

# Centre Fédéré en Vérification

Technical Report number 2002.39

## Turning High-Level Live Sequence Charts into Automata

Yves Bontemps, Patrick Heymans



This work was partially supported by a FRFC grant: 2.4530.02

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

# Turning High-Level Live Sequence Charts into Automata

Yves Bontemps<sup>\*</sup>  
Univ. of Namur - CS Dept  
rue Grandgagnage, 21  
B-5000 - Namur  
Belgium  
ybo@info.fundp.ac.be

Patrick Heymans  
CETIC and Univ. of Namur - CS Dept  
rue Clément Ader, 8  
B-6041 - Gosselies  
Belgium  
phe@info.fundp.ac.be

## ABSTRACT

Message Sequence Charts (MSCs) are a widely used scenario notation. However, we believe that this language lacks message abstraction and the ability to express whether a scenario is an example or a universal rule. Live Sequence Charts (LSCs) improve MSCs by solving these two shortcomings. However, LSCs have no formally defined high-level structuring mechanism. We extend LSCs with composition operators and define their semantics in terms of  $\omega$ -regular traces. We build Büchi automata from these scenarios. We show that standard algorithms on this (low-level) formalism can be used to check consistency and refinement, and to synthesize a state-based specification from a set of consistent requirements.

## 1. INTRODUCTION

Message Sequence Charts are widely used to describe scenarios [19, 2]. They are an easy to understand notation, highlighting actor interactions. Moreover, the MSC standard [15] provides a structuring language, High-Level Message Sequence Charts (HMSC), that makes it possible to compose scenarios through sequence, iteration, concurrency or choice.

Nevertheless, MSCs suffer two serious shortcomings: (1) they do not allow *message abstraction* and (2) they do not explicitly mention the *status* of each scenario.

Message abstraction makes it possible to state that messages *not* appearing in the chart are irrelevant, with respect to the scenario described. They can thus occur during the scenario execution without influencing it. In the scenario of Fig. 1, the user may issue two requests to a remote controller for a central heating system: on and off. Now, assuming that there is a clock built in this remote controller, this scenario

<sup>\*</sup>Research Fellow of the FNRS (Belgian National Fund for Scientific Research)

remains silent on the behaviour of the system when the user asks to display the current time, but it does not disallow these requests. They are simply abstracted away, because they are not relevant to the scenario considered.

The status of the behaviour described by the MSC is also unclear: is it a simple example (the system may sometimes behave like that) or is it a universal rule (the system shall always behave as specified, provided a given condition is met)?

The ability to abstract away irrelevant messages and to specify the status of each scenario is a characteristic of Live Sequence Charts (LSC), a variant of MSCs defined by Harel and Damm in [7]. We believe that LSCs are a good candidate for circumscribing the shortcomings of MSCs. However, as far as we know, LSCs still miss a formally defined high-level structuring mechanism similar to HMSCs.

In Section 2, we provide a taste of our language's constructs including high-level scenario composition operators as well as some lower-level improvements. How we give this language a compositional formal semantics in terms of traces is sketched there too. Section 3 describes how surprisingly simple are the automata recognizing such traces. Based on the good properties of these automata, Section 4 explains how standard algorithms can be used to check the consistency and refinement of LSCs as well as to synthesize state-based specifications from a set of consistent LSCs. The benefits and limits of the approach are discussed in Section 5 where we also give an outlook towards future work.

## 2. HIGH-LEVEL LIVE SEQUENCE CHARTS

Our language comes in three layers: Basic Charts (BCs), Iterative Charts (ICs) and Live Sequence Charts (LSCs).

### 2.1 Basic Charts

At the lowest-level is the language of Basic Charts (BCs). It is an enriched flavour of Message Sequence Charts or UML Sequence Diagrams. It consists of vertical lines, called *instance lines*, along which time runs from top to bottom. Along these lines, the points of interest are called *locations*. Every location is labelled, either with an *event* (i.e. send/receive a message) or with a *condition* (i.e. a propositional formula). Messages are depicted as arrows, flowing from one location to another one.

