

Applying LSCs to the specification of an Air Traffic Control system

Yves Bontemps *
University of Namur
Computer Science Department
Belgium
ybo@info.fundp.ac.be

Patrick Heymans †
University of Namur
Computer Science Department
Belgium
phe@info.fundp.ac.be

Hillel Kugler ‡
Weizmann Institute of Science
Department of Computer Science and Applied Mathematics
Israel
kugler@wisdom.weizmann.ac.il

Abstract

We demonstrate the use of the language of Live Sequence Charts (LSCs) for specifying part of the air traffic control system CTAS (Center TRACON Automation System). We use a recent extension of LSCs to handle symbolic instances, allowing an instance to be associated with a class rather than with an object. This allows us to specify scenario-based requirements that could not have been expressed using concrete objects only. This work can form the basis for applying execution, verification and synthesis methods developed for LSCs, on a real-world case study.

1. Introduction

As suggested by the organizers of the second workshop on Scenarios and State Machines (SCESM), we applied the scenario-based language Live Sequence Charts (LSCs, for short) to the proposed case-study. This case-study is based on the NASA's CTAS (Center TRACON Automation System), *a set of tools designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports* [1, 21].

*Research Fellow of the FNRS (Belgian National Fund for Scientific Research)

†This research was supported by the OSTC (Belgian Federal Office for Scientific, Technical and Cultural Affairs)

‡This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems and by the European Commission project OMEGA (IST-2001-33522).

The requirements for the weather update component of CTAS are natural to specify using a scenario-based language, as was observed in [20]. The inter-object approach is better suited than the intra-object approach, using, e.g. Statecharts [8, 10], for early phases of requirements engineering (RE), in which most of the difficulty is confined to interaction issues. However, a smooth, preferably tool-assisted, transition from inter-object to intra-object specifications is needed, because intra-object notations are better suited for later development stages.

For a discussion on the relation between scenario-based inter-object specifications and state-based intra-object behavior see [9].

Having made the choice of using a scenario-based language, we decided to use LSCs [6], which extend the classical Message Sequence Charts (MSCs) of the ITU [14] or the UML variant [18]. LSCs significantly enhance the expressive power of MSCs by enabling the specification of various kinds of scenarios of the system - including those that are mandatory, those that are allowed but not mandatory, and those that are forbidden. We use a recent extension of LSCs to handle symbolic instances [16], allowing an instance to be associated with a class rather than with an object. This allows us to specify scenario-based requirements that could not have been expressed using concrete objects only.

We found LSCs well suited for describing the case study. Although this is still preliminary work, it seems that we managed to specify accurately interesting properties, in an easy to use and understandable manner. We believe that this work can be a first step in apply-

ing execution, verification and synthesis methods developed for LSCs [13, 12, 11, 4], on a real-world case study.

The paper is organized as follows. Sec. 2 provides a brief overview of the case study, of the assumptions we made while specifying it and of the scope of requirements we handled. Sec. 3 describes the LSC language used and illustrates how the case-study requirements were specified with LSCs.

Sec. 4 describes the main findings of our application of LSCs to the CTAS case study¹ and compares it with related work. We conclude in Sec. 5 with a summary of our observations and contributions and mention future research directions.

2. Case-Study Description

2.1. Overview

The CTAS consists of a central component, called the “communication manager” (CM), exchanging messages with clients of various kinds. Almost every communication occurring in the CTAS goes through the CM which therefore has the responsibility of gathering information and maintaining the states of its clients synchronized.

The subject of the case study is the part of the CTAS dealing with weather data updates. Indeed, some of the CM’s clients are said to be weather-aware (WAClients) in the sense that their work depends heavily on the availability of accurate information about the weather. The critical top-level requirement for the CM is that “all connected WAClients are notified of weather updates and start using the updated data at the same time”.

This requirement has been refined into an operational description of the system control logic which forms a 10-page document written in natural language. We call it the “requirements document” (RD). Actually, the RD describes the logic of several entities, like the clients and the user interface (called the “weather control panel” or WCP).

