a_m}$ the LTL formulae for the assumptions, ' pc ' the pre-chart observer from Sect. 4.2.2, and φ_{mc} the main-chart's LTL formula.

To avoid representing the assumption part as (again large) formulae, we can start with properties which are stronger than (1), i.e. that imply (1), but are easier to check. On the downside, obtaining a counter-example is then no longer *directly* significant as the counter-example may be a false negative. It has to be checked not to be spurious, for example by simulation.

An approach stronger than (1) is to consider the tableau of the parallel composition of all assumptions as an additional observer, i.e. parallel composed to the observer extension. The formula to check is then $(\mathbf{AG} \mathbf{AF} \textit{afair}) \rightarrow (\mathbf{AG} \mathbf{AF} \textit{fair})$ (2) where ' \textit{afair} ' holds iff the tableau is in an accepting state. If (2) passes and if the LSC and all assumptions are bonded, then the system satisfies the LSC. A failure may be a false negative.

An approach stronger than (2) is to check for reachability of a state with $\neg afail \wedge fail$ in the parallel composition of K^A with one iteratively activated observer automaton per assumption, i.e. not the tableaux. If such a state isn't reachable and if the LSC and all assumptions are bonded and time-bounded and no assumption is invariant, then the system satisfies the LSC.

With these three (non exhaustive) options, the algorithm from Sect. 4.2.1 can be extended as follows. Pragmatically we don't consider *all* of the possible combinations with procedures for the commitment. In step 1, only the first option is used for assumptions in order to obtain a reachability property. In each substep of 2, the first option for assumptions can be tried first, followed by the second option from above. In steps 3 and 4, all three options can be tried subsequently, trading size of the formula for size of the model.

Note that a non-bonded assumption is typically easily violated by “clever choice” of transitions in the TSA. Therefore non-bonded assumptions in general are of limited use as an easily violated assumption excludes too many system runs from consideration; the requirement may even be trivially satisfied if all system runs are excluded. As a consequence, all LSCs used in assumption/guarantee style verification should be bonded.

4.2.4 Existential LSCs. Recall from Sect. 2 that an existential LSC is satisfied by a system iff there is at least one system run that adheres to the LSC. Using CTL*, we obtain an equivalent formula by prefixing the LTL formula with the existential quantifiers ‘E’ or ‘EF’ [9].

But the existential mode is different from the universal mode in that one wants to obtain a *witness* if the formula is satisfied. That is, similar to pre-charts, the Büchi criterion isn't used and thus the ACTL/LTL way is practically not relevant. Furthermore, in practice one is typically interested in an example system run that traverses the LSC *completely* [26] instead of taking a legal exit on a cold condition. To achieve all this, the states of the main-chart TSA are turned into non-accepting states except for the state corresponding to the final cut. If the LSC has a pre-chart, the main-chart TSA is again activated by the pre-chart observer ‘*pc*’ as discussed in Sect. 4.2.2. Verifying the existential LSC is then equivalent to reachability of the remaining unique accepting state under all given assumptions.

5 Figures, please.

Table 2 supports our claims with exemplary experimental results. It lists pure model-checking time, i.e. without constructing and loading the transition relation BDD, and without counter-example post processing, for all techniques discussed in Sect. 4⁴. The pre-chart is always an observer and the tableau is used for assumptions. The implementation is taken from [11] and [20]. To isolate the effects under discussion, all experiments use a model that first solves a puzzle to provide some complexity and then adds one or two paths relevant for the LSC.

⁴ (VIS 2.0 [27], Intel Xeon 3.06GHz, 4GByte)

	rch/iter	rch/inv	AGAF/iter	AGAF/inv	ACTL	LTL
Fig.1/tb	36.3s/✗*	48.6s/✗	303.4s/✗	294.9s/✗	755.2s/✗	–
Fig.1/tb	37.3s/✓*	45.0s/✓	41.0s/✓	31.5s/✓	698.8s/✓	–
Fig.1	72.7s/✓	82.2s/✓	81.8s/✓*	57.4s/✓	468.1s/✓	–
Fig.1/as	359.3s/✓	49.2s/✓	47.2s/✓*	29.0s/✓	757.7s/✓	–
Fig.1/nd	56.9s/✓	48.7s/✓	134.2s/✗	128.4s/✗	–	–
[14]/2	31.0s/✓	44.5s/✓	96.3s/✓*	37.5s/✓	258.1s/✓	–
[14]/3	43.6s/✓	121.6s/✓	60.7s/✓*	87.5s/✓	–	–

Table 2. Experimental Results. Model-checking time in seconds.

(✓ = passed, ✗ = failed, – = terminated after 1h, * = first significant result.)

All experiments use a model which first solves a puzzle to provide some complexity and then adds one or two paths relevant for the LSC.

The first (singleton) group is a time bounded version of the LSC from Fig. 1 on a model not satisfying it due to a condition violation. From the table we can see that the reachability-based approach is significantly faster than the others.

The second group uses a model satisfying the LSC. Its first row is the LSC from Fig. 1, ‘as’ is a variant with an assumption, and ‘nd’ is a non-deterministic variant where the condition *Operational* is moved downwards such that it is no longer bonded. In this group, the table indicates that the reachability approach is not always faster, in particular if the property actually holds. In case ‘nd’ we remain inconclusive whether the system satisfies the LSC or not. The ‘✓’ results of the reachability way only indicate that there is no safety violation. The following two ‘✗’ results only indicate that it’s possible to avoid a fair state.

The last group is the example from [14], an untimed but highly concurrent LSC with extraordinary large corresponding formulae. It uses only 6 (9) messages to require that 2 (3) agents are started concurrently, then work concurrently, and then report back concurrently. This LSC is bonded, but not time bounded. From the table we can see that the observer approach allows to establish the property in the given amount of time and space while the formula doesn’t.

Noticeable in the table is that the LTL way is always terminated prematurely, even if the preceding puzzle is removed. If the model has only a single trace and the LSC is changed s.t. it comprises no timing requirements and no concurrency, the task completes within ~1h, thus the LTL way seems rather impractical.

6 Conclusion

Although LSC model-checking is basically solved as each LSC has an equivalent temporal logic formula, in practice direct model-checking of the formula is prohibitively expensive due to its size. Our results indicate that full model-checking power is only necessary for non-bonded LSCs which occur seldom in practice. In contrast, bonded, time-bounded LSCs are as easy as a reachability problem. To the practically most relevant class of bonded, non-time-bounded LSCs, an approach *in between* both techniques applies that uses an observer automaton

and a small fixed liveness formula. Experimental data indicates that it's beneficial to apply the reachability approach even to non-time-bounded LSCs as contradictions are found faster.

Although our criteria classify most practically occurring LSCs correctly, it is of academic interest to have an algorithm *deciding* time boundedness. It would be a minor improvement to identify more states as deterministic by more sophisticated procedures.

References

1. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design* **19** (2001) 45–80
2. Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenarios in system development: Current practice. *IEEE Software* **15** (1998) 34–45
3. Amyot, D., Eberlein, A.: An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunications Systems Journal* **24** (2003) 61–94
4. Knieke, C., Huhn, M., Goltz, U.: Modelling and simulation of an automotive system using LSCs. In Houmb, S.H., Jürjens, J., eds.: *Proc. CSDUML'2005, TUM (2005)* 0–0 TUM-TR.
5. Combes, P., Harel, D., Kugler, H.: Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In: *Proc. ATVA 2005, Number 3707 in LNCS (2005)*
6. Bunker, A., Gopalakrishnan, G., Slink, K.: Live sequence charts applied to hardware requirements specification and verification: A VCI bus interface model. *Software Tools for Technology Transfer* **7** (2004) 341–350
7. Bontemps, Y., Heymans, P., Kugler, H.: Applying LSCs to the specification of an air traffic control system. In: *Proc. SCESM'03. (2003)*
8. Bohn, J., Damm, W., Wittke, H., Klose, J., Moik, A.: Modelling and validating train system applications using state and live sequence charts. In: *Proc. IDPT 2002, Society for Design and Process Science (2002)*
9. Toben, T., Westphal, B.: On the expressive power of LSCs. In Wiedermann, J., et al., eds.: *Proc. SOFSEM 2006, Volume 2., Inst. of CS, AS, Prague (2006)* 33–43
10. Thomas, W.: Automata on infinite objects. In van Leeuwen, J., ed.: *Handbook of theor. comp. sc. (vol. B): Formal Models And Semantics. MIT (1990)* 133–191
11. Klose, J.: *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior. PhD thesis, C. v. O. Universität Oldenburg (2003)*
12. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML Verification Environment. In Cuellar, J.R., Liu, Z., eds.: *Proc. SEFM 2004. (2004)* 174–183
13. Westphal, B.: LSC verification for UML models with unbounded creation and destruction. In Byron Cook, Scott Stoller, W.V., ed.: *Proc. SoftMC 2005, Volume 144:3 of ENTCS., Elsevier B.V. (2005)* 133–145
14. Toben, T., Westphal, B.: Concurrent LSC verification. In Lazic, R., ed.: *Proc. AVoCS 2005, Volume 145 of ENTCS., Elsevier B. V. (2006)* 95–111
15. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: *Proc. FMCAD'02. (2002)* 378–398
16. Harel, D., Kugler, H., Pnueli, A.: Smart play-out extended: Time and forbidden elements. In: *Proceedings International Conference on Quality Software (QSIC'04), IEEE Computer Society Press (2004)*