Locations belonging to the same instance line can be grouped in *coregions*. In that case, they can be reached in any order. As in the ITU standard [15], we allow the analyst to

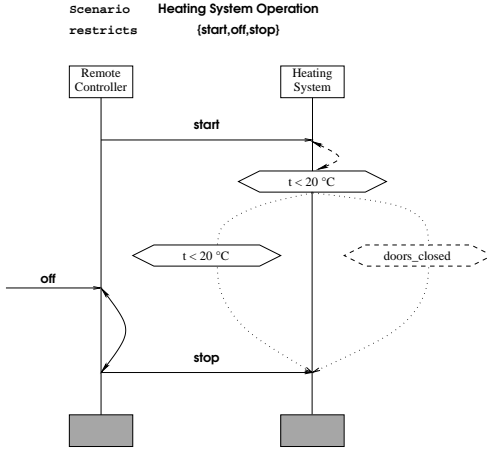


Figure 1: A sample basic chart

impose some ordering constraints between events belonging to the same coregion.

Generalizing a construct introduced by [16], we allow to express that two events may (or must) occur simultaneously. This construct is named *simultaneous region*.

A further contribution of our language is the addition of invariants, i.e. properties that should hold continuously between two points in time.

In the spirit of [7], almost all the above constructs come in two flavours: hot and cold. Violating a hot invariant results in an irrecoverable error, while violating a cold invariant causes a normal, though premature, exit from the chart. Regarding simultaneous regions, a hot region *imposes* that its composing events occur together while a cold region *enables* two events to happen at the same moment. If a location is cold, the execution does not have to reach the next location. A cold message must be sent but must not necessarily be received. However, this may depend on the temperature of its location.

Fig. 1 provides a sample Basic Chart for a room temperature control system. The system is turned on. After a while, it must reach a temperature lower than 20 degrees. This is modelled by a hot condition, an hexagonal plain line box. This condition may be verified as soon as the start order is received, which is depicted by a cold simultaneous region (a double-headed dashed arrow). From then on, it keeps the temperature below 20 degrees, until the system is shut down. This is represented by a hot invariant, a plain line hexagonal box labelling an arrow, which marks the scope of this invariant. If a door is open when the system is working, it leaves the chart and, in particular, does not have to guarantee that the room temperature stays below 20 degrees. A cold invariant is used to model this. When the user turns the system off, the order to stop the control is immediately sent to the heating system. We use a hot simultaneous region to represent this.

Every BC comes together with a set of events that are said to be *restricted in it*. This set includes (1) all the send and receive event of the messages appearing in the BC and (2) possibly also other events that have to be stated explicitly. While the scenario is executing, restricted events of the first kind can only take place as prescribed by the BC whereas events of the second kind cannot take place at all. Every

event that is *not restricted* does not influence the scenario and is thus *abstracted away*: it can take place at any time during the scenario execution.

Basically, the semantics of a BC is, as for a MSC, a strict partial temporal ordering of locations:  $a < b$  iff  $a$  should be reached before  $b$  is. Events belonging to the same instance line are ordered from top to bottom and the send event of a message must precede its receive event.

We let a run of a BC be a finite sequence

$$v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} v_2 \dots v_{n-1} \xrightarrow{e_n} v_n$$

where  $v$ , with subscripts, are *states* (i.e. valuations of propositions) and  $e$  are *sets of events*. A transition from  $v_{i-1}$  to  $v_i$  on events  $e_i$  means that, after staying in state  $v_{i-1}$  for some time, the system performed events  $e_i$  and ended up in state  $v_i$ .

The initial valuation  $v_0$  should fulfill the conditions attached to the initial locations of every instance and the invariants that start immediately. For every instance, the lowest location reached at the end of the execution must be cold, unless a cold invariant has been violated at some point of the run, in which case arbitrary behaviour is allowed from that point on.

The conditions for the run to reach a set of locations  $L$  in a step  $v_{i-1} \xrightarrow{e_i} v_i$  are

- for every location  $l \in L$ , all its prerequisite locations ( $\{\rho | \rho < l\}$ ) have been reached;
- for every location in  $L$ , all its simultaneous locations are also in  $L$ ;
- the invariants active in  $L$  hold in  $v_i$ ;
- the conditions of  $L$  hold in  $v_i$ ;
- the events of  $L$  are exactly the restricted events in  $e_i$ .

Note that, through the message abstraction mechanism, a run can contain steps that are irrelevant with respect to the specified scenario. In such a step, events that are *not restricted* by the chart may occur and proposition values may change, provided that the active invariants still hold.

