

An Approach for Detecting Implied Scenarios

Henry Muccini
Information & Computer Science
University of California, Irvine
444 Computer Science Building, Irvine, CA
muccini@univaq.it, mucchini@ics.uci.edu

ABSTRACT

Implied scenarios are unexpected behaviors that may be derived by behavioral models when they are synthesized from scenarios. These scenarios are “implied” since they are not described in the scenario specification but each synthesized model of them will present, allowing system executions that are unacceptable (or at least unexpected) with respect to the original specification.

The work we are going to propose builds over an existing work and proposes an algorithm to identify implied scenarios avoiding the construction of the Labelled Transition System (LTS) model. The main advantage of this approach is that we can discover implied scenarios without building the synthesized model, and, much more important, we do not need to put in parallel the synthesized components LTS, saving time and preventing the possibility of state explosion problems. We will also show the ability of this algorithm in discovering implied scenarios going through an example.

1. INTRODUCTION

Scenarios define a temporal ordered sequence of events and, as shown in [22, 4] are used by researchers and industry practitioners for a variety of purposes: they have been used for requirements elicitation, validation and refinement [15, 13], to describe and clarify relevant properties uncovering system requirements [4], in the Software Architecture (SA) field to model architectural properties [7, 2, 5], at the design level to identify objects interactions [16], and for system analysis and testing [17].

A lot of interest has been paid on the realization of state-based, behavioral models in order to refine and complement scenarios; many algorithms [8, 10, 12, 23] and tools [3, 19, 9, 21] have been proposed in this direction. What may happen, however, is that the state-based model synthesized by the system scenarios presents sets of behaviors that do not appear in the scenarios themselves. This unexpected behaviors have been analyzed the first time in [1]. They are called “implied scenarios” and “*arises because specified MSCs can*

combine in unexpected ways when each process has only its own local view of the scenarios” [1].

What the authors propose in [1] is a polynomial-time algorithm to determine if a set of scenarios are “realizable” through state machines (i.e., if exists a concurrent automata which implement precisely those scenarios) and to synthesize such a realization using a deadlock-free model. If scenarios are not realizable, implied scenarios are detected. In the proposed approach, scenarios are modeled using Message Sequence Charts (MSCs) [14].

This initial work has been extended in [20] providing a more expressive language, based on MSCs and High-level MSCs (hMSCs). The authors propose an “algorithm” to synthesize a behavioral model describing the closest possible implementation for a specification based on MSCs, then, they describe a “technique” to detect, on the synthesized model, implied scenarios. The algorithm analyzes the MSCs and hMSC specifications and builds a LTS for each component specified in the MSCs. The complete system is the parallel composition of all synthesized components. The algorithm has been implemented in the LTSA tool [11]. The technique to detect implied scenarios builds a safety property accepting traces that behave correctly according to the MSC and hMSC specifications and check that the synthesized LTS model satisfies such property using LTSA. In other terms, the complete model is built and the property validity is checked on it.

The work we are going to propose builds over [20] and proposes an algorithm to identify implied scenarios avoiding the construction of the LTS model. The main advantage of this approach is that we can discover implied scenarios without building the synthesized model, and, more importantly, we do not need to put in parallel the synthesized components LTS, saving time and preventing the possibility of state explosion problems. In some ways, this work follows the general themes of system (in)completeness and (in)consistency that have received a lot of attention in the last few years [6, 18].

The rest of the paper is structured as follows: Section 2 analyzes the technique proposed in [20] in order to identify *why* and *when* implied scenarios are introduced. Section 3 proposes an algorithm to detect implied scenarios, avoiding the construction of the synthesized model, and applies it to an example. Finally, Section 4 presents some initial results and considerations and conclude the work.