17. Harel, D., Marelly, R.: *Come, Let's Play*. Springer (2003)
18. Sun, J., Dong, J.S.: Model checking Live Sequence Charts. In: ICECCS, IEEE Computer Society (2005) 529–538
19. Grégoire, B.: Automata oriented program verification. Master's thesis, Facultés Universitaires Notre-Dame de la Paix, Namur (2002)
20. Wittke, H.: A Framework for Specification Verification for Complex Embedded Systems. PhD thesis, C. v. O. Universität Oldenburg (2005)
21. Lettrari, M., Klose, J.: Scenario-based monitoring and testing of real-time UML models. In Gogolla, M., Kobryn, C., eds.: *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. Number 2185 in LNCS, Springer-Verlag (2001) 317–328
22. Klose, J., Kropf, T., Ruf, J.: A visual approach to validating system level designs. In: *Proceedings ISSS'02*, October 2–4, 2002, Kyoto, Japan, ACM (2002)
23. Westphal, B., Toben, T.: The good, the bad and the ugly: Well-formedness of Live Sequence Charts. In Baresi, L., Heckel, R., eds.: *Proc. FASE 2006*. Volume 3922 of LNCS. (2006) 230–246
24. Schlör, R.C.: *Symbolic Timing Diagrams: A Visual Formalism for Model Verification*. PhD thesis, C. v. O. Universität Oldenburg (2000)
25. Kugler, H., et al.: Temporal logic for scenario-based specifications. In Halbwegs, N., Zuck, L.D., eds.: *Proc. TACAS 2005*. Volume 3440 of LNCS. (2005)
26. Brill, M., et al.: Formal verification of LSCs in the development process. In Ehrig, H., et al., eds.: *SoftSpez*. Number 3147 in LNCS. Springer-Verlag (2004) 494–516
27. Brayton, R.K., et al.: VIS: A system for verification and synthesis. In Alur, R., Henzinger, T.A., eds.: *CAV*. Volume 1102 of LNCS., Springer (1996) 428–432

A Proofs

Claim of Lemma 1.3 (corrected):

For non-deterministic \mathcal{A} there is in general no LTL formula λ using only atomic proposition ‘*fair*’ s.t. $K^{\mathcal{A}} \not\models \lambda$ implies $K \not\models \mathcal{A}$. \diamond

In the following we actually prove the more general case of CTL*. The counter-example constructed in the proof follows the idea that in case of a *deterministic* observer, all non-determinism in the observer extension of K is contributed by K . Thus path quantifiers in CTL* always chose between *model* behaviour.

When the observer is non-deterministic, then it adds non-determinism to the overall observer extension that doesn’t stem from the model. By the Büchi semantics, it is acceptable to neglect some of the observer paths but all model paths have to be taken into account. These two cases are not distinguishable in CTL*.

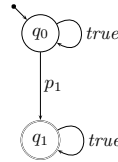
Assuming there was a formula using only ‘*fair*’ for each POTSA, we construct a model which satisfies the formula but actually violates the requirements or fails to satisfy the formula but actually satisfies the requirements.

While this observation affects only a small particular subset of POTSA, it is actually tragic for LSC verification. The POTSA we use as a counter-example actually is the POTSA of the LSC with one non-bonded hot condition with expression p_1 at a hot location. That is, there is no general procedure to check LSCs using the observer way.

Although, the correspondence between bondedness of conditions and applicability of the observer approach is not exact. For instance, the special case that the first non-bonded condition in an LSC is located at a cold cut has the consequence that after reaching this cut, chaos is accepted (the self-loop of the accepting state is annotated by *true*). Thus, as it is the first such condition, there is a language equivalent *bonded* LSC which may be verifiable using the observer way.

Note that this result is only relevant for LSCs. The non-deterministic POTSA obtained for certain Timing Diagrams have a different structure and may not lie in the class of POTSA that aren’t checkable the observer way.

Proof. (of Lemma 1.3) Consider the TSA \mathcal{A} ,



i.e. $Q = \{q_0, q_1\}$, $q_s = q_0$, $C = \emptyset$, $D = \emptyset$, $F = \{q_1\}$, and

$$\rightsquigarrow = \{(q_0, true, true, \emptyset, q_0), (q_1, true, true, \emptyset, q_1), (q_0, p_1, true, \emptyset, q_1)\},$$

where p_1 is an atomic proposition. Note that \mathcal{A} is actually untimed as its clock set is empty.

Let $p_2 \neq p_1$ be another atomic proposition used. Assume there were a formula φ for \mathcal{A} such that for each Kripke structure K compatible with \mathcal{A} we had

$$K \models_{p_2, inv} \mathcal{A} \leftrightarrow K^A \models \varphi.$$

CTL* over $AP = \{fair\}$ is constructed by the following grammar.

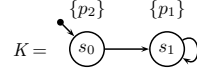
$$\begin{aligned} \sigma &::= fair \mid \neg\sigma \mid \sigma_1 \vee \sigma_2 \mid E\pi \\ \pi &::= \sigma \mid \neg\pi \mid \pi_1 \vee \pi_2 \mid X\pi \mid \pi_1 U \pi_2 \end{aligned}$$

Without loss of generality we can assume that φ is in the following particular disjunctive normalform

$$(\varphi_0 \wedge A\varphi_0^A \wedge E\varphi_0^E) \vee \dots \vee (\varphi_n \wedge A\varphi_n^A \wedge E\varphi_n^E)$$

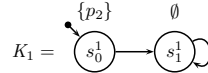
distinguishing non-modal terms φ_i , and universally and existentially quantified terms φ_i^A and φ_i^E .

Consider the following Kripke structure.



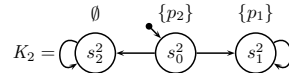
We obviously have $K \models_{p_2, inv} \mathcal{A}$, thus by assumption $K \models \varphi$. Let $(\tilde{\varphi} \wedge E\tilde{\varphi}^E \wedge A\tilde{\varphi}^A)$ be a sub-term of φ that is satisfied by K .

- The A-term $\tilde{\varphi}^A$ is trivially satisfied since otherwise there were a positive occurrence of ‘fair’ relevant to the satisfaction of $\tilde{\varphi}^A$, i.e. a run r of the observer extension where ‘fair’ has to be present to satisfy $\tilde{\varphi}^A$. By the non-deterministic choice of the observer automaton not to change to q_1 and thus not to drive ‘fair’ there is a path of the observer extension which has the same prefix as r but decides not to change to q_1 . This run doesn’t satisfy $\tilde{\varphi}^A$ and thus $A\tilde{\varphi}^A$ isn’t satisfied in contradiction to the assumption.
- The E-term $\tilde{\varphi}^E$ is not trivially satisfied. Otherwise only the non-modal term $\tilde{\varphi}$ would be relevant, i.e. satisfaction would be decided by the (labelling of the) initial state of the Kripke structure. Consequently, for



with $L(S_0)$ the conjunction of labels of the initials states of K assumed above, we had $K_1 \models \tilde{\varphi}$ and thus $K_1 \models \varphi$ while $K_1 \not\models_{p_2, inv} \mathcal{A}$.

- Consider the following modification of K .



As K_2 only adds runs to the behaviour of K , the observer extension $K_2^{\mathcal{A}}$ also has all runs of $K^{\mathcal{A}}$.

So the path satisfying $E \tilde{\varphi}^E$ in $K^{\mathcal{A}}$ is also present in $K_2^{\mathcal{A}}$ and thus $K_2^{\mathcal{A}} \models \varphi$.
But $K_2 \not\models_{p_2, inv} \mathcal{A}$ as the run added to K_2 doesn't adhere to \mathcal{A} . \square

B More Figures, please.

This section presents the complete set of our experimental results for efficient LSC verification.

The `eso_core` model first solves a puzzle to provide some complexity and then adds paths relevant to the LSC. In the `mini_core` model the puzzle is removed.