## 2.2 Iterative Charts

The language of iterative charts (ICs) is based upon BCs. It provides operators to articulate scenarios and build complex scenarios from simpler ones. It is similar to High-Level MSCs [15]. We have made available the following composition operators:

- conditional branching ( $\text{IF}(\phi, IC_1, IC_2)$ );
- strong sequential composition ( $IC_1 ; IC_2$ );
- parallel synchronous composition ( $IC_1 || IC_2$ );
- nondeterministic choice ( $IC_1 \oplus IC_2$ );
- finite iteration ( $IC_1^*$ ).

The companion technical report [5] gives a compositional semantics to each of these constructs, in terms of languages of runs. We also give a mapping from a scenario to an automaton that recognizes precisely the language (of runs) corresponding to this scenario. We prove that this mapping is correct. In spirit, this construction is close to that of [9]

except that we use plain flattened automata instead of structured statecharts, for it makes the correctness proofs easier and enables us to use standard algorithms for verification and synthesis.

In the remainder, we let  $\mathcal{L}(IC)$  be the language of an iterative chart, i.e. the set of all traces that it allows.

The operators above deserve some explanation. First of all, our conditional branching is *state-based*. The condition  $\phi$  is a propositional formula. If the initial state of the run fulfills  $\phi$ , the first scenario is launched, otherwise, the second scenario applies. Secondly, our sequence operator is a *strong* one.  $IC_2$  may begin only *after*  $IC_1$  has been fully completed. The ITU standard prescribes a *weak* sequential composition, which means that the two scenarios are simply “glued” together, instance per instance. It has been shown that this semantical choice results in undecidability of fundamental properties, such as consistency or refinement [1, 14]. Thirdly, the parallel composition of two scenarios is synchronous. We identify identical messages from the two scenarios. For instance, in the scenario of figure 2, the two *failure* messages denote the same instance of the message in a run, as it is the case for the two *back\_to\_life* messages. This example shows how one can use the message abstraction mechanism of BCs together with parallel composition, to combine different views on an overall behaviour. Indeed, the “Failure Logging” scenario does not restrict *page* and *repair* messages, which means that they may occur arbitrarily. The occurrence of these two messages are constrained by the second scenario “Failure Fixing”, in which messages *write(off)* and *write(on)* are considered as irrelevant.

Finally, nondeterministic choice can be seen as a generalization of the notion of (cold) sub-chart introduced in [7]. A cold sub-chart is an optional part of a scenario. We can easily represent it in our language by  $IC \oplus \epsilon$  where  $IC$  is the content of the cold sub-chart and  $\epsilon$  is the empty scenario.

### 2.3 Live Sequence Charts

Our highest-level layer provides a means to specify the status of the described scenarios. Every scenario can be either *existential* or *universal*.

An existential scenario, written  $\diamond IC_1$ , is an example of execution. A system is said to satisfy it if, along one of its runs, it eventually reaches a state from which it successfully behaves as indicated in  $IC_1$ .

The semantics of this type of scenarios is easily given in terms of languages:

$$\mathcal{L}(\diamond IC_1) = \Sigma^* \cdot \mathcal{L}(IC_1) \cdot \Sigma^\omega \quad (1)$$

where  $\Sigma^*$  (resp.  $\Sigma^\omega$ ) denotes the set of all finite (resp. infinite) behaviours. (Note that we switch from finite regular languages to  $\omega$ -regular languages, because we describe reactive systems whose runs are typically infinite.)

Universal scenarios, written  $\square IC_1, IC_2^1$ , denote behaviours which should always be satisfied by the system. This means that, in every run, whenever the system exhibits the behaviour of scenario  $IC_1$ , it shall behave as prescribed by  $IC_2$  afterwards. The language defined by this scenario is

$$\mathcal{L}(IC_1) = \overline{\Sigma^* \cdot \mathcal{L}(IC_1) \cdot \mathcal{L}(IC_2) \cdot \Sigma^\omega} \quad (2)$$

We first take the set of all counter-examples of this scenario. A counter-example is a run in which, at some point,

<sup>1</sup>In the terms of [7],  $IC_1$  is called a pre-chart

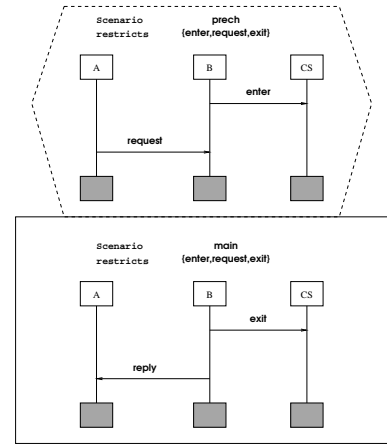


Figure 3: A sample LSC

the activation scenario  $IC_1$  is matched but, afterwards, the second scenario  $IC_2$  is not matched. We then complement this language and get the set of all runs that *never* violate  $IC_2$ . This approach was already followed in [8].

Fig. 3 presents a universal LSC. It is an excerpt of the scenario-based specification of a protocol to achieve mutual exclusion between two distributed processes (A,B), communicating by asynchronous messages [4]. This scenario states that, whenever B receives a request from A and B is already using the critical section (CS), B shall first stop using the CS and then, reply to A.

Finally, note that *no-scenarios* (also called *anti-scenarios*) can also be expressed with our language through expressions of the form  $\square IC, \perp$  where  $\perp$  denotes the chart that can never be fulfilled. Intuitively, such a no-scenario expresses that  $IC$  can never take place.

### 3. LSC AUTOMATA ARE (VERY) SIMPLE

In [5], we show that the automaton obtained from an LSC is very simple. First of all, an existential scenario has a language (eq. 1) of the form  $W \cdot \Sigma^\omega$ , where  $W$  is some regular language. These languages correspond to the class  $G$  which lies at the bottom of Borel’s hierarchy [20]. They can be recognized by deterministic weak automata.

A weak automaton is a Büchi automaton whose state-space  $Q$  can be partitioned into a finite number of disjoint classes  $Q_1, \dots, Q_n$ , such that, for every class, either all the states in it are accepting or all are rejecting. The  $Q_i$  can be partially ordered by  $\geq$  such that

$$\forall i, j. \exists q_i \in Q_i, q_j \in Q_j. q_j \in \delta(q_i) \implies Q_i \geq Q_j,$$

where  $\delta$  is the (possibly nondeterministic) transition relation of the automaton. Intuitively, every transition from a class should lead to the same class or to a “smaller” one. Thus, any infinite run ultimately gets trapped in a class whose states are either all accepting or rejecting. Weak automata have interesting properties: they can be seen both as Büchi and co-Büchi and a simple complementation procedure, called the “breakpoint construction” can be applied to them [3, 17].

Having deterministic weak automata associated to existential LSCs is a welcome result, since there exists an efficient minimization procedure for deterministic weak au-

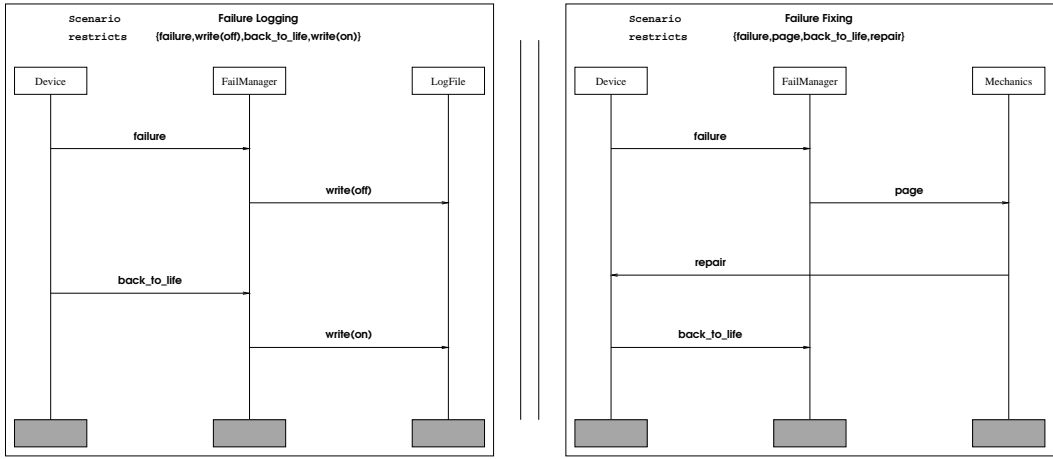


Figure 2: Iterative Chart: Parallel Composition of two BCs

tomata, which gives rise to a normal form, up to isomorphism [18].

The language of a universal chart suffers from a double complementation (eq. 2) which, at first sight, is likely to make it impossible to compute practically. However, we have shown that automata recognizing languages of the form  $W \cdot \Sigma^\omega$  are straightforward to complement and result in deterministic weak automata. The basic idea is that an infinite word  $w_0w_1 \dots$  belongs to  $\overline{W \cdot \Sigma^\omega}$ , iff every finite prefix of  $w_0w_1 \dots$  does not belong to  $W$ :

$$\forall i \geq 0. w_0w_1 \dots w_i \notin W$$

In terms of automata, we should simply ensure that we always avoid accepting states of the deterministic finite automaton recognizing  $W$ .

The append construction still preserves weakness and thus, the automaton recognizing the language of eq. 2 only has to be complemented using the “breakpoint construction”.

## 4. CONSISTENCY, SYNTHESIS AND REFINEMENT

We are now able to define the problems in which we are interested, namely: consistency checking, synthesis and refinement.

Let us assume that the requirements about the system under development are given in two parts: a set of universal LSCs ( $\mathcal{R}_\square$ ) and a set of existential LSCs ( $\mathcal{R}_\diamond$ ).

### 4.1 Consistency

As a first step, the analyst would like to check that the requirements are *consistent*. Inconsistent requirements are impossible to implement, because no system may ever fulfill them. We reduce the problem of consistency checking to checking nonemptiness of the intersection of  $\omega$ -regular languages. The requirements are said to be consistent iff

$$\forall L \in \mathcal{R}_\diamond. \bigcap_{L \in \mathcal{R}_\square} \mathcal{L}(L) \cap \mathcal{L}(L) \neq \emptyset$$

and

$$\bigcap_{L \in \mathcal{R}_\square} \mathcal{L}(L) \neq \emptyset \quad (3)$$

Since a Büchi automaton that recognizes the language of a given LSC can be built, one can apply the product construc-

tion to the resulting automata to compute their intersection and, afterwards, use the double depth-first search procedure to ensure that the intersection is nonempty.

### 4.2 Synthesis

If the requirements are consistent, one would like to automatically obtain an implementation that fulfills them.

In our framework, this is easily done, because we already transform LSCs into automata. Thus, assuming that the requirements are consistent, we simply output the automaton recognizing

$$\bigcap_{L \in \mathcal{R}_\square} \mathcal{L}(L) \quad (4)$$

Again, the standard algorithm to compute the intersection of Büchi automata can be applied to solve this problem.

Of course, there are many drawbacks in implementing immediately this definition. Firstly, computing the product of automata is computationally expensive.

Secondly, the Büchi automata formalism is rather low-level and it would surely be preferable to obtain a specification in a more structured and readable language, such as statecharts [10], for instance.

Thirdly, the resulting specification is a *Global System Automaton* (GSA) [12] and a “distributed” specification (one machine per instance) would certainly be more manageable. Two techniques could be used to perform this “distributed” synthesis. The first one summarizes as “compute the GSA and distribute it into the different instances”, as done in [12]. The second tries to directly generate one state machine per instance, by identifying which information is needed locally by each instance to correctly play the scenarios and then carefully projecting each scenario on every instance. We are currently investigating how this second technique could be applied to LSCs.

### 4.3 Refinement

Now, suppose that the analysts have been working on the specification for a little while. This specification has been cleaned up. Design choices have been made, by removing nondeterminism, for instance.

On their side, nontechnical stakeholders have also a more acute understanding of the future system. Their require-

ments have evolved, which means that new scenarios have been added to the requirements or existing scenarios have been modified.

Somehow, it would be nice to check, after such modifications have occurred, that the requirements  $\mathcal{R}$  are *still* consistent with the specification  $\mathcal{S}$  on which the analysts are working. Of course, one could delete the existing specification and synthesize a new one, with the procedure above. This approach would be wasteful, since it starts the specification from scratch every time the requirements evolve, even slightly.

Thus, a well-thought tool should provide a *verification procedure*, that automatically checks whether  $\mathcal{S}$  is consistent with  $\mathcal{R}$  and, if this is not the case, provides a counter-example.

The definition of “ $\mathcal{S}$  is consistent with  $\mathcal{R}$ ” follows ( $\mathcal{S}$  is assumed to be given as a Büchi automaton as well).

First of all, for every existential scenario in  $\mathcal{R}$ , there should be a run of  $\mathcal{S}$  that finally satisfies it. Formally,

$$\forall L \in \mathcal{R}_\diamond. \mathcal{L}(\mathcal{S}) \cap \mathcal{L}(L) \neq \emptyset. \quad (5)$$

Secondly, every trace of  $\mathcal{S}$  should satisfy every universal scenario.

$$\forall L \in \mathcal{R}_\square. \mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(L) \quad (6)$$

One can easily transform the previous expression into

$$\forall \square IC_1, IC_2 \in \mathcal{R}_\square. \mathcal{L}(\mathcal{S}) \cap \left( \Sigma^* \cdot \mathcal{L}(IC_1) \cdot \overline{\left( \mathcal{L}(IC_2) \cdot \Sigma^\omega \right)} \right) \neq \emptyset.$$

We thus need to complement  $\mathcal{L}(IC_2) \cdot \Sigma^\omega$ . Since this language is of the form  $W \cdot \Sigma^\omega$ , with  $W$  regular, as already argued, this complementation is easily achieved.

## 5. CONCLUSION AND FUTURE WORK

In our view, the main contribution of our work is the formal definition of a full-fledged version of LSCs including (1) high-level scenario composition constructs and (2) miscellaneous improvements such as generalized simultaneous regions and clearly defined conditions and invariants. The resulting LSC language has an improved expressiveness wrt [7]. Though, at the notational level, some questions are left open. Firstly, the scalability of HLSCs to specify large and complex systems still needs to be demonstrated. Some initial results can be found in [6] where the application of LSCs to specify hardware protocols results in a significant reduction of the number of scenarios wrt MSCs. This is mainly due to the message abstraction mechanism. We conjecture that the composition operators of HLSCs would be useful for improving the structuring of LSC specifications. The nondeterministic choice operator improves the expressiveness of LSCs and, together with the other improvements (see above), is felt to facilitate the specifier’s task. We intend to carry out real-world case studies in order to evaluate the relevance of these constructs. Secondly, there are currently some known limits to the expressiveness of HLSCs: most notably, (1) the fact that modal operators ( $\diamond$  and  $\square$ ) can only be used at the outermost level of HLSC expressions and (2) the strong sequential composition. The design of the HLSC notation resulted from a trade-off between usability, expressiveness and ease of computation. Clearly, the above two restrictions avoid impairing the computational aspects. Releasing (1) would lead us to use a more complex class of automata, namely, alternating automata, while releasing (2)

necessitates dealing with infinite state spaces which opens the way to undecidability. Therefore, either of these restrictions will be released only if absolutely necessary. Again, real-world case studies are needed before making such a decision. Finally, the usability and communicability of LSCs heavily relies on their syntactical simplicity, which is one of the main goals of scenario-based notations. We tried not to overload LSCs with new concepts but rather generalize the existing ones for the sake of formality and expressiveness. Of course, this is a subject of debate but we believe that an incremental, structured approach to specifying with LSCs can be useful here. In the spirit of [7] and [11], initial elicitation and communication with stakeholders can be done by using a simplified subset of HLSCs (e.g. using only existential scenarios and a subset of composition operators). Then, more complex constructs can be introduced to generalize the descriptions. To validate the generalized descriptions, an alternative to reviewing is play-out [13], a form of animation which can hide most of the notational complexity.

Regarding formal semantics, we provide a compositional one for HLSCs in terms of traces and a mapping from a set of scenarios to a state-based specification. In [5], we show that the automata yielded by this mapping are weak. Their complementation is easy to implement. The complementation algorithm should behave practically well [3]. However, this description of the system is given as a Büchi automaton obtained through to the complementation of a simpler automaton. Consequently, the specification is likely to be difficult to read, and thus, of little interest for the later stages of development. Finding a technique to generate a readable specification in a more user-friendly formalism (e.g. statecharts) is a problem that we are currently working on. We could hierarchically structure the automaton a posteriori, using, for instance, the approach of [22]. Note that, as argued in [21], a tool that automatically structures a state-based specification can be problematic, because design choices can be hidden in the software. Thus, special attention should be paid to this problem. Moreover, obtaining a distributed specification, i.e. a state machine per instance, is more interesting than building a global specification. This is another point of interest for future research.

If we regard the solution to the synthesis problem given here as of little interest, we believe that our approach to the consistency and refinement, even if rudimentary, is more interesting practically. In comparison with [12], although we treat a more expressive variant of LSCs, we are still able to decide consistency. In our previous work, we addressed the problem of refinement (verification) by translating scenarios to temporal logic [4]. Model checkers could then be used to perform verification. However, this technique was expensive, since it needed two translations to automata (one in the translation process and one in the model checker) to complete the proof. Here, only one automaton is needed. That aspect of our work is closer to [16].

## 6. ACKNOWLEDGEMENTS

Yves Bontemps is supported by the FNRS and the University of Namur. Patrick Heymans is supported by the EU and the Walloon Region through the structural funding programme Objective 1 (Phasing Out) as well as by the University of Namur.

The authors are grateful to the anonymous reviewers for their insightful comments, pointers to related work and di-

rections for future research. Prof. Pierre-Yves Schobbens is thanked for his collaboration and careful proof-reading.

## 7. REFERENCES

- [1] Rajeev Alur and Mihalis Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, number 1664 in LNCS, pages 114–129. Springer-Verlag, 1999.
- [2] Daniel Amyot and Armin Eberlein. An Evaluation of Scenario Notations for Telecommunication Systems Development. In *Proc. of 9th Int. Conference on Telecommunication Systems (ICTS)*, Dallas, USA, March 2001.
- [3] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables. In *Proc. International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, pages 611–625, Sienna, June 2001. Springer-Verlag.
- [4] Yves Bontemps. 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), June 2001.
- [5] Yves Bontemps and Patrick Heymans. Turning High-Level Live Sequence Charts into automata. Technical report, Univ. of Namur - Computer Science Dept, 21, rue Grandgagnage, 5000 - Namur (Belgium), March 2002. available at <http://www.info.fundp.ac.be/~ybo>.
- [6] Annette Bunker and Ganesh Gopalakrishnan. Using Live Sequence Charts for Hardware Protocol Specification and Compliance Verification. In *IEEE International High Level Design Validation and Test Workshop*. IEEE Computer Society Press, November 2001.
- [7] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [8] Martin Fränzle and Karsten Lüth. Visual Temporal Logic as a Rapid Prototyping Tool. *Computer Languages*, 27(1-3):93–113, January 2002. abridged version published in Proc. VFM'99 (First International Symposium on Visual Formal Methods), Nbr 99-08 in Computing Science Report, Eindhoven University of Technology.
- [9] Martin Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *Proceedings of ESEC'95 - 5th European Software Engineering Using Scenarios*, pages 254–271, Berlin, 1995. Springer-Verlag.
- [10] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [11] David Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.
- [12] David Harel and Hillel Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, February 2002. (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).
- [13] David Harel and Rami Marelly. 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, September 2001.
- [14] J. Henriksen, M. Mukund, K. Kumar, and P. Thiagarajan. On message sequence graphs and finitely generated regular MSC languages. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, volume 1853 of LNCS, pages 675–686. Springer-Verlag, 2000.
- [15] MSC-96: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 1996. <http://www.itu.int/>.
- [16] Jochen Klose and Hartmut Wittke. An Automata Based Interpretation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *Proc. of TACAS (Tools and Algorithms for the Construction and Analysis of Systems) 2001*, volume 2031 of LNCS, page 512, Genova, Italy, April 2001. Springer-Verlag.
- [17] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.
- [18] Christof Löding. Efficient minimization of deterministic weak  $\omega$ -automata. *Information Processing Letters*, 79:105–109, 2001.
- [19] Sjouke Mauw, Michel A. Reniers, and T.A.C. Willemse. *Handbook of Software Engineering and Knowledge Engineering*, volume 1 (Fundamentals), chapter Message Sequence Charts in the Software Engineering Process, pages 437–463. World Scientific Publishing Co. Pte. Ltd., December 2001.
- [20] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 134–191. MIT Press and Elsevier Science Publishers, Amsterdam, The Netherlands and Cambridge, Massachusetts, 1990. ISBN 0-262-72015-9 (Second Printing, 1998).
- [21] Sebastian Uchitel, Jeff Magee, and Jeff Kramer. From Sequence Diagrams to Behaviour Models. In *WTUML: Workshop on Transformations in UML. Satellite event of the European Joint Conference on Theory and Practice of Software (ETAPS'01)*, Genova, Italy, April 2001.
- [22] Jon Whittle and Johan Schumann. Automatic Synthesis of Agent Design in UML. In J.L. Rash, C.A. Rouff, W. Truszkowski, D. Gordon, and M.G. Hinchey, editors, *First Workshop on Formal Approaches to Agent-Based Systems (FAABS)*, number 1871 in LNAI, Greenbelt, MD, USA, April 2000. Springer-Verlag.