2. DISCOVERING IMPLIED SCENARIOS

The aim of this paper is to discover implied scenarios

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

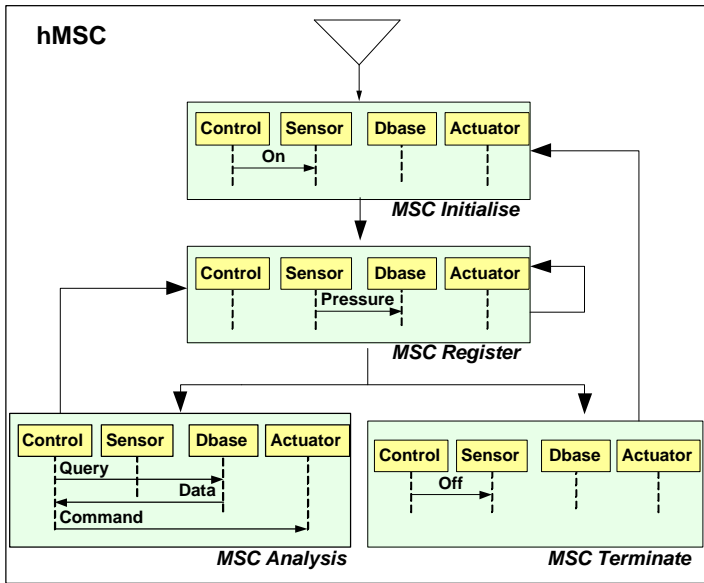


Figure 1: The Sensor-Actuator example

avoiding the construction of the synthesized model. To do that, we are going to analyze if there is anything, in the synthesis process proposed in [20], that can give rise to the generation of an implied scenario. To perform this analysis, we will summarize the synthesis algorithm proposed in [20] and we will reuse the example they used in that paper.

Given a hMSC and the MSCs composing it (like those in Figure 1), they proposed to build a process for each component C in the hMSC alphabet¹ and to specify them using the Finite State Process algebra (FSP) [11]. This FSP process describes all the events belonging to C . The order in which events² are executed in C is identified analyzing the hMSC specification.

An important concept is that of **continuation**: let b_i MSCs in the hMSC specification and C a component, b_2 is a continuation of b_1 if i) b_2 may be reached from b_1 in through one edge on the hMSC or if ii) there is a b_3 such that b_3 is a continuation of b_1 , b_2 is a continuation of b_3 and b_3 does not contain events for component C . Exemplifying, in Figure 1, continuations of “Initialise” for component *Control* are “Register” (for the first rule), “Analysis” (since Register is a continuation of “Initialise”, “Analysis” is a continuation of “Register” and “Register” does not contain events for component *Control*) and “Terminate” (since “Register” is a continuation of “Initialise”, “Terminate” is a continuation of “Register” and “Register” does not contain events for *Control*).

b_2 is a **maximal continuation** of b_1 if i) b_2 is a continuation of b_1 and ii) for all b_3 continuations of b_1 , b_2 is a continuation of b_3 . Using the example, only “Analysis” and “Terminate” are maximal continuations of “Initialise” for component *Control* since i) they are continuations of “Initialise” and ii) “Analysis” and “Terminate” are continuations of “Register”, continuation of “Initialise”.

Following the described algorithm, the synthesized LTSs

¹ *Control, Sensor, Dbase* and *Actuator* in Figure 1.

² the terms “event” and “action” will be used to identify messages exchanged in the MSCs.

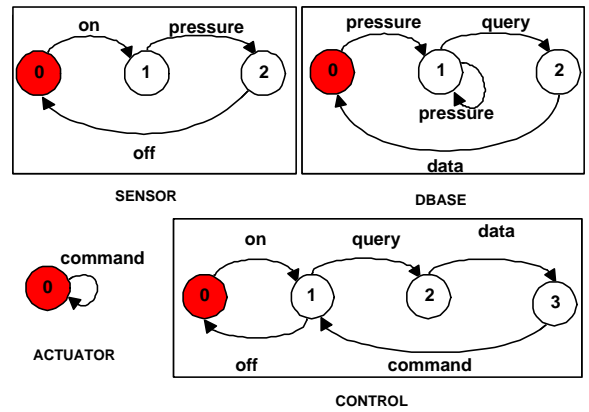


Figure 2: The synthesized LTSs

built for the hMSC in Figure 1 are depicted in Figure 2. The implied scenario detected in [20] using their implied scenario detection algorithm follows:

“*Control* sends *On* to *Sensor*, the *Sensor* sends *Pressure* to the *Dbase*, the *Control* sends *Off* to the *Sensor*, the *Control* sends *On* to the *Sensor* and the *Control* sends *Query* to the *Dbase*”.