The mode of the model describes small modifications of the models in order to fulfill or violate the LSC requirements:

- the ‘timefail’ variant doesn’t satisfy the timing constraints of the LSC,
- the ‘savefail’ variant doesn’t satisfy the safety properties of the LSC,
- the ‘livefail’ variant satisfies the safety but not the liveness properties of the LSC, and
- the ‘nondet’ variant provides paths satisfying different runs of the LSC.

The combination of puzzle and model is given in the section name.

The ‘spec’ column of each table names the checked LSC. They are provided in the following section. For each model, mode and LSC the six different ways

- reach/iter,
- reach/inv,
- AGAF/iter,
- AGAF/inv,
- ACTL, and
- LTL

were checked, the outcome is shown in column ‘result’ (the task may pass, fail, or be terminated due to a timeout).

The five last columns list the duration (in seconds) of loading the model into the VIS (‘load’), actually checking the model (‘check’), translating the error-path into a simulation script (‘er2er’), and back-simulating the error-path (‘sim’)

The rightmost column ‘total’ gives the duration of the total task in seconds. This time is typically larger than the sum of the figures to the left as these four times are only the most significant phases of the checking procedure. Yet there are other (fast) phases which don’t dominate the total time but contribute to it.

B.1 Puzzle ‘eso_core’, Model ‘sec_xing’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	5.6	72.7	0.0	0.0	90
	g_inv	pass	11.7	82.2	0.0	0.0	123
	agaf_iter	pass	11.2	81.8	0.0	0.0	123
	agaf_inv	pass	9.1	57.4	0.0	0.0	84
	actl	pass	5.1	468.1	0.0	0.0	482
	ltl	term	5.6	-1	-1	-1	-1
sec_xing_bl	g_iter	pass	10.1	37.3	0.0	0.0	57
	g_inv	pass	18.9	45.0	0.0	0.0	91
	agaf_iter	pass	14.3	41.0	0.0	0.0	82
	agaf_inv	pass	9.5	31.5	0.0	0.0	55
	actl	pass	5.3	698.8	0.0	0.0	726
	ltl	term	5.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	pass	173.6	359.3	0.0	0.0	547
	g_inv	pass	13.6	49.2	0.0	0.0	98
	agaf_iter	pass	15.9	47.2	0.0	0.0	93
	agaf_inv	pass	9.0	29.0	0.0	0.0	51
	actl	pass	6.0	757.7	0.0	0.0	782
	ltl	term	6.0	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	6.9	18.3	0.0	0.0	35
	g_inv	pass	11.7	35.9	0.0	0.0	71
	agaf_iter	pass	8.8	35.8	0.0	0.0	67
	agaf_inv	pass	8.1	23.1	0.0	0.0	50
	actl	pass	5.4	23.1	0.0	0.0	34
	ltl	pass	6.7	28.9	0.0	0.0	59
sec_xing_nbond	g_iter	pass	8.8	56.9	0.0	0.0	79
	g_inv	pass	6.6	48.7	0.0	0.0	66
	agaf_iter	fail	7.3	134.2	0.4	3.8	203
	agaf_inv	fail	7.2	128.4	0.3	3.1	199
	actl	term	6.5	-1	-1	-1	-1
	ltl	term	6.2	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	7.6	20.3	0.0	0.0	40
	g_inv	pass	5.7	21.9	0.0	0.0	36
	agaf_iter	pass	7.3	24.7	0.0	0.0	43
	agaf_inv	pass	5.8	25.0	0.0	0.0	41
	actl	pass	6.2	64.6	0.0	0.0	80
	ltl	pass	5.8	3064.6	0.0	0.0	3156

B.2 Puzzle ‘eso_core’, Model ‘sec_xing_livefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	7.5	25.9	0.0	0.0	43
	g_inv	pass	11.7	42.8	0.0	0.0	84
	agaf_iter	fail	8.0	82.5	0.4	3.0	155
	agaf_inv	fail	6.5	95.3	0.4	2.7	197
	actl	fail	6.2	612.7	0.3	2.4	658
	ltl	term	6.1	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	12.0	35.8	0.3	3.6	60
	g_inv	fail	20.5	53.0	0.5	5.7	115
	agaf_iter	fail	12.2	164.2	0.3	5.7	248
	agaf_inv	fail	15.2	165.7	0.3	5.1	271
	actl	fail	6.1	705.5	0.3	2.4	773
	ltl	term	6.5	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	9.6	39.8	0.4	5.2	64
	g_inv	fail	15.0	53.7	0.5	7.1	118
	agaf_iter	fail	10.1	154.5	0.4	6.7	238
	agaf_inv	fail	9.2	140.3	0.4	5.4	230
	actl	fail	6.2	723.2	0.3	3.0	783
	ltl	term	6.3	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	7.0	5.6	0.0	0.0	22
	g_inv	pass	9.7	6.6	0.0	0.0	42
	agaf_iter	fail	8.4	21.8	0.3	3.4	52
	agaf_inv	fail	7.4	22.8	0.3	3.3	65
	actl	fail	5.8	33.0	0.3	2.5	53
	ltl	fail	6.1	54.1	0.3	0.0	83
sec_xing_nbond	g_iter	pass	6.2	16.9	0.0	0.0	33
	g_inv	pass	6.5	16.1	0.0	0.0	32
	agaf_iter	fail	5.6	68.5	0.3	4.5	135
	agaf_inv	fail	5.8	69.3	0.3	4.9	136
	actl	term	5.8	-1	-1	-1	-1
	ltl	term	5.7	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	7.1	24.2	0.0	0.0	41
	g_inv	pass	6.6	16.6	0.0	0.0	32
	agaf_iter	fail	6.3	22.9	0.3	2.5	46
	agaf_inv	fail	5.6	26.8	0.3	2.8	48
	actl	fail	5.7	63.4	0.3	2.6	85
	ltl	term	5.9	-1	-1	-1	-1

B.3 Puzzle ‘eso_core’, Model ‘sec_xing_nondet’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	5.9	71.9	0.0	0.0	87
	g_inv	pass	9.9	111.3	0.0	0.0	159
	agaf_iter	pass	8.2	32.9	0.0	0.0	57
	agaf_inv	pass	7.0	32.4	0.0	0.0	49
	actl	pass	6.1	603.1	0.0	0.0	626
	ltl	term	5.8	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	22.6	39.0	0.3	8.1	79
	g_inv	fail	36.1	59.1	0.4	10.1	146
	agaf_iter	fail	25.6	293.5	0.4	10.9	397
	agaf_inv	fail	20.8	245.0	0.4	9.0	336
	actl	pass	5.6	723.3	0.0	0.0	763
	ltl	term	6.1	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	pass	86.1	125.5	0.0	0.0	221
	g_inv	pass	22.7	202.7	0.0	0.0	275
	agaf_iter	pass	25.4	217.5	0.0	0.0	268
	agaf_inv	pass	97.0	191.9	0.0	0.0	347
	actl	pass	6.8	777.1	0.0	0.0	820
	ltl	term	6.3	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	5.8	19.0	0.0	0.0	34
	g_inv	pass	10.2	29.9	0.0	0.0	70
	agaf_iter	fail	7.1	185.3	0.4	3.5	276
	agaf_inv	fail	6.5	191.4	0.3	3.4	287
	actl	fail	5.9	88.1	0.3	2.3	110
	ltl	fail	6.2	287.4	0.3	0.0	354
sec_xing_nbond	g_iter	pass	6.8	74.7	0.0	0.0	91
	g_inv	pass	6.3	67.4	0.0	0.0	83
	agaf_iter	fail	5.7	182.8	0.3	5.4	254
	agaf_inv	fail	6.3	181.2	0.3	5.0	253
	actl	term	5.8	-1	-1	-1	-1
	ltl	term	6.2	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	6.4	19.3	0.0	0.0	35
	g_inv	pass	6.4	21.2	0.0	0.0	38
	agaf_iter	pass	5.5	45.3	0.0	0.0	61
	agaf_inv	pass	5.7	46.0	0.0	0.0	62
	actl	pass	5.8	58.3	0.0	0.0	72
	ltl	pass	5.9	3252.2	0.0	0.0	3350

