Centre Fédéré en Vérification

Technical Report number 2004.34

Lightweight Formal Methods for Scenario-based Software Engineering

Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens

This work was partially supported by a FRFC grant: 2.4530.02

http://www.ulb.ac.be/di/ssd/cfv

Lightweight Formal Methods for Scenario-based Software Engineering

Yves Bontemps[⋆], Patrick Heymans, and Pierre-Yves Schobbens

Institut d'Informatique, University of Namur
rue Grandgagnage, 21
B5000 - Namur (Belgium)
{ybo,phe,pys}@info.fundp.ac.be

1 Introduction

The difficulty to produce quality software requirements has long been identified [2]. They all too often turn out to be unsuitable, incomplete, ambiguous, contradictory, redundant, continually changing, and so on. Researchers and practitioners have devoted much efforts trying to find solutions to this. *Scenario*-oriented solutions are among the most successful attempts. They became increasingly popular over the past ten years, through the widespread adoption of UML[1] and Use Cases[2]. But, *Scenario-Based Software Engineering* (SBSE) actually covers a wider family of techniques expanding over elicitation, specification, verification, validation, inspection, prototyping, animation, negotiation,... [**?**,**?**,**?**]. We are mostly interested in scenario-based specifications, their verification and their use for code generation.

Specification techniques range from the most informal ones to those having a precise, mathematically defined semantics. Our focus is on the latter, which are a necessary prerequisite to unambiguous specification and efficient automation.

Our contributions concern Live Sequence Charts (LSCs) [3], a notation introduced by David Harel and Werner Damm in order to overcome some limitations of Message Sequence Charts (MSCs) [4], namely, the lack of *message abstraction* and the inability to specify the *modality* of a scenario. As an example, consider the following standard distributed system requirement: *"Whenever a process enters the critical section, it eventually exits it"*. Fig. 1 represents the corresponding MSC.

If this scenario is to be interpreted as recommended by the ITU standard, it means that *Start_using* is followed by *Stop_using*, without any other message being exchanged in the interval. This entails that the process may not send any requests to the critical resource when using it! Of course, the intended meaning of this scenario is different: the process starts using the critical section and after some time, during which no message *relevant to this requirement* is sent, it releases its lock on the critical resource.

The status or modality of the behaviour described by the MSC is also unclear: is it a simple example only used for illustrative purposes or is it a universal rule

[⋆] FNRS Research Fellow

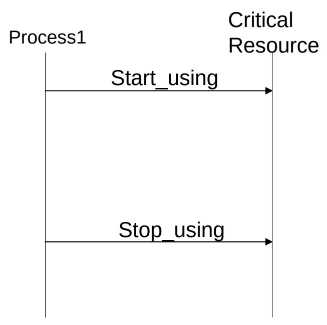


Fig. 1. Critical section requirement specified with MSC

("In given circumstances, the system shall always behave as specified")? Thus LSCs abstract away irrelevant messages and attach a modality to each scenario: universal, example (existential) or even counter-example (anti-scenario), as we will see in Section 3.2.

A typical SBSE process (see Fig. 2) is usually based on Use Cases. In such a process, one progressively moves from concrete, partial examples (or counter-examples) of behaviour to more general requirements statements. This way of doing fosters communication between software engineers and the other stakeholders [**?**]. Additionally, it facilitates the identification of test cases and the production of user-documentation. As this human-intensive bottom-up elicitation task progresses, the precision of the corresponding documentation should also evolve from informal, error-prone representations to more formal models. Hence, LSCs with their multiple modalities, intuitive MSC-like syntax and their formal semantics, seem worth considering to support the task.

But this is only the start of the process. What we devote our interest to in this paper are the subsequent steps. In his vision paper [5], David Harel essentially sees it as building a system model (made of two interrelated models, a structural model and a state-based behavioural model) and then producing code from it. He foresees a bright future in which the *synthesis* and *verification* will be formal and largely automated (see Fig. 2). The present paper is a first step in this direction.

A challenge that we face at this point is to transforming scenarios into a system model (and subsequently into code). This is a major *paradigm shift.* We start with a *scenario-based inter-component perspective* (scenarios typically describe component interactions vs internal actions) and we end with a *state-based intra-component perspective.* General techniques are thus computationally extremely expensive. We show here how to improve on this cost. In counterpart, we have to abandon the idea to provide exhaustive algorithms, and just keep

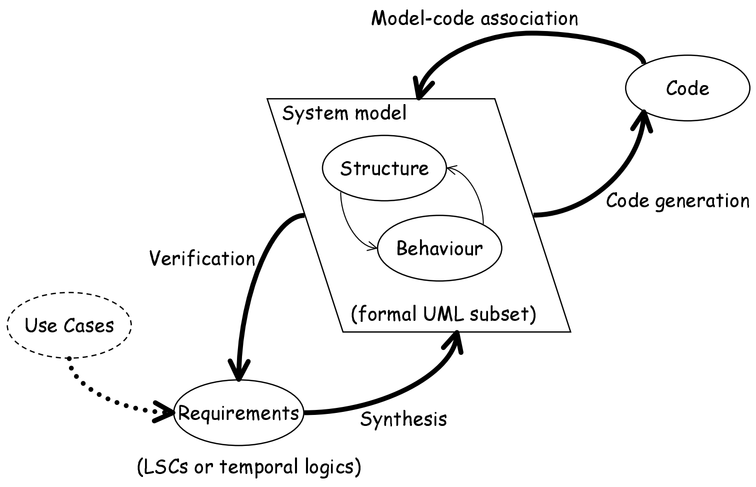


Fig. 2. Scenario-based Software Engineering (adapted from [5])

soundness. Still, we are convinced that our algorithms are effective in detecting specification problems and generating implementations.

2 Running Example

To illustrate our approach, we will use excerpts from a variation of the Center TRACON System (CTAS Case Study) from NASA [7, 8]. This system coordinates various air traffic related clients, in order to ensure that they all use the same weather information. We will focus on the part of the system in charge of updating the weather reports used by clients. The system is made of the following components:

Weather Control Panel (WCP): the User Interface through which operators manually trigger updates;

Communication Manager (CM): the central part of the system, in charge of synchronizing the various clients;

Client: those are distributed on the various sites, where accurate weather data is needed. We will assume that there are only two clients and that they are already connected to the system. In the original system, there is a part of the system in charge of connecting, disconnecting and initializing clients;

Database: from which the clients retrieve weather reports. We assume that there is only one database to which all clients direct their queries;

Terminal: represents the computers on the distributed site that make use of the weather reports downloaded by clients.