This behavior represents an implied scenario since the *Query* message the *Control* sends to the *Dbase* component is requested over an old value of *Pressure*, and this kind of behavior is not provided by the MSCs and hMSC specifications in Figure 1.

We are now interested to understand *why* the synthesized system contains this implied scenario. The only motivations offered by the existent papers are that “*components do not have enough local information to prevent the implied execution*” [20] or “*MSCs can combine in unexpected ways when each process has only its own local view of the scenarios*” [1].

If we analyze the synthesized LTSs in order to understand why the implied behavior happens, we have:

1. the *Control* sends the *On* message to the *Sensor*. They both modify their initial state reaching state 1, i.e., (*Sensor*,1), (*Control*,1). This notation will be used in the following to identify the actual state of components;
2. the *Sensor* sends the *Pressure* msg to the *Dbase*, storing the actual pressure value. (*Sensor*,2) and (*Dbase*,1);
3. the *Control* sends *Off* to the *Sensor*. (*Control*,0), (*Sensor*,0);
4. the *Control* sends *On* to the *Sensor*. (*Sensor*,1), (*Control*,1);
5. at this point, the unexpected behavior happens: even if the *Sensor* does not send any *Pressure* value to the *Dbase* (as required by the MSC “*Register*”), the *Control* is ready to send a *Query* msg to the *Dbase*. Looking at Figure 1 what happen is that the system can directly evolve from MSC “*Initialise*” to “*Analysis*”, without running MSC “*Register*”.

What happened in this scenario is that the *Dbase* component, once reached state 1 does not evolve, even if the system

```

Database = Register,
Register = (pressure -> (internalAction ->Analysis |
    internalAction -> Register)) \{internalAction}.
Analysis = (query -> data -> Register),

```

Figure 3: FSP synthesis of the Dbase component

evolves and even if the *Sensor* and the *Control* components go back to their initial state. When *Control* is ready to send the Query command to the *Dbase* (*Query,1*), the *Dbase* is ready to reply to it (*Dbase,1*), even if the *Sensor* has not yet registered a new pressure value on the *Dbase*.

What we learnt is that *the unexpected behavior is due to the Dbase component which did “not have enough local information” to understand it needs to change its state*. Understanding **why** the *Dbase* component behaves in this way will give us a key to discover implied scenarios.

If we analyze closer the synthesis algorithm, synthesizing the *Dbase* component (Figure 3) we can see that the *Dbase*, after receiving Pressure (in “Register”), is ready to receive refreshed values for Pressure or a Query command (since its state can evolve in “Register” and “Analysis”). The problem is that when the system evolves in “Terminate” the *Dbase* state does not change, it is (potentially) already able to receive queries. In other words, in the synthesized model the *Dbase* behavior in “Terminate” has been **augmented** by new actions, since it was not able to participate in “Terminate” in the hMSC specification. In the following we will say that “Terminate” is an augmented node³ for the *Dbase*. If another component is able to match the augmented behavior in “Terminate”, then they can synchronize producing the unexpected behavior.

What we can conclude at this point is that if we can identify, for each component, its augmented nodes, we can discover implied scenarios without synthesizing the LTS model. Naturally, the solution is a little bit more complicated, because we need to take many factors into a consideration. Next section will be used to present an algorithm that handles all such factors and detect implied scenarios.

3. AN ALGORITHM TO DISCOVER IMPLIED SCENARIOS

Initially, we will identify the high-level activities to be performed in order to discover implied scenarios then, we will describe an algorithm to implement each activity. Finally, the algorithm will be applied to the Sensor-Actuator example in order to put in practice.