B.4 Puzzle ‘eso_core’, Model ‘sec_xing_safefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	fail	7.6	56.4	0.3	3.2	80
	g_inv	fail	11.7	72.0	0.6	5.0	122
	agaf_iter	fail	8.8	253.1	0.4	3.5	387
	agaf_inv	fail	6.6	244.1	0.3	3.3	343
	actl	fail	5.8	633.3	0.3	2.5	666
	ltl	term	6.1	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	10.5	36.3	0.4	3.2	60
	g_inv	fail	15.4	48.6	0.5	4.6	98
	agaf_iter	fail	10.9	303.4	0.3	2.9	423
	agaf_inv	fail	13.3	294.9	0.4	4.2	403
	actl	fail	6.1	755.2	0.3	1.8	784
	ltl	term	6.8	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	8.7	38.9	0.4	5.1	62
	g_inv	fail	16.7	51.9	0.6	7.1	111
	agaf_iter	fail	13.8	225.3	0.4	7.2	359
	agaf_inv	fail	8.8	243.5	0.4	7.9	367
	actl	fail	5.8	805.5	0.3	4.4	851
	ltl	term	6.3	-1	-1	-1	-1
sec_xing_mini	g_iter	fail	8.2	30.1	0.3	4.8	55
	g_inv	fail	10.2	31.1	0.6	5.2	69
	agaf_iter	fail	9.4	41.6	0.4	3.5	94
	agaf_inv	fail	6.5	37.2	0.4	3.6	74
	actl	fail	5.9	42.4	0.3	2.7	64
	ltl	fail	6.3	42.3	0.3	0.0	73
sec_xing_nbond	g_iter	fail	6.1	24.7	0.3	6.5	46
	g_inv	fail	6.7	22.2	0.4	5.2	43
	agaf_iter	fail	5.6	196.7	0.3	3.2	268
	agaf_inv	fail	5.1	187.1	0.3	3.0	259
	actl	term	5.8	-1	-1	-1	-1
	ltl	term	5.9	-1	-1	-1	-1
sec_xing_ntim	g_iter	fail	7.5	27.1	0.4	3.9	50
	g_inv	fail	6.9	23.9	0.4	3.9	44
	agaf_iter	fail	5.5	44.7	0.3	4.4	74
	agaf_inv	fail	6.6	43.5	0.3	3.4	74
	actl	fail	5.6	84.9	0.3	2.4	111
	ltl	fail	6.0	3415.2	0.3	0.0	3499

B.5 Puzzle ‘eso_core’, Model ‘sec_xing_timefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	fail	6.4	62.7	0.3	3.8	82
	g_inv	fail	8.7	81.0	0.6	6.9	124
	agaf_iter	fail	7.7	400.9	0.4	3.8	540
	agaf_inv	fail	7.6	347.1	0.3	2.7	454
	actl	fail	5.9	650.2	0.3	2.2	687
	ltl	term	6.1	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	9.8	86.9	0.4	3.9	109
	g_inv	fail	14.5	125.3	0.6	4.4	176
	agaf_iter	fail	13.9	349.6	0.4	4.3	483
	agaf_inv	fail	14.8	260.4	0.4	3.4	359
	actl	fail	7.2	743.5	0.4	2.5	806
	ltl	term	6.2	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	160.3	150.3	0.4	117.4	441
	g_inv	fail	219.3	220.6	0.5	154.4	687
	agaf_iter	term	223.6	-1	-1	-1	-1
	agaf_inv	fail	9.1	148.9	0.4	6.3	252
	actl	fail	5.8	767.4	0.5	3.4	834
	ltl	term	6.2	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	6.3	22.6	0.0	0.0	38
	g_inv	pass	8.5	27.8	0.0	0.0	54
	agaf_iter	pass	6.8	23.3	0.0	0.0	51
	agaf_inv	pass	6.5	19.4	0.0	0.0	38
	actl	pass	5.8	29.7	0.0	0.0	41
	ltl	pass	6.2	28.3	0.0	0.0	44
sec_xing_nbond	g_iter	fail	7.6	28.0	0.4	6.0	52
	g_inv	fail	6.4	20.7	0.4	6.3	43
	agaf_iter	fail	5.4	193.1	0.4	3.5	264
	agaf_inv	fail	5.7	200.9	0.4	2.8	270
	actl	term	5.7	-1	-1	-1	-1
	ltl	term	5.7	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	6.5	22.3	0.0	0.0	38
	g_inv	pass	5.8	18.7	0.0	0.0	34
	agaf_iter	pass	6.0	17.4	0.0	0.0	35
	agaf_inv	pass	5.7	18.9	0.0	0.0	34
	actl	pass	5.8	58.5	0.0	0.0	73
	ltl	pass	5.7	3566.2	0.0	0.0	3694

B.6 Puzzle ‘eso_core’, Model ‘sync_par_d’

spec	way	result	load	check	er2er	sim	total
sync_par_cold_a2	g_iter	pass	6.5	23.4	0.0	0.0	40
	g_inv	pass	11.2	41.6	0.0	0.0	88
	agaf_iter	pass	10.7	38.7	0.0	0.0	73
	agaf_inv	pass	7.6	30.7	0.0	0.0	56
	actl	pass	5.5	71.5	0.0	0.0	88
	ltl	term	7.8	-1	-1	-1	-1
sync_par_cold_a3	g_iter	pass	11.4	27.1	0.0	0.0	51
	g_inv	pass	29.9	119.6	0.0	0.0	181
	agaf_iter	pass	16.7	46.3	0.0	0.0	89
	agaf_inv	pass	27.0	93.8	0.0	0.0	148
	actl	term	5.3	-1	-1	-1	-1
	ltl	term	8.7	-1	-1	-1	-1
sync_par_hot_a2	g_iter	pass	6.0	31.0	0.0	0.0	46
	g_inv	pass	8.2	44.5	0.0	0.0	83
	agaf_iter	pass	9.8	96.3	0.0	0.0	134
	agaf_inv	pass	6.8	37.5	0.0	0.0	59
	actl	pass	4.6	258.1	0.0	0.0	381
	ltl	term	5.0	-1	-1	-1	-1
sync_par_hot_a3	g_iter	pass	7.6	43.6	0.0	0.0	65
	g_inv	pass	33.1	121.6	0.0	0.0	197
	agaf_iter	pass	13.2	60.7	0.0	0.0	109
	agaf_inv	pass	29.4	87.5	0.0	0.0	139
	actl	term	4.4	-1	-1	-1	-1
	ltl	term	5.1	-1	-1	-1	-1

B.7 Puzzle ‘mini_core’, Model ‘sec_xing’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	0.1	0.3	0.0	0.0	3
	g_inv	pass	0.2	0.5	0.0	0.0	5
	agaf_iter	pass	0.1	0.4	0.0	0.0	5
	agaf_inv	pass	0.1	0.4	0.0	0.0	4
	actl	pass	0.0	1.9	0.0	0.0	5
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl	g_iter	pass	0.2	0.9	0.0	0.0	4
	g_inv	pass	0.3	1.6	0.0	0.0	8
	agaf_iter	pass	0.3	2.3	0.0	0.0	6
	agaf_inv	pass	0.3	2.2	0.0	0.0	6
	actl	pass	0.0	3.7	0.0	0.0	6
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	pass	0.2	1.2	0.0	0.0	6
	g_inv	pass	0.3	2.1	0.0	0.0	13
	agaf_iter	pass	0.4	2.9	0.0	0.0	8
	agaf_inv	pass	0.3	2.8	0.0	0.0	9
	actl	pass	0.0	4.3	0.0	0.0	8
	ltl	term	0.1	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	0.1	0.2	0.0	0.0	2
	g_inv	pass	0.1	0.2	0.0	0.0	3
	agaf_iter	pass	0.1	0.1	0.0	0.0	5
	agaf_inv	pass	0.1	0.2	0.0	0.0	5
	actl	pass	0.0	0.0	0.0	0.0	3
	ltl	pass	0.0	2.8	0.0	0.0	7
sec_xing_nbond	g_iter	pass	0.1	0.4	0.0	0.0	3
	g_inv	pass	0.1	0.4	0.0	0.0	3
	agaf_iter	pass	0.1	0.5	0.0	0.0	3
	agaf_inv	pass	0.1	0.6	0.0	0.0	3
	actl	pass	0.0	14.5	0.0	0.0	20
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	0.1	0.3	0.0	0.0	3
	g_inv	pass	0.1	0.3	0.0	0.0	3
	agaf_iter	pass	0.1	0.4	0.0	0.0	3
	agaf_inv	pass	0.1	0.3	0.0	0.0	2
	actl	pass	0.0	0.1	0.0	0.0	2
	ltl	pass	0.0	2837.8	0.0	0.0	2933