3 Models and Relationships

3.1 Structure

Fig. 3 gives the structural view of our example through a variant of object diagrams. Boxes represent agents. Associations (arrows) between agents are directed, denoting one-way communication channels. They are typed by the names of the messages/events they carry.

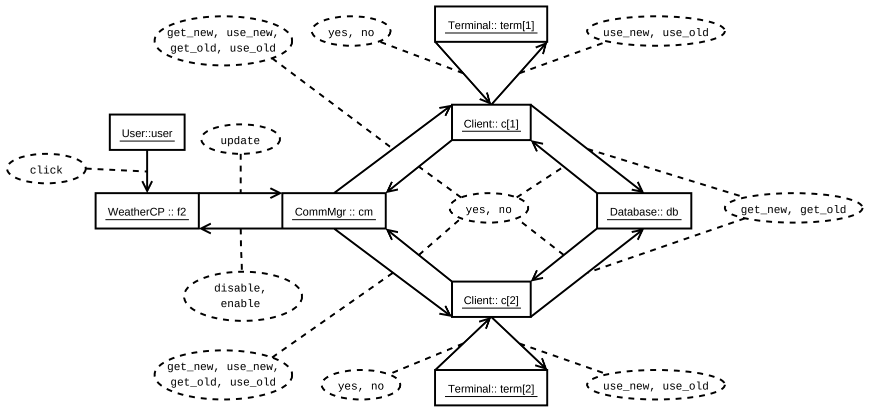


Fig. 3. Structure model

3.2 Inter-agent specifications

Inter-agent specifications are partial "one story for all agents" [10][1] scenario-based descriptions constraining the overall system behaviour.

An LSC is similar to a Sequence Diagram [1] or a bMSC [4]. Fig.4 presents two universal LSCs. An event (arrow) is instantaneous and can only appear between agent instances (vertical *lifelines*) which classes were declared to control or receive it, respectively, in the structure diagram. The points of a lifeline where events occur (i.e. the sources and targets of arrows) are called *locations*. On a given lifeline, locations are ordered chronologically from top to bottom. Events being instantaneous, senders and receivers synchronize on them.

Universal scenarios (uLSCs) embeds a general trigger-response pattern as well as a frame axiom restricting which events can, must or cannot happen during the execution of a scenario. uLSCs consist of two concatenated *basic charts*: the *prechart* (i.e. the trigger) and the *main chart* (i.e. the response). The former is surrounded by a dashed hexagon. The latter comes below the prechart within a solid rectangle (see fig.4). The scenario in fig.4(a) asserts that, whenever the user

[1] [10] speaks in terms of objects rather than agents.

`click`s on the weather control panel `f2` and `f2` sends an `update` order to `cm`, `cm` must `disable` `f2` and set its own status to "updating" through `status_up`. Because all events appearing in the scenario are restricted, this scenario forbids the occurrence of `click` or `update` between `disable` and `status_up`.

The uLSC in fig 4(b) contains an ALT-box: only one of the two subboxes is chosen. ALT-boxes are treated in [11], where all results are carried over.

Abstract Syntax In compliance with the semantics of MSCs [4, 12], a basic chart defines a *labeled partial order* on events [13]. First of all, events (arrow labels) are in Σ. We assume that events contain information about their sender and receiver. Let Σ_a^s (resp. Σ_a^r) be the set of events sent (resp. received), by agent a. Locations are sources and targets of arrows. Two locations l and l' are directly ordered if they belong to the same lifeline and l is drawn higher up than l'. Since communication is instantaneous, the two locations of a same arrow shall be reached simultaneously; hence, they are order-equivalent. The transitive closure of this direct ordering defines a *preorder.* All locations of an equivalence class must be labeled by the same event. This ensures that the quotient of the preorder defines a labeled partial order.

Definition 1 (Labeled partial order (LPO)). *A Σ-labeled partial order (LPO) is a tuple $\langle L, \leq, \lambda\rangle$, where*

- *L is a set of* locations*. If L is finite, the LPO is called* finite*.*
- *$\leq \subseteq L \times L$ is a partial order on L (a transitive, anti-symmetric and reflexive relation).*
- *$\lambda : L \to \Sigma$ is a labeling function associating events to locations.*

The LPO is deterministic *if furthermore $\forall l, l' \in L : \lambda(l) = \lambda(l') \implies l \leq l' \vee l' \leq l$. A* linearization *of a finite LPO is a word of $w_1 \dots w_n \in \Sigma^*$ such that its canonical LPO $\langle [n], \leq, \{(i, w_i) | i \in [n]\}\rangle$, where $[n]$ is a shortcut for the set $\{1, \dots, n\}$, is isomorphic to some linear (total) order $\langle L, \leq', \lambda\rangle$ with $\leq \subseteq \leq'$. An* ideal *(or* cut*) in an LPO is a set $c \subseteq L$ such that $\forall l \in c : \forall l' \in L : l' \leq l \implies l' \in c$. By abuse of language, we call "ideal" the LPO resulting in the projection of an LPO on a given ideal.*

Using ideals, one can define the following transition system. It has the property that $\emptyset \xrightarrow{w}^* c$ if, and only if, c is linearized by w.