High-level activities:

- Step1: For each component *C* in the hMSC alphabet, we have to identify all its augmented nodes. In this way we can identify possible causes of unexpected behaviors;
- Step2: for each augmented node of each component, we have to identify its value, i.e., we have to identify the events the component can perform in the augmented node.

However, augmented nodes may generate unexpected behaviors only when they contains events not previously allowed in the MSC model, then

³the term “node” will be used in the following to denote an MSC.

Step3: for each augmented node, we have to compute the **enrichment**, i.e., we have to identify those events the components were not allowed to send in the original MSC specification;

Step4: finally, we can detect the implied scenarios by just analyzing the enriched nodes: if components *A* and *B* in node “M” can communicate using enriched events then, this communication generates an unexpected behavior (i.e., the implied scenario).

The Algorithm:

- Step1.1: for each component *C* in the MSC alphabet, we create a copy of the hMSC for that component (hMSC_{*C*}) (Figure 4);
- Step1.2: for each hMSC_{*C*}, for each node M in the hMSC_{*C*}, we check if component C has events in M. If yes, we annotate M with “Y” otherwise with “N”. This step may be performed by simply looking at the hMSC specification;
- Step1.3: Each Y-annotated node in a hMSC_{*C*} that directly reaches a N node, is an augmented node (AN) for component C. It happens because the synthesis algorithm proposed in [20], computing the maximal continuation for component *C* in “Y”, assigns an augmented behavior to component *C* if it will not react (“N”) in the next MSC. A way to catch augmented nodes is just applying the following rule: “Y” is an augmented node for *C* if it is maximal continuation differs from its continuation. For example, the *Dbase* component has in “Register” a maximal continuation that differs from its continuation, then “Register” is an augmented state. Moreover, this augmented behavior is propagated to other nodes in the hMSC_{*C*} following these rules:

- if a node annotated by “Y” is followed by a “N”, the “Y” AN is propagated to node “N”; it happens because node “N” is in the path among “Y” and its maximal continuations and then it is already able to execute extra behaviors;
- if a “N” node (N1) is followed by another “N” node (N2), the state the *C* component has in N1 is propagated to N2. It happens because *C* in N2 is still in the maximal continuation of N1;
- if a “N” node is followed by a “Y” node, the “Y” node is not augmented, since component *C* will be able to act in “Y”.

Summarizing, all the “N” nodes in the path among an “Y” annotated node and its maximal continuations “Y1”, ..., “Yn” will become ANs. Notice that they will become augmented only after “Y” is reached once. The notation “AN_{*Y*}” will indicate this dependence on “Y”.

- Step2: For each augmented node AN, we have to identify what events the *C* component is enabled to act (i.e., its state). To do that it is sufficient to visit the hMSC_{*C*} assigning to component *C* in AN the state *C* has in the maximal continuation of AN (i.e., Augmented (*C*, AN)

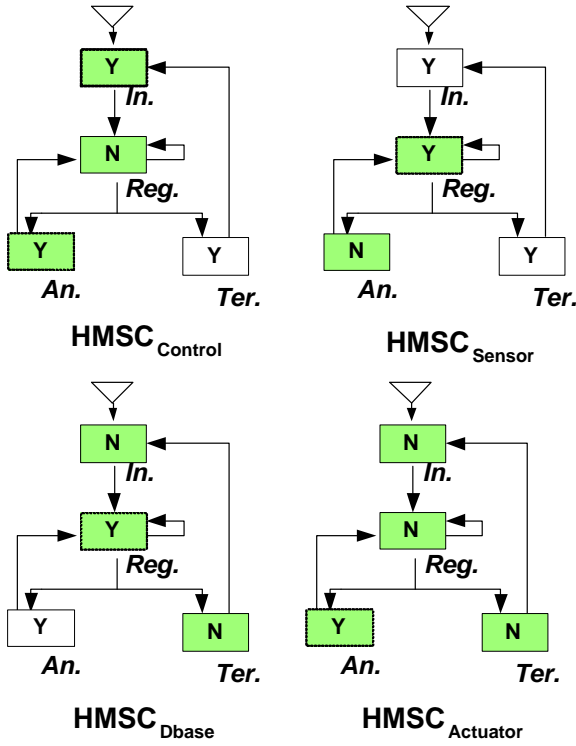


Figure 4: The hMSC_Cs

= State (C, maxCon(AN)). Notice that Augmented (C, AN_Y) = State (C, maxCon(AN_Y)) only after “Y” is reached once.

Step3: To identify the enrichment we apply the following rule: the enrichment of component *C* in an AN is given by the actions *C* may perform in AN minus the actions performed by the Y states directly reachable by AN (i.e., $\text{Enrich}(C, \text{AN}) = \text{Augmented}(C, \text{AN}) - \text{State}(C, Y)$, for all Y directly reachable by AN).

Step4: For each enriched node EN, for each pairs of components *A* and *B* in EN, if they can communicate using enriched events, an unexpected behavior carries out. The implied scenarios can be detected in this way: if event $a \in \text{Enrich}(A, \text{AN1})$ and $a \in \text{Enrich}(B, \text{AN1})$ then every path from the initial hMSC node to AN1 will represent an implied scenario.

In Practice:

Step1Pr: In Figure 4, hMSC_Cs are built for components *Control*, *Sensor*, *Dbase* and *Actuator*. Each hMSC_C node is annotated with Y or N to point out the presence/absence of process *C* in that node. To understand the annotation is sufficient to look at Figure 1.

Following the rules the algorithm dictates in Step1.3,

- “Initialise”, “Register”^{Initialise} and “Analysis” are ANs for component *Control*;
- “Register” and “Analysis”^{Register} for the *Sensor*;
- “Register”, “Terminate”^{Register} and “Initialise”^{Register} for *Dbase*;

hMSC _C	MSC Initialise	MSC Register	MSC Analysis	MSC Terminate
Control	send Query rec. Data send Off	send Query rec. Data send Off	send Query rec. Data send Off	
Sensor		send Pressure rec. Off	send Pressure rec. Off	
Dbase	rec. Query send Data rec. Pressure	rec. Query send Data rec. Pressure		rec. Query send Data rec. Pressure
Actuator	rec. Command	rec. Command	rec. Command	rec. Command

Figure 5: The enriched states

- “Analysis”, “Register”^{Analysis}, “Terminate”^{Analysis} and “Initialise”^{Analysis} for the *Actuator*.

Filled nodes in Figure 4 identify ANs;

Step2Pr: Figure 5 shows the values of the augmented nodes for each component in each node. In fact, following the rules identified in Step2 we have that:

Control:

- Augmented (Control, In._{Reg.}) = State (Control, {An., Term.})⁴;
- Augmented (Control, Reg.) = State (Control, {An., Term.});
- Augmented (Control, An.) = State (Control, {An., Term.}).

Sensor:

- Augmented (Sensor, Reg.) = State (Sensor, {Reg., An.});
- Augmented (Sensor, An._{Reg.}) = State (Sensor, {Reg., An.}).

Dbase:

- Augmented (Dbase, In._{Reg.}) = State (Dbase, {In., Reg., Ter.});
- Augmented (Dbase, Reg.) = State (Dbase, {In., Reg., Ter.});
- Augmented (Dbase, Ter._{Reg.}) = State (Dbase, {In., Reg., Ter.}).

Actuator:

- Augmented (Actuator, In._{An.}) = State (Actuator, {In., Reg., An., Ter.});
- Augmented (Actuator, Reg._{An.}) = State (Actuator, {In., Reg., An., Ter.});
- Augmented (Actuator, An.) = State (Actuator, {In., Reg., An., Ter.});
- Augmented (Actuator, Ter._{An.}) = State (Actuator, {In., Reg., An., Ter.}).