B.8 Puzzle ‘mini_core’, Model ‘sec_xing_livefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	0.1	0.3	0.0	0.0	3
	g_inv	pass	0.2	0.6	0.0	0.0	6
	agaf_iter	fail	0.2	0.5	0.1	0.1	6
	agaf_inv	fail	0.1	0.4	0.1	0.1	5
	actl	fail	0.0	3.6	0.1	0.0	11
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	0.2	1.6	0.1	0.1	4
	g_inv	fail	0.3	3.7	0.1	0.1	11
	agaf_iter	fail	0.3	2.5	0.1	0.1	8
	agaf_inv	fail	0.2	1.9	0.1	0.1	6
	actl	fail	0.0	14.3	0.0	0.0	28
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	0.2	2.4	0.1	0.1	6
	g_inv	fail	0.5	5.2	0.1	0.1	22
	agaf_iter	fail	0.3	1.9	0.1	0.3	9
	agaf_inv	fail	0.3	1.9	0.1	0.2	10
	actl	fail	0.0	15.2	0.1	0.0	21
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	0.0	0.0	0.0	0.0	2
	g_inv	pass	0.1	0.0	0.0	0.0	5
	agaf_iter	fail	0.1	0.1	0.1	0.0	4
	agaf_inv	fail	0.1	0.0	0.0	0.0	3
	actl	fail	0.0	0.0	0.0	0.0	10
	ltl	fail	0.0	2.8	0.0	0.0	6
sec_xing_nbond	g_iter	pass	0.1	0.3	0.0	0.0	3
	g_inv	pass	0.1	0.3	0.0	0.0	4
	agaf_iter	fail	0.1	0.5	0.1	0.1	5
	agaf_inv	fail	0.1	0.5	0.1	0.1	5
	actl	fail	0.0	20.7	0.0	0.0	26
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	0.1	0.2	0.0	0.0	3
	g_inv	pass	0.1	0.2	0.0	0.0	2
	agaf_iter	fail	0.1	0.3	0.0	0.0	3
	agaf_inv	fail	0.1	0.3	0.1	0.0	3
	actl	fail	0.0	0.1	0.0	0.0	3
	ltl	fail	0.0	3348.2	0.0	0.0	3450

B.9 Puzzle ‘mini_core’, Model ‘sec_xing_nondet’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	pass	0.1	0.5	0.0	0.0	3
	g_inv	pass	0.2	0.9	0.0	0.0	9
	agaf_iter	pass	0.1	0.8	0.0	0.0	4
	agaf_inv	pass	0.1	0.8	0.0	0.0	3
	actl	pass	0.0	2.7	0.0	0.0	6
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	0.2	1.6	0.1	0.1	5
	g_inv	fail	0.4	2.8	0.1	0.1	9
	agaf_iter	fail	0.2	2.5	0.1	0.2	7
	agaf_inv	fail	0.2	2.1	0.1	0.2	7
	actl	pass	0.0	4.5	0.0	0.0	11
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	pass	0.2	1.1	0.0	0.0	4
	g_inv	pass	0.4	1.3	0.0	0.0	12
	agaf_iter	pass	0.2	3.2	0.0	0.0	8
	agaf_inv	pass	0.2	2.7	0.0	0.0	7
	actl	pass	0.0	5.2	0.0	0.0	10
	ltl	term	0.1	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	0.1	0.2	0.0	0.0	2
	g_inv	pass	0.1	0.4	0.0	0.0	5
	agaf_iter	pass	0.1	0.3	0.0	0.0	3
	agaf_inv	pass	0.1	0.3	0.0	0.0	5
	actl	pass	0.0	0.1	0.0	0.0	3
	ltl	pass	0.0	3.4	0.0	0.0	8
sec_xing_nbond	g_iter	pass	0.1	0.4	0.0	0.0	2
	g_inv	pass	0.1	0.4	0.0	0.0	3
	agaf_iter	pass	0.1	1.0	0.0	0.0	4
	agaf_inv	pass	0.1	1.1	0.0	0.0	3
	actl	pass	0.0	16.4	0.0	0.0	21
	ltl	term	0.1	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	0.1	0.5	0.0	0.0	3
	g_inv	pass	0.1	0.4	0.0	0.0	3
	agaf_iter	pass	0.1	0.5	0.0	0.0	3
	agaf_inv	pass	0.1	0.5	0.0	0.0	3
	actl	pass	0.0	0.2	0.0	0.0	2
	ltl	pass	0.1	2702.3	0.0	0.0	2768

B.10 Puzzle ‘mini_core’, Model ‘sec_xing_safefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	fail	0.1	0.4	0.1	0.0	4
	g_inv	fail	0.2	0.5	0.1	0.0	6
	agaf_iter	fail	0.2	0.9	0.1	0.1	11
	agaf_inv	fail	0.1	0.6	0.1	0.1	5
	actl	fail	0.0	3.5	0.1	0.0	7
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	0.2	1.5	0.1	0.0	5
	g_inv	fail	0.3	2.9	0.1	0.1	8
	agaf_iter	fail	0.4	3.1	0.1	0.2	17
	agaf_inv	fail	0.2	1.7	0.1	0.1	8
	actl	fail	0.0	12.4	0.1	0.0	17
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	0.3	3.0	0.1	0.1	7
	g_inv	fail	0.3	3.3	0.1	0.1	13
	agaf_iter	fail	0.3	3.8	0.1	0.4	16
	agaf_inv	fail	0.2	2.6	0.1	0.2	11
	actl	fail	0.0	12.5	0.1	0.0	19
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_mini	g_iter	fail	0.1	0.2	0.1	0.0	4
	g_inv	fail	0.1	0.2	0.1	0.0	5
	agaf_iter	fail	0.1	0.3	0.1	0.0	5
	agaf_inv	fail	0.0	0.2	0.0	0.0	4
	actl	fail	0.0	0.0	0.0	0.0	6
	ltl	fail	0.0	2.8	0.0	0.0	7
sec_xing_nbond	g_iter	fail	0.2	0.6	0.1	0.0	4
	g_inv	fail	0.1	0.4	0.0	0.0	3
	agaf_iter	fail	0.2	0.9	0.1	0.1	7
	agaf_inv	fail	0.1	0.7	0.1	0.1	6
	actl	fail	0.0	21.6	0.1	0.0	29
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_ntim	g_iter	fail	0.1	0.5	0.1	0.0	4
	g_inv	fail	0.1	0.3	0.1	0.0	4
	agaf_iter	fail	0.1	0.4	0.1	0.0	4
	agaf_inv	fail	0.1	0.3	0.1	0.0	3
	actl	fail	0.0	0.1	0.0	0.0	2
	ltl	fail	0.0	2980.1	0.0	0.0	3062

B.11 Puzzle ‘mini_core’, Model ‘sec_xing_timefail’

spec	way	result	load	check	er2er	sim	total
sec_xing	g_iter	fail	0.1	0.3	0.1	0.0	3
	g_inv	fail	0.2	0.6	0.1	0.0	7
	agaf_iter	fail	0.1	0.6	0.1	0.1	6
	agaf_inv	fail	0.1	0.6	0.1	0.1	6
	actl	fail	0.0	3.4	0.0	0.0	7
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl	g_iter	fail	0.2	0.9	0.1	0.0	3
	g_inv	fail	0.3	1.6	0.1	0.1	6
	agaf_iter	fail	0.2	2.7	0.1	0.2	13
	agaf_inv	fail	0.2	3.2	0.1	0.2	9
	actl	fail	0.0	12.2	0.0	0.0	16
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_bl_ass	g_iter	fail	0.2	2.1	0.1	0.1	6
	g_inv	fail	0.4	4.4	0.1	0.1	11
	agaf_iter	fail	0.2	2.2	0.1	0.2	11
	agaf_inv	fail	0.2	2.3	0.1	0.2	10
	actl	fail	0.0	13.3	0.0	0.0	21
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_mini	g_iter	pass	0.0	0.1	0.0	0.0	2
	g_inv	pass	0.1	0.1	0.0	0.0	5
	agaf_iter	pass	0.0	0.2	0.0	0.0	5
	agaf_inv	pass	0.1	0.1	0.0	0.0	4
	actl	pass	0.0	0.0	0.0	0.0	3
	ltl	pass	0.0	2.8	0.0	0.0	6
sec_xing_nbond	g_iter	fail	0.1	0.6	0.1	0.0	3
	g_inv	fail	0.1	0.8	0.1	0.0	4
	agaf_iter	fail	0.1	0.7	0.1	0.1	6
	agaf_inv	fail	0.1	0.6	0.1	0.1	6
	actl	fail	0.0	22.9	0.1	0.0	29
	ltl	term	0.0	-1	-1	-1	-1
sec_xing_ntim	g_iter	pass	0.1	0.2	0.0	0.0	3
	g_inv	pass	0.1	0.3	0.0	0.0	3
	agaf_iter	pass	0.1	0.4	0.0	0.0	3
	agaf_inv	pass	0.1	0.3	0.0	0.0	3
	actl	pass	0.0	0.1	0.0	0.0	2
	ltl	pass	0.0	2965.3	0.0	0.0	3010