The class diagram in Fig. 1 details the relationships and state spaces associated to these domains. Cardinalities of classes are indicated in their upper right corner, denoting how many instances of the class can exist in the application domain at the same time. The system boundary is used to make a distinction between the objects that we consider part of the system and those we consider part of the environment. The choice

¹The specification can be found at <http://www.info.fundp.ac.be/~ybo/scesm>.

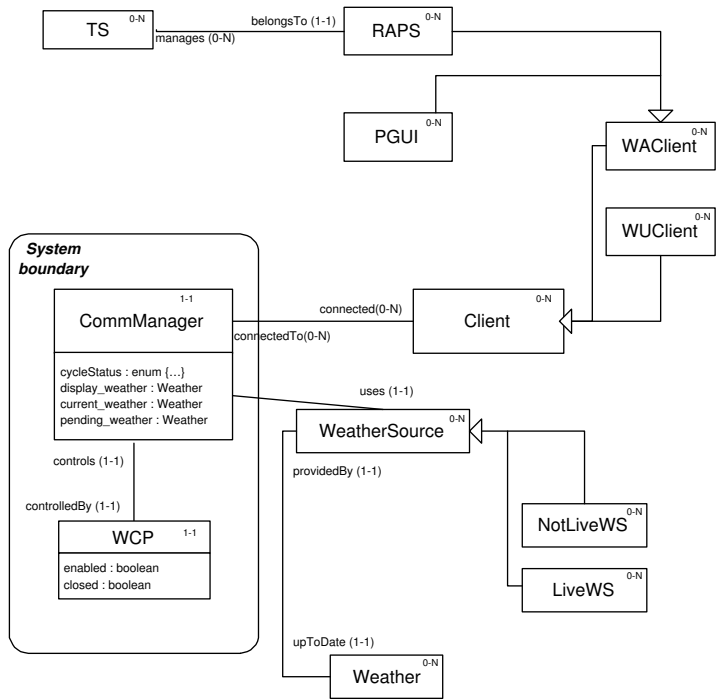


Figure 1. Class diagram

of the system boundary is a consequence of the aforementioned top-level critical requirement. This separation could be refined or modified in different stages of the development.

2.2. Assumptions and scope of the study

The case study presents how weather-aware clients connect to the CM and how the CM notifies these clients of weather data updates.

The RD is organized in two main sections. Sec. 1 is very short (1/3 page) and just states some general assumptions. Sec. 2 is 9 pages long and is divided into 13 subsections (2.1 to 2.13) focusing on the requirements.

Sec. 2.1 to 2.5 of the RD describe the states of the various agents and the messages they exchange. Sec. 2.6 to 2.13 describe the logic of these agents, in the form of scenarios, like (Req. 2.6.2) “The CM should perform the following actions when a weather-aware client attempts to establish a socket connection (a) it should set the weather-aware client’s weather status to pre-initializing, (b) it should set the Weather Cycle status to pre-initializing, (c) it should disable the F2 Weather control panel update button [...]”.

In general, we managed to stick to the content of the RD except for the following.

First, we had to abstract from the content of Sec. 2.9. This section states that, depending on the weather source in use, the CM should read the time of a rendez-vous file to determine if some new weather data is available. In a more conceptual fashion, we assumed that the weather source used by the CM is always associated with the most up-to-date weather forecast and that the CM keeps track of the forecast currently used. Thus, whenever the most up-to-date forecast is different from the current weather data, this data must be updated.

We ignored Sec. 2.10, which gives the runtime options to the CM and was considered to be too design-oriented to be modeled. We also ignored Sec. 2.12 because, although we could not fully understand it, we believed that it contained mostly design information.

Finally, we also added an unmentioned acknowledgment message when a client wants to connect to the CM. This was felt necessary in order to model the fact that the CM can accept or block incoming connections, depending on its current state.

3. Live Sequence Charts

Live Sequence Charts [6] are graphically very similar to MSCs [14] but semantically richer, in that they make an explicit distinction between things that *may* occur (*existential scenarios* or provisional behavior) and things that *must* happen (*universal scenarios* or mandatory behavior). For instance, the LSC of Fig. 2 translates exactly Req. 2.6.2 (see Sec. 2.2): *whenever* a disconnected weather-aware client tries to connect to the CM, the CM *must* set the client’s weather status to preinitializing, enter the preinitializing mode and disable the F2 WCP.

In [15], the author recognizes that there are four interpretations of MSCs: existential, universal, exact and forbidden (negation). In [15] the choice between the different interpretations is left open and is not part of the syntax of the language. The universal interpretation of [15] means that in every execution of the SUD, the MSC behavior must eventually be observed; This is different from our interpretation, as in [15] any behavior can occur before and after this observation.

The authors of [17] introduce a trigger/action mechanism into MSCs, which is somewhat similar to the prechart construct of LSCs. If an instance matches the trigger, it must follow its action afterwards.

3.1. Basic constructs

An LSC, like an MSC, is made of several interacting *instances*, depicted by vertical lines, along which time

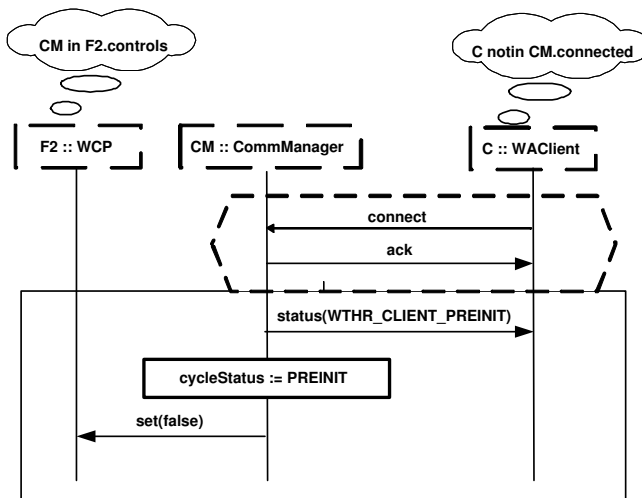


Figure 2. Requirement 2.6.2

flows from top to bottom. These instances interact by passing messages to each other, represented by arrows, going from the sender to the receiver.

An LSC specification describes the (infinite) set of runs that the future system will be allowed to perform.

Universal charts (e.g., Fig. 2) are used to specify restrictions over all possible system runs. A universal chart typically contains a *prechart* (top dashed hexagon), that specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body.

LSCs provides additional constructs like conditions, if-then-else, (see Fig. 3) and assignments (see section 3.3).

3.2. Symbolic Instances

Existential quantification over instances Consider Req. 2.6.2 from the RD (Sec. 2.2). It can apply to *any* WAClient instance that is not connected to the CM. This requirement is specified in Fig. 2 using symbolic instances as introduced in [16]. The meaning of this chart is that any client that is initially not connected (binding condition appears above the instance name), that sends a connection request, which is acknowledged, causes the scenario appearing in the main chart to occur, where the status message is sent to the connecting instance. Existentially quantified symbolic instances have their name surrounded by dashed-line box.

Universal quantification over instances Although existential symbolic instances improve expressiveness, they are sometimes not sufficient. For

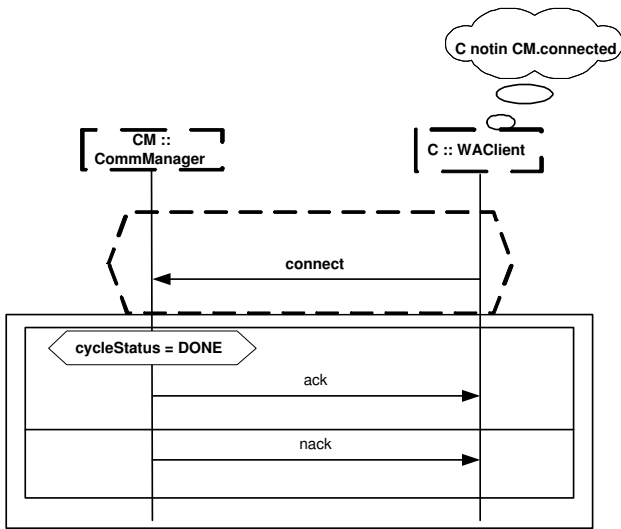


Figure 3. Requirement 2.8.1

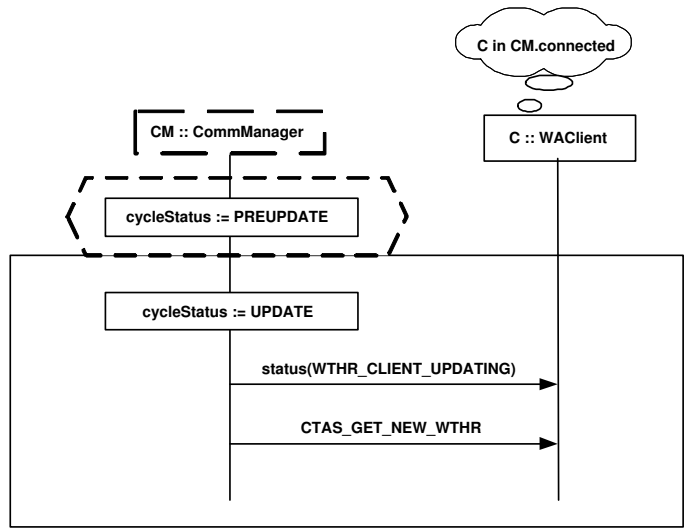


Figure 4. Requirement 2.8.10

instance, Req. 2.8.10 states that: The CM should perform the following actions when its status is PREUPDATING: (a) [...] (b) it should set the weather status of all connected weather-aware clients to ‘‘updating’’. Thus, we must be able to send a message to all client instances, *without knowing their identities nor even their number*. This requirement is specified in Fig. 4 by a universal quantification of the client instance as described in [16]. If this chart is activated *all* the connected clients will get the weather update messages described in the chart. Universally quantified symbolic instances have their name surrounded by solid-line box.

Our work on the case study has motivated us to suggest an extension of [16] to allow universal instances also in precharts. Consider Req. 2.8.12: ‘‘The CM should perform the following actions when the Weather Cycle status is ‘‘updating’’ and all connected weather-aware clients have responded yes to the CTAS_GET_NEW_WTHR message [...]’’.

This requirement is specified in Fig. 5, the meaning of the chart is that only after all connected clients get the new weather message and answer yes, the prechart is completed successfully. Then, the main chart is activated, causing the specified message communication to be sent to all connected clients.

Providing detailed formal semantics of this extension and studying how it can be integrated into an execution mechanism as done for other symbolic instance extensions [16] is a topic we are currently working on. We are also investigating other extensions allowing to

combine existential and universal quantification over instances in the same chart.

3.3. Assignments and Frame Axiom

The world’s possible states are described by a class diagram. Recall from section 2 that we distinguished between the system to be built and its environment. Attributes, as well as events, are controlled either by the former or the latter. On the one hand, some attribute values may change spontaneously as time passes (e.g., a new Weather Forecast becomes available): they are typically environment-controlled values.

On the other hand, some values may *not* change spontaneously (e.g., the WeatherSource used by CM), because they are controlled by the system. We add a frame axiom [5] to these variables: their value will only change through assignments specified in the LSCs.

Assignments are treated as events occurring between states in runs. Thus, they can be put in LSCs and monitored in precharts. Therefore, one can ‘‘check whether some variable holds some value’’ (condition) or ‘‘check whether some variable is *set* to some value’’ (assignment).

As one sees on Fig. 1, framed variables are those who are inside the system boundary.

This approach makes the rules of updating system variables explicit, through assignments and framed variables, whereas they were more implicit and part of the framework in [13, 12].

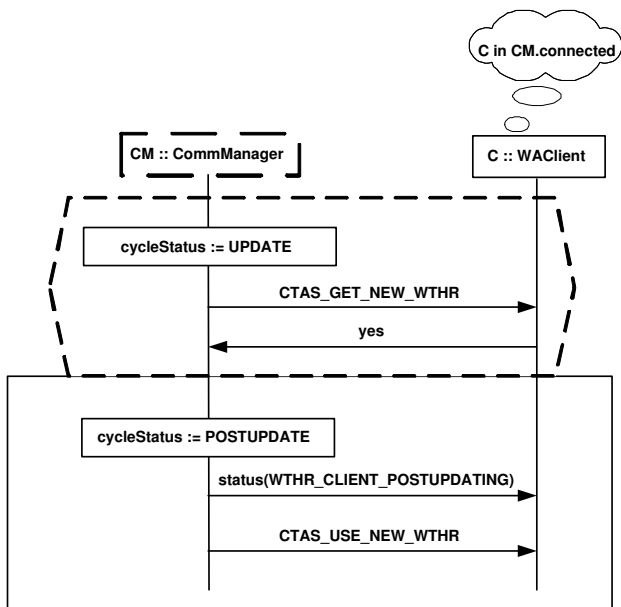


Figure 5. Requirement 2.8.12

4. Main findings

4.1. Language

Problems with specifying behavior depending on universal / existential messages using MSCs were observed by [20]. A proposal to annotate MSCs to cope with this problem and modify the synthesis algorithm to take this into account was suggested. Our approach uses and improves syntactic and semantic notions from LSCs to express class-level scenarios. In contrast with [20], these notions were *not* encoded in any synthesis algorithm, but were added explicitly to the semantics of LSCs.

With this LSC language, we could model almost seamlessly all the scenarios of the case-study. Indeed, as in [20], a negligible amount (about one day) of clerical work was needed to translate the RD into scenarios. In contrast with [20], who dealt with 19 scenarios, we translated about 40 statements into scenarios.

The reader will see that LSCs are a straightforward graphical representation of the textual requirements. We strongly believe that even non-technical stakeholders can understand LSCs, and actively participate in capturing them.

In comparison with [20], and with MSCs in general, we were able to translate the response pattern [7]: “whenever something occurs, do something else afterwards”. This is not possible with MSCs, because they ignore the notions of activation and liveness (enforcing actions) [6, 2].

4.2. Flaws discovered

Finally, specifying requirements using LSCs allowed us to find two possible flaws in the original RD. First of all, at the beginning of a new connection opening, the WCP is disabled. If a connection failure occurs in the “initialization” phase, the CM closes the connection, goes back to its “done” state but does not re-enable the control panel. This flaw can be demonstrated using play-out [13] or verification techniques [3], and expressed in the form of an existential LSC (i.e. an MSC), see Fig. 6. This flaw is due to the presence of several interacting scenarios: Req. 2.6.2, 2.8.3 and 2.8.6. By lack of space, we cannot reproduce these requirements here. This kind of problems is typical to scenario-based specifications, in which scenarios interaction can cause undesired behavior. Clearly, algorithms for validating the specification, either by directly animating it or by executing synthesized code, would be useful in coping with these problems.

Secondly, when the CM detects that a new weather forecast is available, it launches the update procedure. However, the RD does not specify that the newly available weather forecast should be saved in the “pending_weather” variable. We believe that this is a flaw, because executing the update procedure with an uninitialized variable could lead to unpredictable results.

As far as we know, these flaws have not been discovered by [20], possibly because their purpose was not to analyze the existing RD but rather to experiment their synthesis algorithm.

We do not know if these flaws have been propagated to the implementation of the CM. We consider these flaws as real “bugs” and not as under-specification problems.

5. Conclusion

We applied LSCs with symbolic instances to the proposed case study. On this case study, LSCs proved understandable and easy to use. Our work on the case study has motivated several extensions and new ideas for improving the language. They allowed us to discover what we think were two flaws in the original specification.

However, there is no free lunch. Since LSCs are much more expressive than MSCs, developing synthesis and analysis algorithms for them is a much more challenging task than for classical MSCs. Nevertheless, all the requirements information is included in the LSCs; the synthesis algorithms have to take this into account, under the semantics of LSCs, and they need