⁴State (C, {X,Y}) = State (C, X) + State (C, Y).

Step3Pr: Filled nodes in Figure 5 show the enriched nodes. In fact, following the rules identified in Step3, we have that:

Control:

- Enrich (Control, In.*Reg.*) = Augmented (Control, In.*Reg.*);
- Enrich (Control, Reg.) = \emptyset ;
- Enrich (Control, An.) = Augmented (Control, An.).

Sensor:

- Enrich (Sensor, Reg.) = \emptyset ;
- Enrich (Sensor, An.*Reg.*) = Augmented (Sensor, An.*Reg.*) State (Sensor, Reg.) = receive Off.

Dbase:

- Enrich (Dbase, In.*Reg.*) = Augmented (Dbase, In.*Reg.*) - State (Dbase, Reg.) = (receive Query, send Data);
- Enrich (Dbase, Reg.) = \emptyset ;
- Enrich (Dbase, Term.*Reg.*) = Augmented (Dbase, Term.*Reg.*).

Actuator:

- Enrich (Actuator, In.*An.*) = Augmented (Actuator, In.*An.*);
- Enrich (Actuator, Reg.*An.*) = \emptyset ;
- Enrich (Actuator, An.) = Augmented (Actuator, An.);
- Enrich (Actuator, Ter.*An.*) = Augmented (Actuator, Ter.*An.*).

Step4Pr: Finally, we analyze if different components enriched states may match in the same node, taking a look at each column in Figure 5. We can see is that in “Initialise”, *Control* and *Dbase* can match with Query, in “Analysis”, *Control* and *Sensor* can match with Off. But “Initialise” and “Analysis” are enriched after “Register” is reached at least once; then, any system execution from the initial node to the “Initialise” node will generate an implied scenario, as long as “Register” is reached once before “Initialise”. Any system execution from the initial node to the “Analysis” node will generate an implied scenario, as long as “Register” is reached once before “Analysis”.

To give an example, two different implied scenarios that may be detected are:

- i) “*Control* sends On to *Sensor*, the *Sensor* sends Pressure to the *Dbase*, the *Control* sends Off to the *Sensor*, the *Control* sends On to the *Sensor* and the *Control* sends Query to the *Dbase*” that has been also detected in [20];
- ii) “*Control* sends On to *Sensor*, the *Sensor* sends Pressure to the *Dbase*, the *Control* sends Query to the *Dbase*, the *Dbase* sends Data to the *Control*, the *Control* sends Command to the *Actuator* and the *Control*

is able to send Off to the *Sensor* even if the system is not evolved in “Register” (as required in the hMSC specification).

4. INITIAL RESULTS AND CONCLUSIONS

We presented an approach to detect implied scenarios avoiding the construction of the global LTS model synthesized by the MSC and the hMSC specifications. We have not yet performed a completeness and correctness analysis of our algorithm, then, we cannot claim that the algorithm discovers all (and only) scenarios that are implied. Anyway, its application to some different systems (the algorithm has been also applied to the ATM systems in [21]) gives us confidence on its quality: our approach discovered in all the examined examples more implied scenarios than the LTSA-implied tool [21] and all the results we had represent scenarios that are implied.

An actual limitation the approach presents is that implied scenarios may be only detected analyzing both bMSC and hMSC specifications, therefore, it does not work with single bMSC specifications, hiding implied scenarios detectable with the approach proposed in [1].

In future work we are planning to present a more formal description of the described approach and a complexity analysis. An initial analysis puts in evidence how it should be easily automated using graph-based analysis techniques: representing hMSC_C using a graph whose nodes are MSCs and are labelled by information on how components are behaving inside the MSC, we can easily implement our four steps approach. Steps from 1 to 3 are no more than visit of this graph and analysis of the information stored in the nodes. Moreover, the different hMSC_C graphs may be concurrently analyzed until the last step which requires results analysis.

Acknowledgments

The author would like to acknowledge the Italian M.I.U.R. National Project SAHARA and the University of California Irvine that partly supported this work and the anonymous reviewers for their valuable help in understanding weaknesses and strengths of the presented approach.