B.12 Puzzle ‘mini_core’, Model ‘sync_par_d’

spec	way	result	load	check	er2er	sim	total
sync_par_cold_a2	g_iter	pass	0.1	0.4	0.0	0.0	6
	g_inv	pass	0.1	0.7	0.0	0.0	14
	agaf_iter	pass	0.1	1.0	0.0	0.0	8
	agaf_inv	pass	0.1	0.6	0.0	0.0	3
	actl	pass	0.0	0.2	0.0	0.0	5
	ltl	term	0.0	-1	-1	-1	-1
sync_par_cold_a3	g_iter	pass	0.3	1.4	0.0	0.0	5
	g_inv	pass	0.3	2.6	0.0	0.0	9
	agaf_iter	pass	0.5	15.5	0.0	0.0	27
	agaf_inv	pass	0.3	4.0	0.0	0.0	8
	actl	term	0.0	-1	-1	-1	-1
	ltl	term	0.0	-1	-1	-1	-1
sync_par_hot_a2	g_iter	pass	0.1	0.4	0.0	0.0	10
	g_inv	pass	0.1	0.7	0.0	0.0	14
	agaf_iter	pass	0.1	0.6	0.0	0.0	4
	agaf_inv	pass	0.1	0.3	0.0	0.0	11
	actl	pass	0.0	0.7	0.0	0.0	4
	ltl	term	0.0	-1	-1	-1	-1
sync_par_hot_a3	g_iter	pass	0.1	1.4	0.0	0.0	7
	g_inv	pass	0.7	1.8	0.0	0.0	16
	agaf_iter	pass	0.3	9.6	0.0	0.0	17
	agaf_inv	pass	0.5	3.4	0.0	0.0	7
	actl	pass	0.0	1751.4	0.0	0.0	1789
	ltl	term	0.0	-1	-1	-1	-1

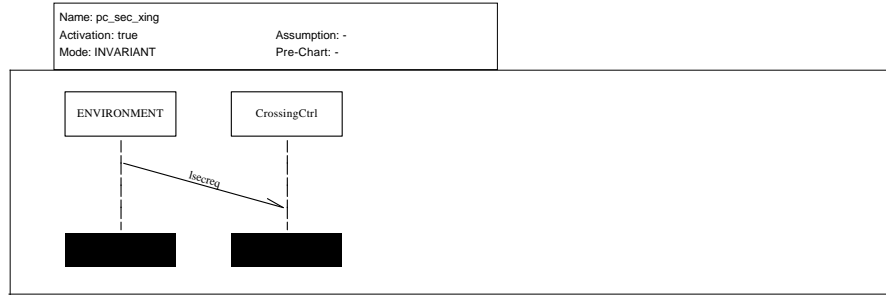
C LSCs

Note that the tool LSCEditor(1) we use to draw LSCs graphically separates the charts for the Pre- and Main-Chart. Thus unlike in Fig. 1, the precharts in the following printouts appear as *separate* charts and are linked to the main-chart by name references in the instance head.

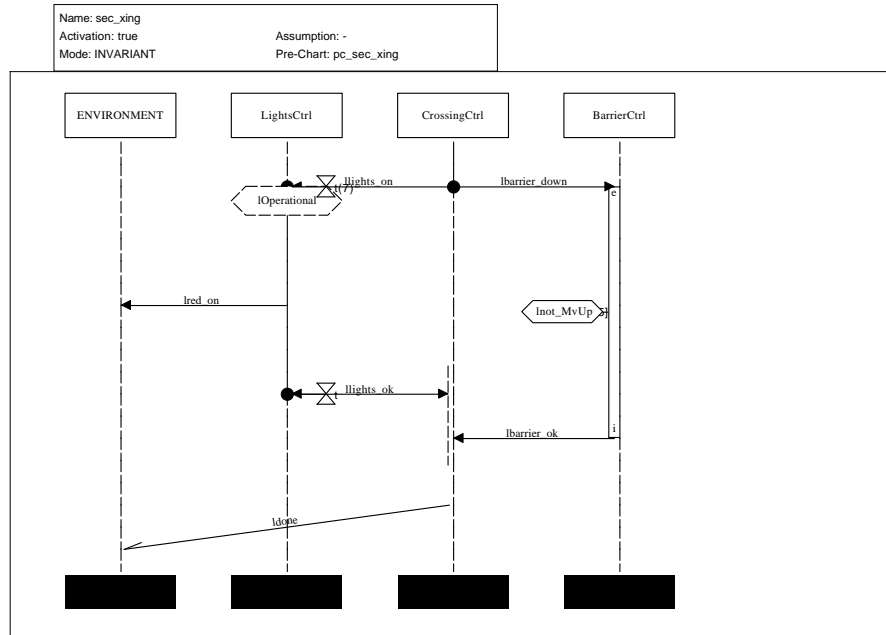
C.1 'sec_xing'

(denoted as 'Fig. ??' in Table 2 in the paper) The original 'sec_xing' LSC (*the mainchart's TSA is deterministic but not time bounded*).

Pre-Chart:



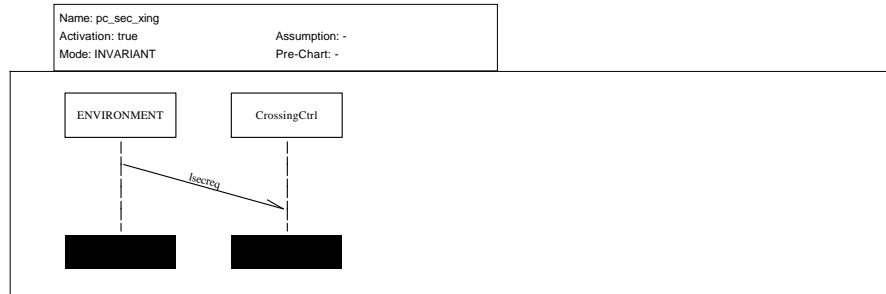
Main-Chart:



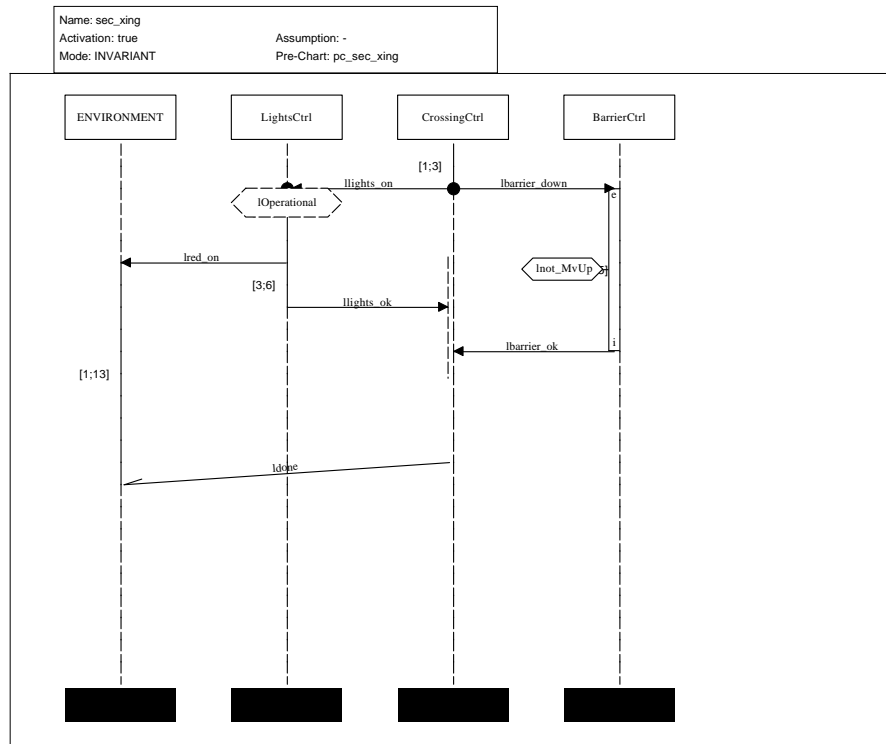
C.2 sec_xing_bl

(denoted as 'Fig. 1/tb' in Table 2) Time bounded modification of 'sec_xing' (the mainchart's TSA is deterministic and time bounded).