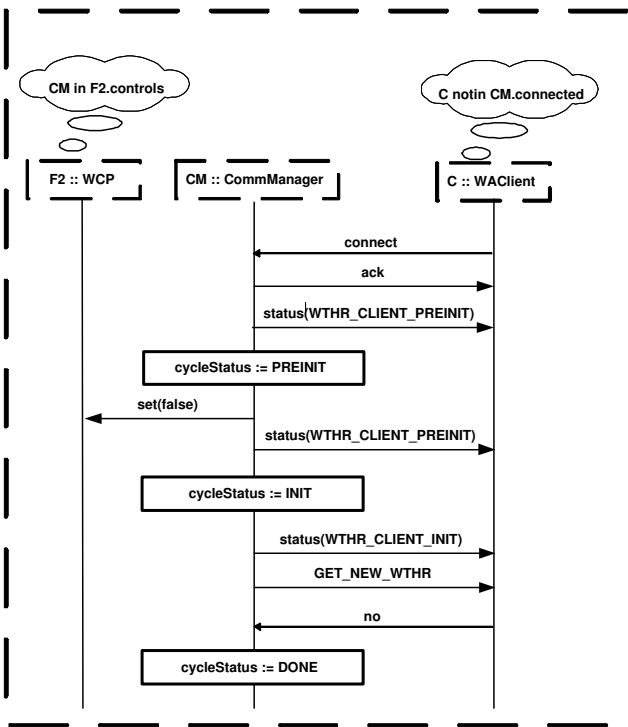


Figure 6. Flaw in the RD

not make any decision about the interpretation of the requirements.

The distinction that is made in LSCs between the system and its environment (here, through the annotation of a class diagram) enables the user to express *descriptive* and *prescriptive* statements, in a seamless manner. These two statuses are clearly and formally distinguished, see Sec. 3.3. The ability to make this distinction was spotted in [19] as one of the main challenges for formal description techniques.

Finally, we hope that this work can form the basis for applying new execution, verification and synthesis methods developed for LSCs, on a real-world case study.

Acknowledgments Prof. David Harel and Prof. Pierre-Yves Schobbens are thanked for their support and comments. The organizers of the SCESM'03 workshop and J. Whittle are thanked for having made the case study requirements document publicly available.

References

[1] Center TRACON Automation System. <http://www.ctas.arc.nasa.gov/>.

- [2] J. Bohn, W. Damm, J. Klose, A. Moik, and H. Wittke. Modeling and validating train system applications using StateMate and Live Sequence Charts. In H. Ehrig, B. J. Krämer, and A. Ertas, editors, *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*. Society for Design and Process Science, 2002.
- [3] Y. Bontemps and P. Heymans. Turning high-level Live Sequence Charts into automata. In *Proc. of "Scenarios and State-Machines: models, algorithms and tools" (SCESM) workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May 2002. ACM. <http://www.cs.tut.fi/~tsysta/ICSE/papers/>.
- [4] Y. Bontemps and P.-Y. Schobbens. Synthesizing open reactive systems from scenario-based specifications. In F. Balarin and J. Lilius, editors, *Proc. of the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03)*, Guimarães, Portugal, June 2003. IEEE Computer Science Press.
- [5] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *Transactions on software engineering (TSE)*, 21(10):785–798, October 1995.
- [6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering*. ACM, May 1999.
- [8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [9] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.
- [10] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, July 1997.
- [11] D. Harel and H. 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.).
- [12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, Portland, Oregon, pages 378–398, 2002.
- [13] D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/ Play-Out Approach. *Software and System Modeling*, 2003. To appear. (Preliminary version appeared as Weizmann Institute Technical Report MCS01-15).
- [14] MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 2000. <http://www.itu.int/>.

- [15] I. H. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technischen Universität München, July 2000.
- [16] R. Marelly, D. Harel, and H. Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. In *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, pages 83–100, Seattle, WA, 2002.
- [17] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In *SIGSOFT2002/FSE-10*, Charleston, SC, USA, November 2002. ACM, ACM Press.
- [18] UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), <http://www.omg.org>.
- [19] A. van Lamsweerde. Formal specification: a roadmap. In *ICSE - Future of SE Track*, pages 147–159, 2000.
- [20] J. Whittle and J. Schumann. 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, May, 20th 2002. ACM. <http://www.cs.tut.fi/~tsysta/ICSE/papers/>.
- [21] J. Whittle and S. Uchitel. Second International Workshop on Scenarios and State Machines: Models, Algorithms and Tools: Case Study, 2002. <http://www.doc.ic.ac.uk/~su2/SCESM/CS>.