5. REFERENCES

- [1] R. Alur, K. Etessami and M. Yannakakis. Inference of Message Sequence Charts. In Proc. 22nd Int. Conf. on Software Engineering, ICSE2000, Limerick, Ireland.
- [2] L. Bass, P. Clements and R. Kazman. Analyzing Development Qualities at the Architectural Level. BOOK: Software Architecture in Practice, Chapter 9.
- [3] H. Ben-Abdallah and S. Leue. MESA: Support for Scenario-Based Design of Concurrent Systems. In Proc. of the 4th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’98. LNCS 1384.
- [4] K. M. Benner, M. S. Feather, W. L. Johnson and L. A. Zorman. Utilizing Scenarios in the Software Development Process. Information System Development Process, Elsevier Science Publisher B.V. (North-Holland), 1993, 117-134.

- [5] H. de Bruin and H. van Vliet. Scenario-Based Generation and Evaluation of Software Architectures. In Proc. Int. Conf. on *Generative and Component-Based Software Engineering*, September 2001, LNCS 2186.
- [6] IWSSD-10: Tenth International Workshop on Software Specification and Design. Track on (In)completeness, (in)consistency and (sufficient) correctness. San Diego, California, November 2000.
- [7] R. Katzman, G. Abowd, L. Bass and P. Clements. Scenario-Based Analysis of Software Architecture. IEEE Software, 1996.
- [8] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7):643-658, 1994.
- [9] K. Koskimies, T. Männistö, T. Systä and J. Tuomi. Automated support for OO software. *IEEE Software*, 15(1):87-94, 1998. Tool Download at: <http://www.cs.tut.fi/~tsysta/sced/>.
- [10] I. Krüger, R. Grosu, P. Scholz and M. Broy. From MSCs to Statecharts. *Distributed and Parallel Embedded Systems*, Kluwer Academic, 1999.
- [11] Labelled Transition System Analyzer (LTSA). On-line at: <http://www-dse.doc.ic.ac.uk/~jnm/book/>.
- [12] S. Leue, L. Mehrmann and M.Rezai. Synthesizing ROOM models from message sequence chart specifications. In *Proc. Of the 13th IEEE Conf. on Automated Software Engineering*, 1998.
- [13] L. Liu and R. Yu. From Requirements to Architectural Design - Using Goals and Scenarios. In Proc. First Int. Workshop "From Software Requirements to Architectures" (Straw'01), ICSE2001.
- [14] Message Sequence Chart (MSC). ITU Telecommunication Standardization Sector (ITU-T). Z.120 Recommendation for MSC-2000, year 2000.
- [15] C. Potts, K. Takahashi and A.I. Anton. Inquiry-Based Requirements Analysis. IEEE Software, March 1994.
- [16] Rational Corporation. Uml Resource Center. UML documentation, version 1.3. Available from: <http://www.rational.com/uml/index.jhtml>.
- [17] J. Ryser and M. Glinz. A Practical Approach to Validating and Testing Software Systems Using Scenarios. QWE'99: Third International Software Quality Week Europe, Brussels, Nov. 1999.
- [18] Second International Workshop on Living with Inconsistency. ICSE'01 workshop, May 13, 2001 Toronto, Canada.
- [19] UBET. <http://cm.bell-labs.com/cm/cs/what/ubet/>.
- [20] S. Uchitel, J. Kramer and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. in Proc. European Software Engineering Conference (ESEC/FSE'01), Vienna 2001.
- [21] S. Uchitel, J. Magee, and J. Kramer. LTSA and implied Scenarios. On line at: <http://www.doc.ic.ac.uk/~su2/Synthesis/ImpliedScenarios/FSE/index.html>.
- [22] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenario Usage in System Development: A Report on Current Practice. IEEE Software, March 1998.
- [23] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *Proc. 22nd Int. Conference on Software Engineering (ICSE'00)*, 2000.