Pre-Chart:



Main-Chart:

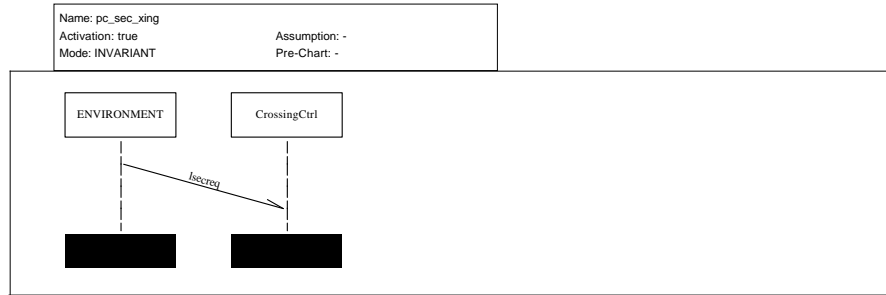


C.3 sec_xing_bl_ass

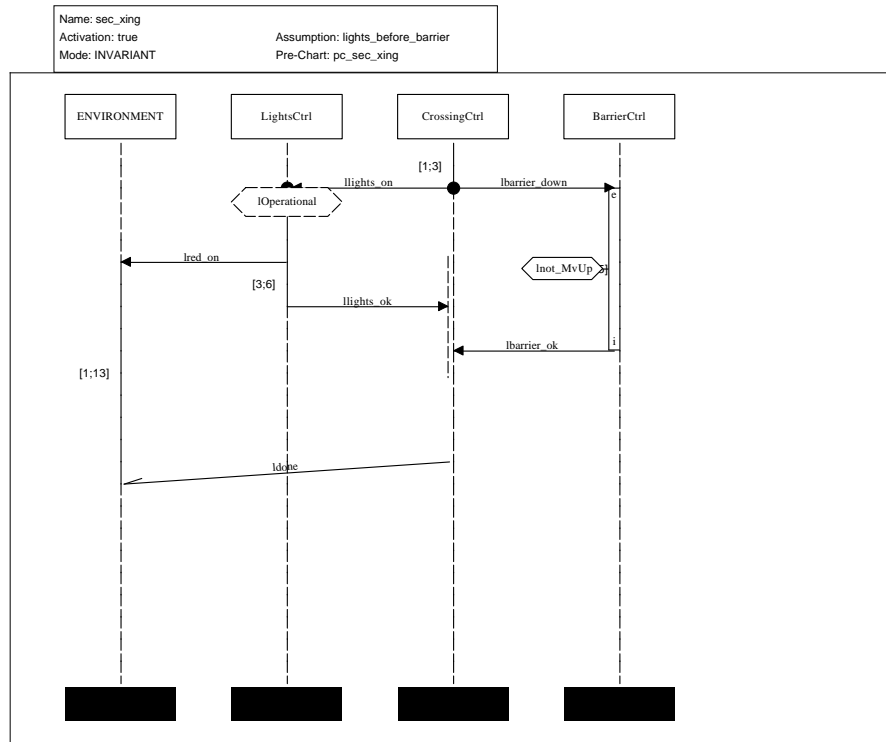
(denoted as 'Fig. 1/as' in Table 2)

sec_xing_bl_ass: 'sec_xing_bl' with assumption of a particular message order
(the mainchart's TSA is deterministic and time bounded)

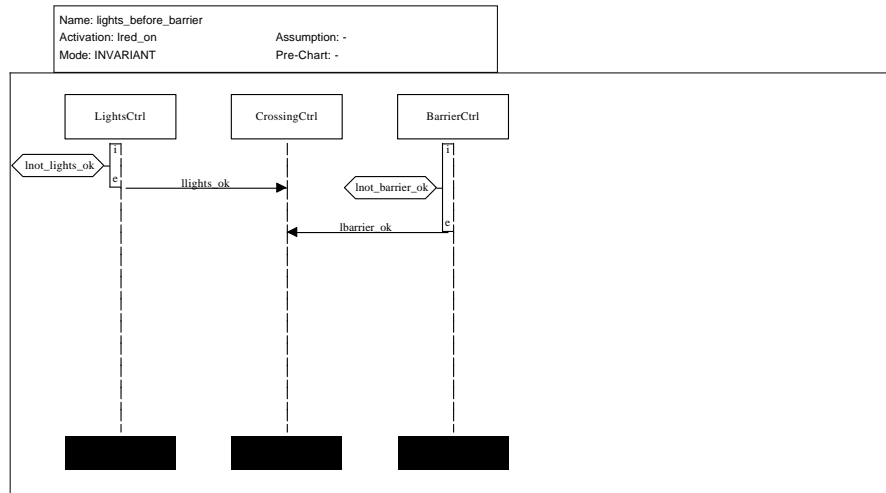
Pre-Chart:



Main-Chart:



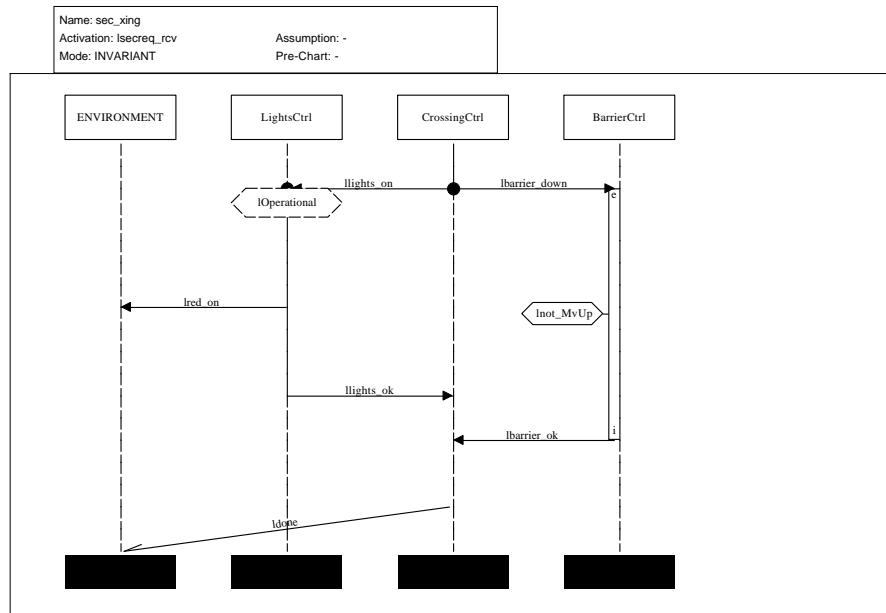
Assumption:



C.4 sec_xing_mini

Very reduced version without any timing requirements, without pre-chart, and without coregion (*the mainchart's TSA is deterministic but not time bounded*).

Main-Chart:

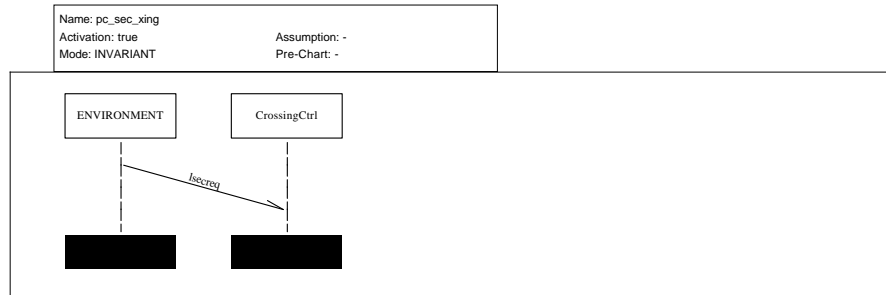


C.5 sec_xing_nbond

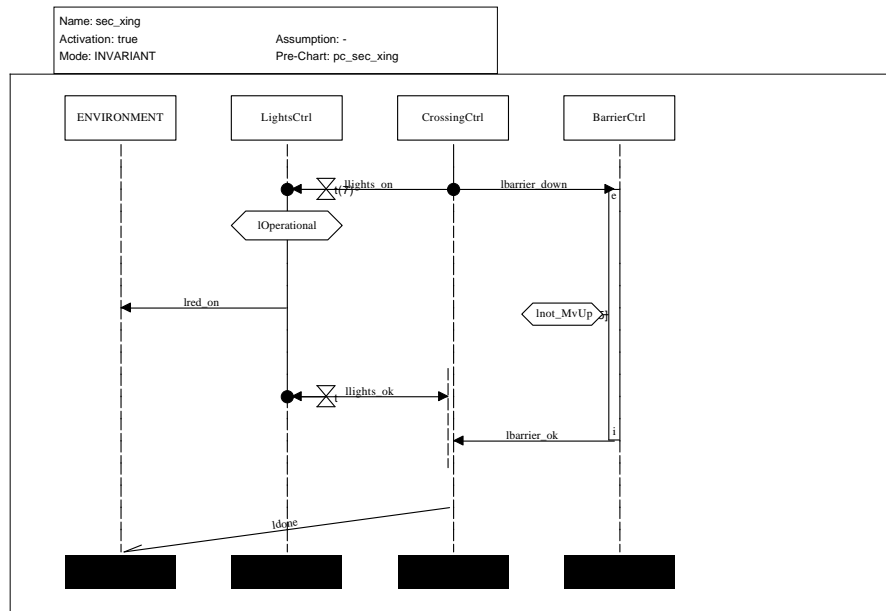
(denoted as 'Fig 1/nb' in Table 2 in the paper)

The condition 'Conditional' turned hot and moved from the simregion such that it is floating now (*the mainchart's TSA is not deterministic and not time bounded*).