Definition 2 (Ideals Transition System, $c \xrightarrow{e} c'$). *The states of this transition system are ideals in the considered LPO (see def. 1). Given two ideals c and c', there is a transition labeled by some event e (written $c \xrightarrow{e} c'$) iff there is an e-labeled location l which has all its predecessors in c, but is not in c and $c' = c \cup \{l\}$.*

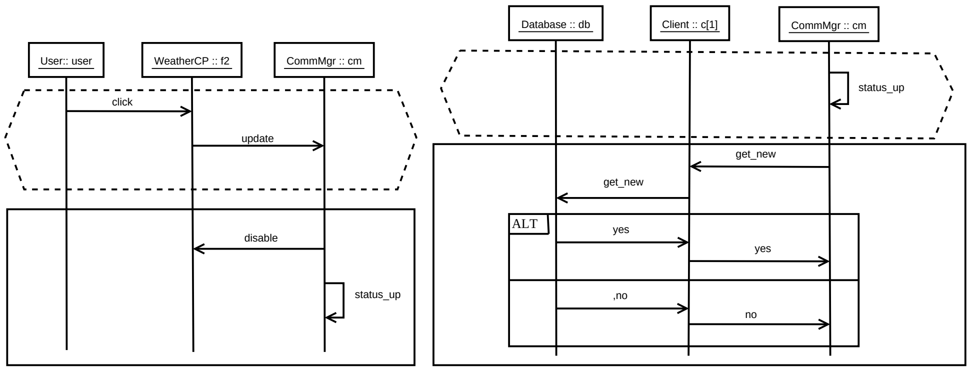
Definition 3 (Universal LSC or uLSC).
A uLSC is a tuple $\langle L, \leq, \lambda, \Sigma_R, P\rangle$ such that

1. *$\langle L, \leq, \lambda \rangle$ is a deterministic Σ_R-LPO. Σ_R contains the* restricted *events*[2] *of the uLSC;*
2. *$P \subseteq L$ is the prechart. Every prechart location should occur before a main chart location: $P \times (L \setminus P) \subseteq \leq$.*

Semantics The semantics of a uLSC is, like linear temporal logics (LTL), given in terms of a *model relation*: for every possible infinite sequence of events $\gamma \in \Sigma^\omega$, we say that γ is a *model* of a uLSC $S = \langle L, \leq, \lambda, \Sigma_R, P \rangle$ (written $\gamma \models S$) iff, for every decomposition $uv\gamma'$ of γ ($u, v \in \Sigma^*$ and $\gamma' \in \Sigma^\omega$), if $v|_{\Sigma_R}$ linearizes P, then

$$\exists w \in \Sigma^* : \gamma' = w\gamma'' \wedge w|_{\Sigma_R} \text{ linearizes } M.$$

The language of a uLSC is its set of models. An LSC specification (say $\mathcal{S}$) is a set of uLSCs and its language ($\mathcal{L}(\mathcal{S})$) is the intersection of the languages of its component uLSCs.



(a) (b)

Fig. 4. Universal Live Sequence Charts (uLSCs)

We decompose the constraint expressed by a uLSC per event. Each participating agent will be responsible for the constraints linked to the events he sends. Consider a finite run $w \in \Sigma^*$ and an uLSC $S = \langle L, \leq, \lambda, \Sigma_R, P \rangle$. We say that this run *activates a location l in S* if there is some suffix v of w such that we can find an ideal c in S

1. in which l is maximal ($l \in c$ and $\forall l' > l : l' \notin c$),
2. which contains the prechart ($c \supseteq P$),
3. which does not contain all the locations ($c \subset L$),
4. which has $v|_{\Sigma_R}$ among its linearizations.

[2] That is, roughly, those events that must take place at the moments determined by the uLSC but cannot happen elsewhere while the scenario's main chart is "executing".

Definition 4 (required event, forbidden event). *If w activates some l such that $e \in \Sigma_R$ labels one of the direct successors of l, then we say that w* requires *e. Conversely, if w activates some l such that $e \in \Sigma_R$ does not label any of the direct successors of l, e is said to be* forbidden *by w.*

Definition 5 (e-safety, e-liveness). *A run $\gamma \in \Sigma^\omega$ is e-safe (resp. e-live) iff for every finite prefix $w \in \Sigma^*$ of γ, if w forbids (resp. requires) e, then $w \cdot e$ is not a prefix of γ (resp. $\exists v : w \cdot v \cdot e$ is a prefix of γ).*

The following theorem asserts that, by checking that forbidden events do not occur and required events eventually occur, we are sure that an LSC will be satisfied.

Theorem 1 (uLSC = Σ-liveness $\cap$ Σ-safety [13]). *For every $\gamma \in \Sigma^\omega$, $\gamma \models S$ iff γ is e-safe and e-live, for every $e \in \Sigma_R$.*

Our ultimate goal is to build an open reactive system. The structure diagram shows all the agents interacting in the application domain. Some of them are system agents, i.e. components of the system that we are in charge of building, while other are environment agents, whose behaviour is beyond our control.

Let $Sys \subset \mathrm{Ag}$ be the set of "system" agents. Their controlled events are $\Sigma_{Sys} = \bigcup_{a \in Sys} \Sigma_a^s$. We define similarly $\Sigma_{Env} = \bigcup_{a \in Env} \Sigma_a^s$, the set of events controlled by the environment, as $Env = \mathrm{Ag} \setminus Sys$).

As we already highlighted, a uLSC constrains how the various agents interact, by forcing them to behave as prescribed in the main chart, when they have been interacting as in the prechart. This constrains the system as well as its environment. Hence, when designing the system, we may safely assume that the environment will fulfill its safety and liveness obligations. This leads to the natural notion of "implementation correctness".

Definition 6 (Correct implementation). *Let Σ be partitioned into Σ_{Sys}, the set of system-controlled events, and Σ_{Env}, the set of environment-controlled events. A set of words $W \subseteq \Sigma^\omega$ is a* correct implementation *of a uLSC iff*

$$\forall \gamma \in W. \begin{cases} \gamma \textit{ is } \Sigma_{Env}\textit{-safe} \implies \gamma \textit{ is } \Sigma_{Sys}\textit{-safe} \\ \gamma \textit{ is } \Sigma_{Env}\textit{-live} \implies \gamma \textit{ is } \Sigma_{Sys}\textit{-live} \end{cases}$$

So, we end up with a system that will guarantee the satisfaction of its specification, provided its environment behaves as assumed.

3.3 Intra-Agent Specifications

We use a variant of the formalism of finite Input/Output Automata for specifying the behaviour of each agent separately. This formalism has been introduced in [14], originally without the restriction of being finite state. It is a conceptually simple model, which allows us to focus on proofs, abstracting from syntactic and semantic complexities. By its very nature, this formalism is adapted for describing distributed systems, when the focus is on interaction and synchronization.

Indeed, the components specified are robust to scheduling; they may not make any assumptions on the relative speed of their environment. Furthermore, since we are interested in open systems, this formalism acknowledges the fact that no component can constrain its environment's behavior; this is guaranteed by the syntactic condition called "input-enabledness". Finally, to effectively support component-based software engineering, our model must make it possible to hierarchically build components from subcomponents, which shall themselves be open systems, while keeping refinement in mind [15]. The framework of I/O automata has composition as a first-class citizen, which guarantees refinement.

Abstract Syntax

Definition 7 (I/O Automaton). *An input-output automaton is a tuple*

$$\langle \Sigma_i, \Sigma_o, Q, q_0, \Delta, \mathcal{P} \rangle,$$

- $\Sigma_i \subseteq \Sigma$ *is a set of input events;*
- $\Sigma_o \subseteq \Sigma$ *is a set of output events. Input and output events are disjoint;*
- Q *is a finite set of states;*
- q_0 *is an initial state;*
- $\Delta \subseteq Q \times (\Sigma_i \cup \Sigma_o) \times Q$ *is a transition relation. An I/O Automaton must be* input-enabled*: for every state q and input event $e \in \Sigma_i$, there must be a state q' such that $\Delta(q, e, q')$;*
- $\mathcal{P} \subseteq 2^{\Sigma_o}$ *is a fairness partition. It is a set of equivalence classes between output events that must be treated fairly (see def. below).*

Semantics A run of an I/O automaton is an alternating sequence of states and events, $r = s_0 e_0 s_1 e_1 \ldots$, starting at q_0 and following the transition relation: for every $i > 0$, $\Delta(s_{i-1}, e_{i-1}, s_i)$. The trace of r is the sequence of events observed on r ($e_0 e_1 \ldots$). An event e is said to be enabled at state q if there is a transition $\Delta(q, e, q')$. For a fairness class $E \in \mathcal{P}$, r is E-fair if, for every event $e \in E$,

1. there are infinitely many states s_i such that e is not enabled at s_i; or,
2. e occurs infinitely often in r.

A run is fair if it is E-fair, for every E in $\mathcal{P}$. The language of an I/O Automaton $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$ is the set of words $\{\gamma \in \Sigma^\omega | \mathcal{A} \text{ has a fair run on } \gamma\}$.

Two I/O Automata can be composed, using a variation of the usual synchronous product of automata.

Definition 8 (Composition of I/O automata). *The composition of two automata $\mathcal{A}^1$ and $\mathcal{A}^2$ is defined if their output events are distinct ($\Sigma_o^1 \cap \Sigma_o^2 = \emptyset$). In that case, $\mathcal{A} = \mathcal{A}^1 \times \mathcal{A}^2$ is*

1. $Q = Q^1 \times Q^2$*;*
2. $q_0 = (q_0^1, q_0^2)$*;*
3. $\Sigma_i = (\Sigma_i^1 \setminus \Sigma_o^2) \cup (\Sigma_i^2 \setminus \Sigma_o^1)$ *i.e. only input events controlled by neither agents are input events of the composition;*

4. $\Sigma_o = \Sigma_o^1 \cup \Sigma_o^2$: we do not hide "local events", in order to ensure associativity;
5. $\Delta((q^1, q^2), e, (s^1, s^2))$ iff
  – $e \in \Sigma^1 \cap \Sigma^2$ and $\Delta^i(q^i, e, s^i)$, for $i = 1, 2$;
  – or, $e \in \Sigma^1 \setminus \Sigma^2$, $q^2 = s^2$ and $\Delta^1(q^1, e, s^1)$ or vice-versa.
6. $\mathcal{P} = \mathcal{P}^1 \cup \mathcal{P}^2$, i.e. we keep the original fairness conditions.

The composition operation enjoys the following properties:

Lemma 1. *For every I/O Automata $\mathcal{A}^1, \mathcal{A}^2, \mathcal{A}^3$, provided composition is defined, we have*

Associativity: $\mathcal{A}^1 \times (\mathcal{A}^2 \times \mathcal{A}^3) = (\mathcal{A}^1 \times \mathcal{A}^2) \times \mathcal{A}^3$.
Commutativity: $\mathcal{A}^1 \times \mathcal{A}^2 = \mathcal{A}^2 \times \mathcal{A}^1$.
Refinement (Trace inclusion): $\mathcal{L}(\mathcal{A}^1 \times \mathcal{A}^2) \subseteq \mathcal{L}(\mathcal{A}^1)$

Proof. Associativity and commutativity are shown in [14]. The former relies on the fact that $\mathcal{A}^1$ output events caught by $\mathcal{A}^2$ are not hidden. Trace inclusion comes from the fact that $\mathcal{A}^2$ cannot block an $\mathcal{A}^1$ transition in the composition, by input-enabledness (see def.7) . Therefore, fairness is preserved.

3.4 Relationships between Models

Usually, the meanings of inter- and intra-agent models overlap. Along the lines of [10], we take advantage of this redundancy by relating models in two ways:

Model checking: given a uLSC model $\mathcal{S}$ and a state-based model associating an I/O automaton to every *system* agent $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, we check that the composed system fulfills $\mathcal{S}$: $\mathcal{L}\left(\prod_{i=1}^n \mathcal{A}_i\right) \subseteq \mathcal{L}\left(\mathcal{S}\right)$
Synthesis: given a uLSC model $\mathcal{S}$, we verify that it is possible to find one automaton per system agent such that their composition is a correct implementation of $\mathcal{S}$.

4 Previous Answers

There has already been much research on these two issues. However, the proposed solutions suffer performance problems.

4.1 Model Checking

LSCs can be translated to temporal logics [17, 18] and fed into a model checker. LSCs can be translated to LTL, or CTL, or even ACTL [19], or Büchi automata [20].

The formula obtained from a uLSC is

$$\Box \bigwedge_{w \in \ \mathrm{lin}(P)} (\phi_w \implies \bigvee_{v \in \ \mathrm{lin}(P) \cdot \mathrm{lin}(M)} \phi_v),$$

where $\phi_\epsilon = \top$, $\phi_{a \cdot u} = (a \wedge \bigcirc(N \ \mathcal{U} \phi_u))$, and $N = \neg e_1 \wedge \ldots \wedge \neg e_n$, where $\Sigma_R = \{e_1, \ldots, e_n\}$.

However, these approaches face two obstacles:

1. The formula presented above is exponential in the size of the specification.
2. Computing the product of the numerous I/O Automata composing the system can lead to the state explosion problem.

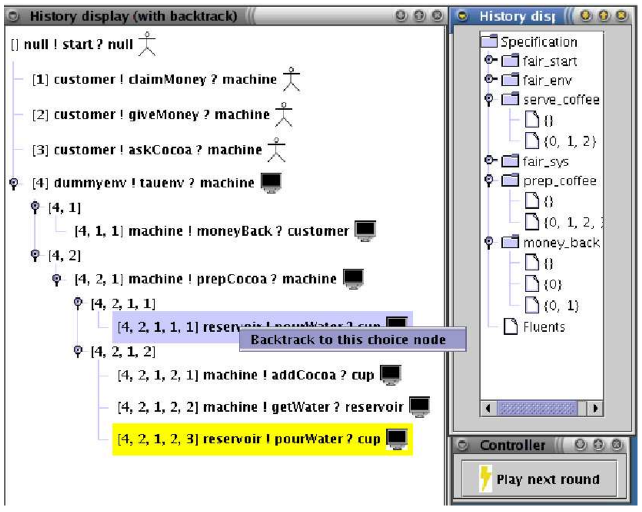
4.2 Synthesis (aka Realizability Checking)

We have implemented the exponential time algorithm presented in [13]. This program, called a *Realizability Checker* [21], builds a correct implementation. This state machine is often exponential in size, and hardly readable by users.

We use the explanation power of animation [22] to illustrate the flaws found by the realizability checker. If the specification is realizable, an implementation is built. Then, the analyst plays the role of the user, i.e. triggers environment events. The animator simulates the reaction of the system, according to the synthesized strategy. If the implementation is not behaving as expected, the model can be adjusted and synthesized again.

If the specification is *not* realizable, the algorithm builds a *sabotage plan* for the environment. Controlling environment events, it will lead *every* system implementation to failure. The roles are thus reversed during animation: the analyst plays the role of the future system whereas the animator plays an evil environment, following the sabotage plan. The analyst will always be driven to a state where he will have to violate some constraint. The animator announces conflicting constraints, such as "scenario X requires event *e* but this event is forbidden by scenario Y". The analyst can backtrack in the execution and try alternatives. Fig.5 presents a screenshot of (part of) the animator.

Fig. 5. Realizability checker/Animator screenshot


History display (with backtrack)
History disp
Backtrack to this choice node
Controller
Play next round

```
[] null ! start ? null
  [1] customer ! claimMoney ? machine
  [2] customer ! giveMoney ? machine
  [3] customer ! askCocoa ? machine
  [4] dummyenv ! tauenv ? machine
    [4, 1]
      [4, 1, 1] machine ! moneyBack ? customer
    [4, 2]
      [4, 2, 1] machine ! prepCocoa ? machine
        [4, 2, 1, 1]
          [4, 2, 1, 1, 1] reserv
        [4, 2, 1, 2]
          [4, 2, 1, 2, 1] machine ! addCocoa ? cup
          [4, 2, 1, 2, 2] machine ! getWater ? reservoir
          [4, 2, 1, 2, 3] reservoir ! pourWater ? cup
```

```
Specification
  fair_start
  fair_env
  serve_coffee
    {}
    {0, 1, 2}
  fair_sys
  prep_coffee
    {}
    {0, 1, 2,
  money_back
    {}
    {0}
    {0, 1}
  Fluents
```

The synthesis algorithm described above depends crucially on the *perfect information hypothesis*: every agent can sense every event. This ignores the interface description from the structural model, which explicitly defines which events are visible to each agent. It is more interesting to synthesize a distributed system in which every agent only listens to events specified in its interface. However, telling whether such an implementation exists is undecidable [21].

5 A Lightweight Approach

Since the problem is undecidable, we propose lose completeness but keep soundness: Every implementation produced is correct, but our algorithm may fail to find some implementations.

5.1 Model Checking

First, we suggest to use the techniques for minimizing the size of the formulae generated from uLSCs devised in [18, 17]. Typically, a uLSC can be split into several small formulae, in which we only need to check the proper ordering of *pairs* of events, and not all linearizations (as in sec. 4.1).

In order to address the state explosion problem, we suggest to ignore all components that do not participate in the verified uLSC. Suppose that only agents i through k participate in S. Hence, it is sufficient to check that the uLSC is correct, wrt the subsystem composed of agents i through k only. Since we demonstrated that I/O automata composition is a refinement, proving that the uLSC is satisfied by this reduced system is enough to show that the global system is correct, too:

$$\mathcal{L}\left(\prod_{j=1}^{n} \mathcal{A}_j\right) \subseteq \mathcal{L}\left(\prod_{j=i}^{k} \mathcal{A}_j\right) \subseteq \mathcal{L}(S).$$

Furthermore, when the subsystem can satisfy the LSC on its own, it indicates that the design achieves low coupling: the fulfillment of the property does not depend on components which are not directly involved in it.

However, if model checking fails, it might be a false negative: the counter-example could have been avoided, had we included more agents in the system, which one can try.

5.2 Synthesis

Our lightweight algorithm is illustrated in fig. 6. Its steps are detailed in the rest of this section.

(1) Agent Selection As opposed to the previous algorithm, the lightweight algorithm focuses on a single agent at a time. It does not try to find a strategy for all participants in one run. For the rest of this section, let a be the selected agent.

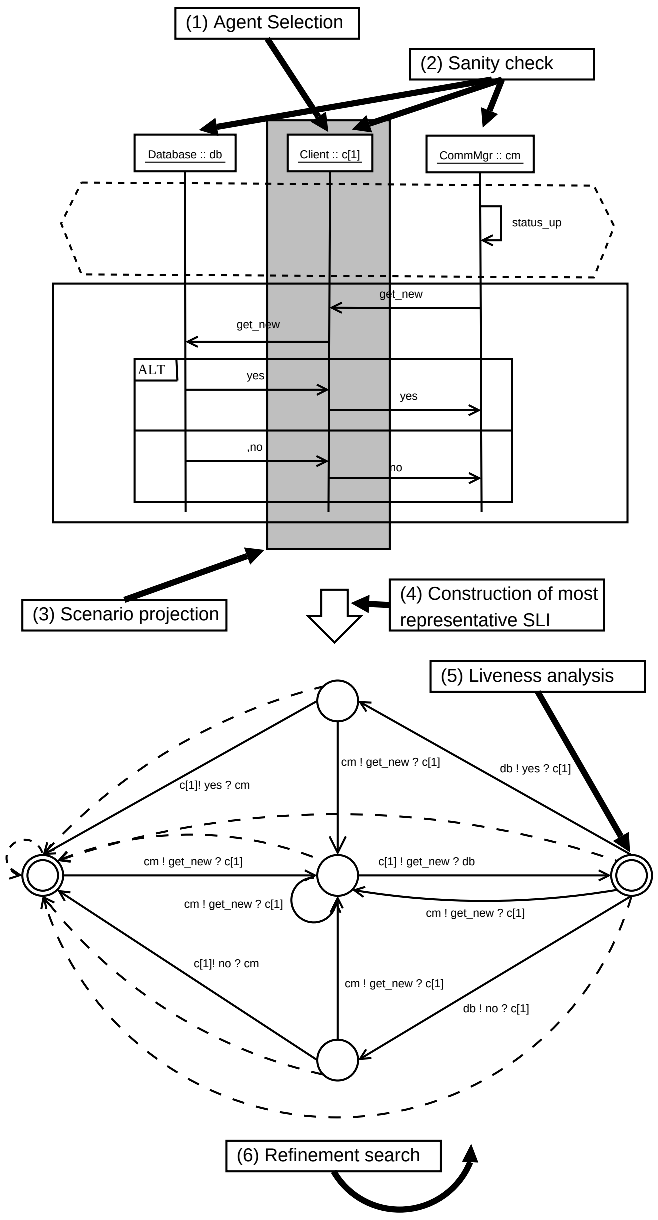


Fig. 6. Standard Local Implementation (SLI) for `c[1]`

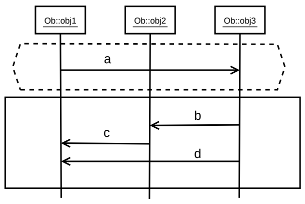


Fig. 7. Mismatch between causal order and visual order

(2) Sanity check All scenarios in which a participates actively are checked to ensure that their causal order matches their visual order. Two locations are causally related if they are sending locations on the same lifeline or if they are the sending and receiving locations of the same event. This is done in polynomial time [23]. Fig.7 gives an example of a uLSC which does not fulfill this condition. Clearly, it is not simply distributable for agent `obj3`, because `d` may only be sent after `c` has occurred, which `obj3` cannot see.

If this sanity check fails, the algorithm stops and explains why specification is not distributable.

(3) Scenario Projection All scenarios are projected onto the lifeline of agent a (e.g., the upper part of fig.6 illustrates an attempt to synthesize an implementation for `c[1]`). All uLSCs in which a is not required to perform any event are discarded. For instance, the scenario of fig 4(a) would be discarded because `c[1]` does not take part in it. In summary, Step 3 produces a set of non-empty uLSCs, reduced to the lifeline of a, one for each uLSC in which at least one event controlled by a is restricted.

(4) Construction of Most Representative SLI The I/O automaton built is input-enabled. It records in I all possible cuts of every scenario. The invariant of the automaton is: for every word w, if the automaton reads w and ends up in a state I then, for every ideal c, $c \in I$ iff some suffix of $w|_{\Sigma_R}$ linearizes c. For instance, in Fig.6, the center state of the I/O automaton records the configuration where the last event was `get_new` (from `cm` to `c[1]`), as all its incoming transitions indicate. This means that the prechart of the projected scenario has been matched and `get_new` (from `c[1]` to `db`) is now required from agent `c[1]`. Since this event is not forbidden at that state, the *Standard Local Implementation (SLI)* rule (see below) allows it to be scheduled.

Definition 9 (Standard Local Implementation (SLI)). *Let the projected specification be composed of m non-empty uLSCs: $\{S_1, \ldots, S_m\}$. An I/O automaton fulfilling the following constraints is called a* Standard Local Implementation (SLI)*:*

$$\langle \Sigma^r_a, \Sigma^s_a, Q, q_0, \Delta, \{\Sigma^r_a\}\rangle$$

where

– $Q = \prod_{i=1}^{m} 2^{2^{L_i x}}$, *i.e. every state keeps one configuration per uLSC, a configuration being a set of ideals.*
– $q_0 = (\{\emptyset\}, \ldots, \{\emptyset\})$,
– $\Delta((I_1, \ldots, I_m), e, (I'_1, \ldots, I'_m))$ *implies both the following statements*
  1. *Δ follows the ideals transition system:*
     - *if $e \notin \Sigma^i_R$, $I'_i = I_i$;*
     - *if $e \in \Sigma^i_R$, $I'_i = \{c' | \exists c \in I_i : c \xrightarrow{e} c'\} \cup \{\emptyset\}$. The empty ideal is always added, because it is linearized by the empty word, which is a suffix of every word $w \in \Sigma^*$, thus preserving the invariant.*
  2. *If $e \in \Sigma^s_a$, there is some i such that $c \in I_i$ requires e and, for every j, there is no $c \in I_j$ forbidding e.*

Such an implementation is called "standard" because it follows the classical way of extracting state machines from MSCs (see sec.6). It is dubbed "local" because it only considers a single agent, restricting a scenario to the local view of that agent.

Note that there may exist many SLIs for a given specification, because the condition on Δ is only an implication. They differ only in the scheduling of Σ^s_a events. Thus, it is possible to order SLIs: an SLI $\mathcal{A}$ is *more general* than an SLI $\mathcal{A}'$ ($\mathcal{A}' \sqsubset \mathcal{A}$) iff, at every state q, if $\mathcal{A}'$ allows $e \in \Sigma^s_a$ event, then $\mathcal{A}$ allows e, too.

(5) Liveness Analysis The I/O automaton built according to the SLI rule is always safe, because the forbidden events may not be scheduled. To show this, we prove that the hypotheses made by a about the global state are valid:

Lemma 2 (SLIs are sound). *Let $\mathcal{A}$ be an SLI. Consider a finite run $w \in \Sigma^*$, decomposed in two parts $uv = w$ and a scenario L_j. If $v|_{\Sigma_R}$ linearizes some ideal c in L_j and $\mathcal{A}$ has a run on $v|_{\Sigma_a}$ leading to a state $(I_1, \ldots, I_j, \ldots, I_n)$, then I_j contains $c|_{\Sigma_a}$.*

Proof. By induction on w.

Lemma 3 (SLIs are safe). *All behaviours induced by an SLI are Σ^s_a-safe.*

Since SLIs guarantee Σ^s_a-safety, it suffices to ensure that the considered automaton is Σ^s_a-live to verify that it is a correct implementation.

Theorem 2. *Let $\mathcal{A}$ be an SLI. If all runs in $\mathcal{A}$ are Σ^s_a-live, then $\mathcal{A}$ is a correct implementation of a system consisting of agent a.*

Proof. By definition 6, if $\mathcal{A}$ is Σ^s_a-live (assumption) and Σ^s_a-safe (lem.3), it is a correct implementation.

In general, liveness is not true of all SLIs, because some required event might be postponed forever, since it is always unsafe. The liveness condition needs to be algorithmically checked; this is done in time quadratic in $|\mathcal{A}|$: $\mathcal{A}$ is analyzed to check that, on all fair infinite paths, there are infinitely many occurrences of e or e is not required in infinitely many states, for every $e \in \Sigma^s_a$. In fig. 6, the states in which no event is required are drawn with a double line. This SLI example is live for agent `c[1]`.

(6) Refinement Search If $\mathcal{A}_\top$ is not a correct implementation, i.e. it is not live, we can try to find another SLI $\mathcal{A}$ such that $\mathcal{A} \sqsubset \mathcal{A}_\top$ and $\mathcal{A}$ is live. In order to do so, we consider refinement as a two-person game, between a "protagonist" and an "antagonist". The protagonist may remove some edges labeled by Σ_a^s events while the antagonist tries to prove that the resulting automaton is still unlive. If the protagonist has no winning strategy in this game, there is no live SLI for agent a. This game can be solved using classical algorithms, in time polynomial in the size of the graph [24].

Remark 1 (Safety Assumptions). An SLI allows agents to make safety assumptions about their environment, which makes compositional reasoning feasible [25]. For instance, when synthesizing agent i, we can make use of the fact that we know *beforehand* that agents $1, \ldots, k$ will also be synthesized using the same method. In that case, when agent i receives an event from another "to-be-synthesized" agent j, he knows that some ideals of the configuration are not valid anymore. Indeed, if agent j sends this message, he must be required to do so. Now, if there is only one scenario which requires him to send j, the agent we are synthesizing can deduce the exact position in this scenario. For synthesizing the SLI of fig.6, our algorithm used this assumption.

Remark 2 (Efficiency). By construction, the I/O Automaton built here is *necessarily* smaller than the automaton constructed in [13]. This justifies our claim that this localized technique can sometimes be more efficient than the exact centralized one. However, in the worst case, the SLI is as big as the solution for the centralized case (and thus, exponential in the size of the specification [26]).

Our running example is specified with 25 scenarios and contains 8 components. Our implementation of the centralized synthesis algorithm fails to analyze it, because of its size. However, the implementation of the lightweight algorithm successfully synthesizes an SLI for every component, but `cm` and `db` (see the next remark). `cm` cannot be synthesized because it participates in all scenarios; projecting the specification on it does not drastically reduce the size of the specification. The SLIs that we obtained had less than 20 states each and their synthesis took only a couple of seconds. This synthesis relied on the additional safety assumptions explained above.

Remark 3 (A Bad Case). The following specification cannot be implemented by any SLI. Assume that we have two scenarios for `db`, asserting that it must answer either `db ! yes ? c[i]` or `db ! no ? c[i]` to all queries of client `c[i]`. There is no SLI implementing this requirement for `db`, because the agent must remember the last request that it replied to. Otherwise, the system runs into starvation. Consider the following execution: `c[1]` and `c[2]` query `db`. This leads to some state q. In this state, `db` answers to `c[1]`. Immediately after, `c[1]` queries `db` again, going back to q. Since `db` has no means to remember that it replied to `c[1]` before, it replies to `c[1]` again. Thus, we enter a loop in which `c[2]` will never get a reply. Allowing `db` to use some fixed amount of additional memory, here one bit, could help in avoiding this situation.

6 Related Work

6.1 Play-out

The play-out approach [27, 28] is related to ours. The play-out algorithm works as follows. A state of the system is a set of ideals, called *live copies*. The user generates an environment event at a time. Assume that this event is e and the current state is $\{c_1, \ldots, c_n\}$. The following rules apply:

1. For every i $(1 \leq i \leq n)$, if $c_i \xrightarrow{e} c'_i$, then c_i is replaced by c'_i. Otherwise, if c_i is in the prechart, it is dropped.
2. If e labels a minimal location of some scenario S_j, a new live copy for S_j is spawn. Thus, $\{l\}$ is added to the next state, where l is the first location in S_j labeled by e.

If, in the next state, some system events are required but not forbidden, one of them is picked and performed. This updates the state, which, in turn, can trigger new events. We followed a similar scheme for designing the SLI rule: an agent will only schedule e if e is required and not forbidden. However, we use the global view of the behaviour to avoid being trapped in unsatisfiable states.

6.2 Synthesis from MSCs

Conceptually, the synthesis algorithms of [29–31] are very close to ours, except that they apply to MSCs. For every agent, a state machine is built, which tracks its current position on its lifeline. When it reaches a position in which the MSC dictates to send an event e, the machine proposes e. For MSCs, this procedure yields a distributed implementation encompassing all the behaviours specified by the MSCs. Nevertheless, it is possible that this distributed implementation is not correct. This happens when the implementation allows more behaviours than specified by the MSCs. These additional behaviours are called *implied scenarios*. Much work has been devoted to detecting and reporting on those implied scenarios [29, 32, 30, 33]. The picture is slightly different in our case. Our SLIs do not necessarily encompass all the behaviours of the scenario-based specification. Indeed, in step 5 of our procedure, we detect liveness violations, that is *missing* scenarios.

The problem of component-based proofs from MSCs is investigated in [34], where causal MSCs are identified.

References

1. Object Management Group (UML Revision Task Force): OMG UML Specification (2.0). (2003) `http://www.omg.org/uml`.
2. Jacobson, I.: Object Oriented Software Engineering: a Use-Case Driven Approach. ACM Press/Addison-Wesley (1992)
3. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design **19** (2001) 45–80

4. : MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC) (2000) `http://www.itu.int/`.
5. Harel, D.: From play-in scenarios to code : An achievable dream. IEEE Computer **34** (2001) 53–60 a previous version appeared in Proc. of FASE'00, LNCS(1783), Springer-Verlag.
6. Jackson, D.: Automating first-order relational logic. In: Proc. ACM SIGSOFT Conf. Foundations of Software Engineering., San Diego (2000)
7. Bontemps, Y., Heymans, P., Kugler, H.: Applying LSCs to the specification of an air traffic control system. In Uchitel, S., Bordeleau, F., eds.: Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools" (SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03), Portland, OR, USA, IEEE (2003) available at `http://www.info.fundp.ac.be/~ybo`.
8. Whittle, J., Schumann, J.: Statechart Synthesis from Scenarios: an Air Traffic Control Case Study. In: Proc. of "Scenarios and State-Machines: models, algorithms and tools" workshop at the 24th Int. Conf. on Software Engineering (ICSE 2002), Orlando, FL, ACM (2002) `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.
9. Harel, D.: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming **8** (1987) 231–274
10. Harel, D.: From play-in scenarios to code: An achievable dream. IEEE Computer **34** (2001) 53–60
11. Bontemps, Y.: Realizability of scenario-based specifications. Diplôme d'études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), rue Grandgagnage, 21, B5000 - Namur (Belgium) (2003)
12. Cobben, J., Engels, A., Mauw, S., Reniers, A., M.: Formal Semantics of Message Sequence Charts (ITU-T Recommendation Z.120 Annex B). International Telecommunication Union, Eindhoven, The Netherlands. (1998) `http://www.itu.int`.
13. Bontemps, Y., Schobbens, P.Y.: Synthesizing open reactive systems from scenario-based specifications. In Balarin, F., Lilius, J., eds.: Proc. of the 3rd Int. Conf. on Applications of Concurrency to System Design (ACSD'03), Guimarães, Portugal, IEEE Computer Science Press (2003) 41–50
14. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2** (1989) 219–246
15. Broy, M.: Unifying models and engineering theories of composed software systems. In Broy, M., Pizka, M., eds.: Models, Algebras and Logics of Engineering Software. Volume 191 of NATO Science Series, III: Computer and Systems Sciences. IOS Press (2003) 1–41 ISBN 1-58603-342-5.
16. Peled, D.A., Clarke, E.M., Grumberg, O.: Model Checking. MIT Press, Cambridge, Massachusetts (2000) ISBN 02-620327-08.
17. Kugler, H., Harel, D., Pnueli, A., Yuan, L., Bontemps, Y.: Temporal Logic for Live Sequence Charts. Unpublished draft (2001)
18. Bontemps, Y.: Automated Verification of State-based Specifications Against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications). Master's thesis, University of Namur, rue Grandgagnage, 21 - 5000 Namur(Belgium) (2001)
19. Emerson, E.A.: 16. In: Temporal and Modal Logic. Volume B. MIT Press and Elsevier Science Publishers, Cambridge, Massachusetts (1990) 997–1072 ISBN 0-262-72015-9 (Second Printing, 1998).
20. Klose, J., Wittke, H.: An Automata Based Interpretation of Live Sequence Charts. In Margaria, T., Yi, W., eds.: Proc. of TACAS (Tools and Algorithms for the

Construction and Analysis of Systems) 2001. Volume 2031 of LNCS., Genova, Italy, Springer-Verlag (2001) 512

21. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel (1992)
22. Heymans, P.: Animating Albert II Specifications. PhD thesis, University of Namur (2001)
23. Alur, R., Holzmann, G.J., Peled, D.: An analyser for mesage sequence charts. In Margaria, T., Steffen, B., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1055 of Lecture Notes in Computer Science., Springer-Verlag (1996) 35–48
24. Grädel, E., Thomas, W., Wilke, T., eds.: Automata Logics, and Infinite Games: A Guide to Current Research. Volume 2500 of Lect. Notes in Comp. Sci. Springer (2002) ISBN 3-540-00388-6.
25. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems **14** (1992) 1–60
26. Bontemps, Y.: Optimal algorithms for the synthesis of state-machines from live sequence charts. Technical report, Institut d'Informatique, University of Namur, Namur, Belgium (2003) Contact the author to obtain a copy of this paper.
27. Harel, D., Marelly, R.: Come, let's play! Scenario-based programming using LSCs and the Play-engine. Springer (2003) ISBN 3-540-00787-3.
28. Harel, D., Marelly, R.: Capturing and Analyzing Behavioral Requirements: The Play-In/Play-Out Approach. Technical Report MCS01-15, The Weizmann Institute of Science, Faculty of Mathematics and Computer Science, Rehovot, Israel (2001)
29. Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. In: Proceedings of 22nd International Conference on Software Engineering. (2000) 304–313
30. Uchitel, S.: Elaboration of Behaviour Models and Scenario-based Specifications using Implied Scenarios. PhD thesis, Imperial College London (2003)
31. Krüger, I.H.: Distributed System Design with Message Sequence Charts. PhD thesis, Technischen Universität München (2000)
32. Uchitel, S., Kramer, J., Magee, J.: Detecting implied scenarios in message sequence chart specifications. In Gruhn, V., ed.: Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01). Volume 26, 5 of SOFTWARE ENGINEERING NOTES., New York, ACM Press (2001) 74–82
33. Ben-Abdallah, H., Leue, S.: Syntactic Detection of Process Divergence and Nonlocal Choice in Message Sequence Charrts. In Brinksma, E., ed.: Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS'97. Number 1217 in LNCS, Enschede, The Netherlands, Springer-Verlag (1997) 259–274
34. Finkbeiner, B., Krüger, I.H.: Using message sequence charts for component-based formal verification. In: Proc. of OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems, Tampa Bay, FL, USA (2001)