Pre-Chart:



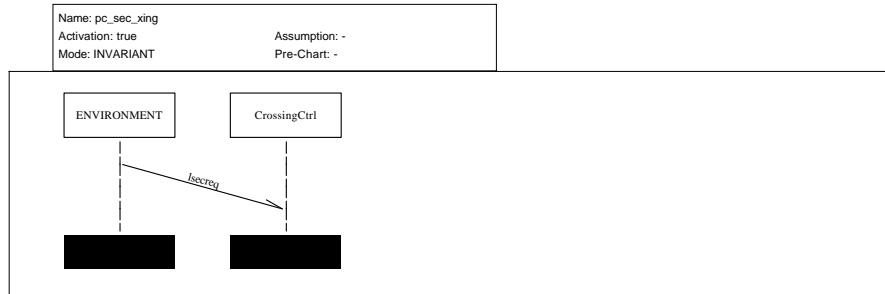
Main-Chart:



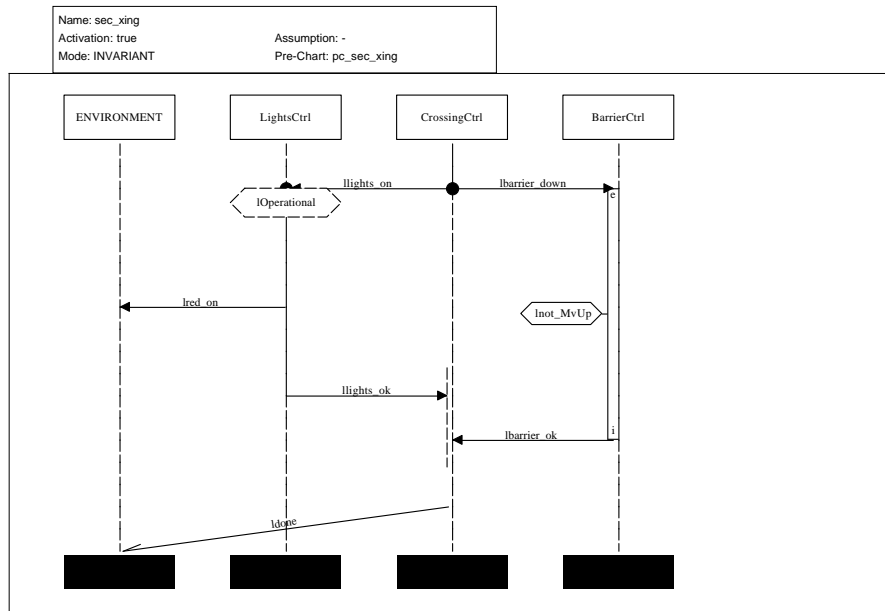
C.6 sec_xing_ntim

Teduced version without any timing requirements (*the mainchart's TSA is deterministic but not time bounded*).

Pre-Chart:



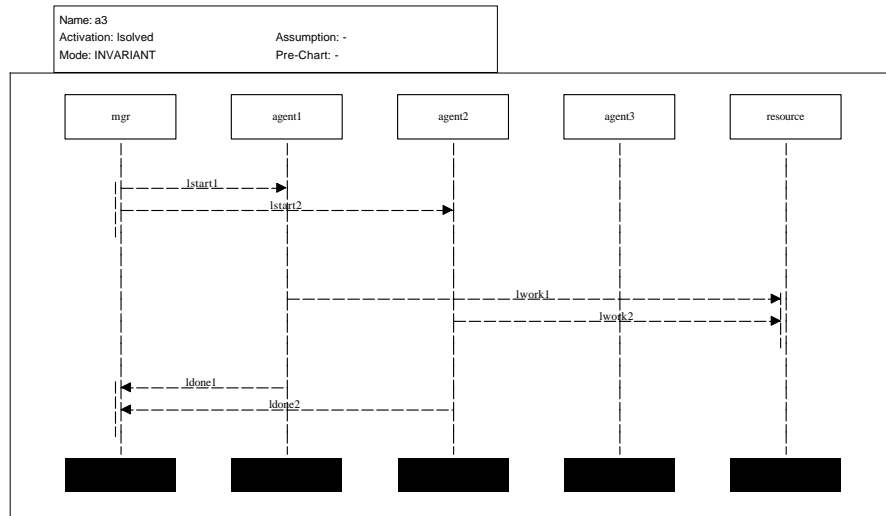
Main-Chart:



C.7 sync_par_cold_a2

Three times two concurrent messages; cold messages and instances (*the main-chart's TSA is deterministic but not time bounded*).

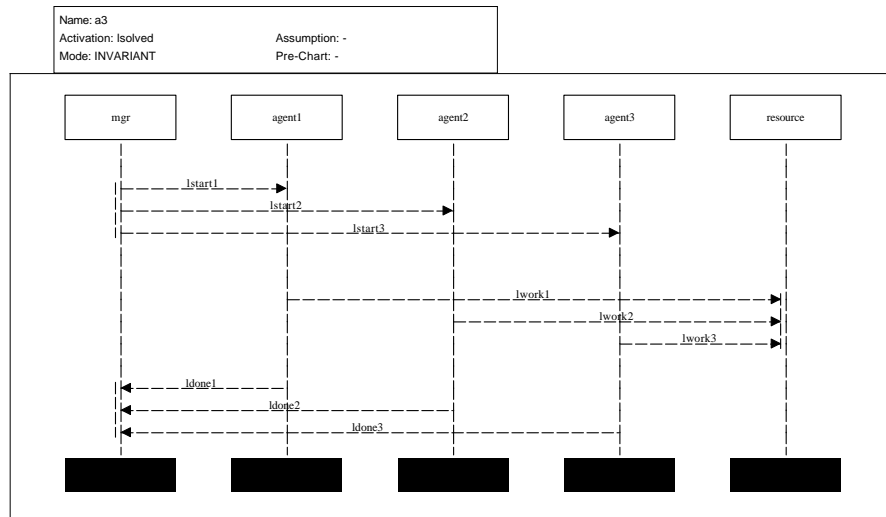
Main-Chart:



C.8 sync_par_cold_a3

Three times three concurrent messages; cold messages and instances (*the main-chart's TSA is deterministic but not time bounded*).

Main-Chart:

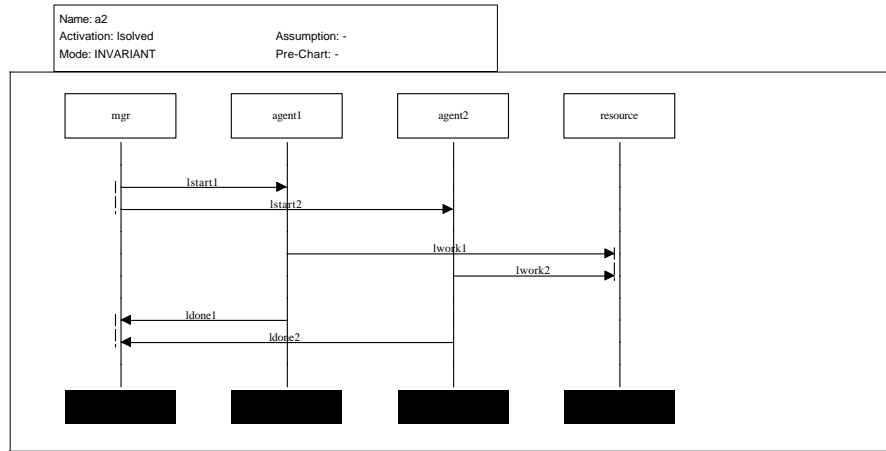


C.9 sync_par_hot_a2

(denoted as '[14]/2' in Table 2)

Three times two concurrent messages; hot messages and instances (*the main-chart's TSA is deterministic but not time bounded*).

Main-Chart:



C.10 sync_par_hot_a3

(denoted as '[14]/3' in Table 2)

Three times three concurrent messages; hot messages and instances (*the mainchart's TSA is deterministic but not time bounded*).

Main-Chart:

