

Towards Formal but Flexible Scenarios

Bikram Sengupta Rance Cleaveland

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
{sbikram,rance}@cs.sunysb.edu

Abstract

We argue that the evident utility of scenario-based modeling can be greatly enhanced when enriched with a mathematically precise notion of conditionality and partiality, such as that found in Triggered Message Sequence Charts (TMSCs). Unlike traditional Message Sequence Charts, the TMSC formalism allows scenarios to be conditional and partial, supports logical as well as behavioral operators for scenario composition, and is equipped with a refinement notion drawn from traditional process theory that provides a rigorous basis for stepwise development of specifications. This paper illustrates these points via the Center TRACON Automation System for flight control.

1 Introduction

Message Sequence Charts [2, 15] have gained wide acceptance as a visual, scenario-based specification language for distributed software systems. A single MSC depicts one possible exchange of messages a set of processes may engage in as they execute. Various behavioral constructs (e.g. parallel, sequential and alternative composition of MSCs [15]) have been proposed to assemble complex MSC-based specifications out of sub-specifications.

The formal MSC semantics is based on a process-algebraic approach that has been described in [15]. This semantics renders MSC-based specifications as *deterministic* and *complete*: MSC specifications are in essence interpreted as deterministic finite-state machines whose languages are the “execution runs” of the specification, and implementations of such specifications are required to exhibit exactly the same execution sequences as the specification. However, MSCs are heavily used in capturing requirements in the initial stages of system development, when many design-related issues typically remain unresolved and not all eventual implementation scenarios may be known. Hence, although the sound mathematical underpinning of

the semantics in [15] has appeal, we contend that it is overly rigid: on the one hand, requirements that *constrain* rather than just *prescribe* implementation behavior cannot be expressed; on the other, it imposes substantial burdens on the requirements elicitation process, since all eventual behaviors must be included in an initial specification. Moreover, the standard MSC semantics does not support refinement-based reasoning and thus lacks a framework within which specifications may be built via a sequence of steps, with a refinement relation providing the rigorous basis for assessing whether or not a more detailed specification is indeed faithful to the specification it is intended to elaborate on.

In this paper, we argue that these points are addressed by the Triggered Message Sequence Chart (TMSC) formalism [16]. In the following sections, we put forward the position that TMSCs represent an intuitive, flexible and yet, at the same time, fully formal and hence mechanically analyzable framework for scenario-based specifications.

2 TMSCs

The TMSC visual syntax enriches that of MSCs with capabilities to express *conditional* and *partial* behavior. An example TMSC is shown in Fig. 1.

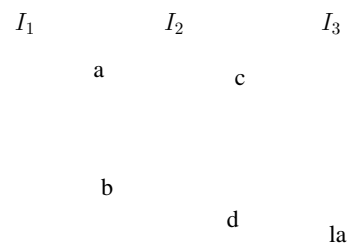


Figure 1. An Example TMSC

Conditional Scenarios. TMSCs allow the partition of the sequence of atomic actions (message sends and receives, and local actions) for each parallel process (called an *instance*) in a scenario into two subsequences: a *trigger* and an *action*. This partition is effected by a dashed horizontal line running through the instances of a TMSC as shown in Fig 1. A TMSC scenario stipulates that in any system execution, if an instance performs the sequence of events constituting its *trigger*, then the subsequent behavior of the instance must include the sequence of events which constitute its *action*; otherwise there are no constraints on the behavior of the instance. The performance of the action is thus *conditional* on the occurrence of the trigger, and represents a *constraint* that the instance needs to satisfy in the scenario depicted by the TMSC.

In designing the visual syntax of TMSCs we were confronted with representing causality, on the one hand (triggers “cause” actions), and the passage of time on the other (the events within the instances of a TMSC occur in “top-to-bottom” order). These notions are orthogonal, and in particular an event (say, a send) occurring in the action of one instance (below the dashed line) may lead to an event (the corresponding receive) occurring in the trigger of another instance (above the dashed line). This leads “upward” arrows such as the one labeled *d* in Fig. 1. The presence of these upward arrows should not be misconstrued as representing “time flowing backward”: they are merely used to record action-trigger dependencies between different instances. The events in individual instances still occur in the usual top-to-bottom order as represented in the syntax.

Partial Scenarios. TMSCs have the ability to express whether a scenario is complete, or whether it is *partial* and thus *extensible*. This is achieved by the presence/absence of a small bar at the foot of each instance in a TMSC. The presence of a bar (as in instance I_1 in Fig 1) indicates that the instance cannot proceed beyond this point in this scenario, while the absence (as in instance I_2) means that the behavior of this instance beyond the TMSC is left unspecified, i.e. there are no constraints on its subsequent behavior.

Following the above informal description of a TMSC, Fig. 1 may be read as follows:

If I_1 sends *a* to I_2 , then it should receive *b* from I_2 and terminate; if I_2 receives *a* from I_1 and *c* from I_3 , then it should send *b* to I_1 and *d* to I_3 , and its subsequent behavior is left unspecified; if I_3 sends *c* to I_2 and receives *d* from I_2 , then it should perform the local-action *la* and terminate.

The trigger/action requirements are thus localized to each instance. Also, the trigger (action) of one instance may depend on the action (trigger) of another instance in a TMSC, which results in messages crossing the horizontal

line as shown in Fig. 1. Thus a trigger in a TMSC need not be a “consistent cut” in the sense of [6].

The flexibility of TMSCs lies in their provision of a uniform framework for expressing different types of scenarios: not only may conditional and partial scenarios as described above be captured, but also traditional (deterministic and complete) MSC-based scenarios simply by placing the trigger line at the very top of a TMSC (above all events), and placing a bar at the foot of each instance in the TMSC: an empty trigger would mean that each instance immediately needs to satisfy its action, while the bar would signify the termination of the instance as soon as it completes its action. Because triggers/actions are localized, TMSCs offer an even more fine-grained specification capability, whereby the behavior of one instance may be deterministic/complete, while that of another may be conditional/partial in the same TMSC.

Structured Specifications. For small systems, it suffices to have an unstructured set of scenarios to specify their behavior. However, for large and complex systems, such an approach would quickly become unmanageable. In our framework, single TMSCs serve as the basic building blocks for more complex scenario-based specifications. Specifically, we have proposed an algebra of operators to facilitate a structured approach to specification development. The resulting terms, called *TMSC expressions*, have the following syntax:

$S ::= M$	(single TMSC)
X	(variable)
$S \parallel S$	(parallel composition)
$S \mp S$	(delayed choice)
$S; S$	(sequential composition)
$recX.S$	(recursive operator)
$S \oplus S$	(internal choice)
$S \wedge S$	(logical and)

The TMSC language offers a selection of *behavioral* and *logical* operators, as opposed to the purely behavioral constructs available for structuring MSCs. Thus \parallel , \mp , $;$ and rec are behavioral, \wedge is logical, and \oplus falls in both categories. $S_1 \parallel S_2$ denotes the parallel composition of S_1 and S_2 (the semantics of which, as described in [16], differs significantly from the standard semantics in [15]) while $S_1; S_2$ represents the *asynchronous concatenation* [5, 15] of the two expressions. We have two choice operators: (i) $S_1 \mp S_2$, which represents the “deterministic choice” between S_1 and S_2 : a correct refinement must be able to behave like both S_1 and S_2 until their behaviors differ, at which point a choice is allowed; and (ii) $S_1 \oplus S_2$, which is the non-deterministic choice between S_1 and S_2 : a successful refinement may choose to refine either. In this latter respect, \oplus has overtones

of logical disjunction. $S_1 \wedge S_2$ represents the logical conjunction of S_1 and S_2 . S_1 and S_2 may specify two different constraints on a system’s behavior, and we write $S_1 \wedge S_2$ to specify a system that has to satisfy both the constraints. Finally, the recursive operator *rec* allows us to model recurring behavior of processes in which a new execution “cycle” starts whenever there is a reference to the variable used in the recursive definition.

Semantics. Undoubtedly, the greatest appeal of scenarios lies in their simplicity and intuitive semantics. Accordingly, there may be a temptation to avoid defining a formal semantics, lest it obscure the clarity of scenarios with mathematical detail. However, we feel that a formal semantics is necessary to fully exploit the potential of scenarios and integrate them with mainstream software development. For example, tool support for analyzing scenarios has great value with the increasing complexity of real-world systems, and such tools should ideally have a firm, platform-independent semantic footing. Secondly, as groups of scenarios are composed in different ways to specify a system in a structured manner, it is important to ensure that the same specification does not mean different things to different people. Moreover, a formal semantics also serves as a common platform for comparing scenarios with specifications given in other notations. For these reasons, we argue in favor of having a formal semantics for scenario-based languages, and broadly support the approach of [12, 13, 15] and others who advocate the same.

We base the formal semantics of the TMSC language on *acceptance trees* [11], which provide a simple yet powerful behavioral model for non-deterministic systems. Acceptance trees retain the operational appeal of labeled transition systems (state machines) while efficiently representing the non-determinism inherent in a system by annotating the nodes in the tree with *acceptance sets*. There are primarily two reasons why we chose the acceptance tree framework for TMSCs. Firstly, as long as the trigger of an instance in a conditional scenario is not satisfied, or after an instance has completed its action in a partial scenario, there are no constraints on its behavior. Semantically, we interpret this behavior as being completely non-deterministic, and can thus use acceptance trees to formally represent such scenarios. The semantics of TMSC expressions may then be defined by appropriate constructions over the acceptance trees of its sub-expressions. Secondly, as a byproduct of this approach, we immediately obtain a refinement ordering in terms of the *must preorder*, which relates acceptance trees based on their non-determinism. Specifically, TMSC expression S_2 refines TMSC expression S_1 , written as $S_1 \sqsubseteq_{\text{must}} S_2$, if S_2 in effect includes a subset of the non-deterministic behaviors of S_1 . An advantage of having this generic refinement notion is that it can be *applied uniformly* to differ-

ent types of scenarios: it ensures that prescriptive behaviors expressed by deterministic scenarios are indeed performed, complete scenarios are not extended, constraints expressed by conditional scenarios are correctly adhered to, and unspecified behavior in partial specifications is properly “filled in”. Another useful feature of the TMSC semantics is that it is *compositional*: a TMSC expression may be refined by refining its sub-expressions. Acceptance trees thus provide us with a formal yet flexible semantic framework which complements the flexible visual syntax of TMSCs.

Tool support. We have designed a tool named TRIM that analyses system requirements expressed in the TMSC language. TRIM is built on top of the Concurrency Workbench for the New Century (CWB-NC) [7, 8, 14], an easy-to-retarget verification tool for finite-state systems. The CWB-NC has efficient routines for computing a number of behavioral equivalences and preorders between systems, including the must-preorder. Moreover, the modular design of CWB-NC makes its analysis routines independent of the specific input language the tool accepts. For these reasons, a natural approach to TRIM was to develop a TMSC front-end for CWB-NC.

The front-end for the TMSC language consists of a description of the syntax of the language, and a routine for computing the single-step transitions of TMSC expressions. The latter was derived from a set of Plotkin-style SOS rules [10] that we designed to express the operational semantics of TMSCs. The operational semantics is an approximation of the formal denotational TMSC semantics in [16], in the sense that we had to bound (using a parameter) the number of messages in transit, as CWB-NC requires systems to be finite-state. The two semantics coincide for specifications whose number of pending messages (i.e. messages sent but not received) is always within this bound.

The syntactic and semantic components described above were integrated to create the TMSC interface within CWB-NC. TMSC expressions typed in by an user are automatically translated to acceptance trees by CWB-NC using our transitions routine. The CWB-NC back-end provides us with a (must-preorder) refinement-checker for these trees, and also a number of other useful routines like simulation, temporal-logic model checking etc. for TMSC expressions.

Related Work. Krueger [12] proposes a powerful stream-based semantics for MSCs, and provides a trigger-like composition operator for joining MSCs. However, this operator is primarily used to specify liveness properties of a system, whereas our work on conditional scenarios is motivated by the need to identify when requirements imply constraints on behavior, and when they do not (although TMSC conditional scenarios may also be used to encode liveness properties for temporal-logic model checking of TMSC expres-

sions). [12] also has a guarded MSC concept, where $P : M$ is used to mean that the execution of MSC M occurs only if P is true. Unlike the triggers in a TMSC, however, P is a state predicate rather than a sequence of events and hence does not convey information about the system behavior that makes the predicate true.

TMSCs also have similarities with Live Sequence Charts (LSCs) [9], which include trigger-like notions of activation charts and precharts; the main difference between TMSCs and LSCs lies in how such preconditions are arrived at and how they are used. LSCs primarily seek to distinguish between “live” or mandatory behavior, (universal LSCs, or behavior which should be visible in each system run), and provisional behavior (existential LSCs, or behavior displayed in at least one run) in a specification. The idea is that in the early stages of the design process, one is only able to describe scenarios that should occur in at least one run of the system (we may not know which), and the activation condition of such existential LSCs will be typically weak, even degenerating to true, since our partial knowledge of the system may be inadequate for specifying anything stronger. Later, as the behavior of the system becomes clearer, the designer can incrementally add liveness annotations, and an existential view may become a universal one, such that its activation condition now tells us exactly when that scenario should occur. Thus, the activation chart evolves with increasing knowledge about the system, and its purpose is to succinctly describe the condition under which the body of the chart will occur during a system run. However, the trigger/action requirements of a TMSC are primarily meant to *refine* (non-deterministic) base specifications; such system specifications often arise in our framework when we combine the specifications of subsystems. Since subsystem behavior typically represents a local view, the base specification obtained by the composition of such behaviors may be relatively coarse (or overly permissive) when it comes to specifying *global* system requirements. Thus, some of the global execution sequences produced by the composition of subsystems may be *undesirable*. We then use constraints in the form of TMSC conditional scenarios to *weed out* such sequences and retain only the desirable ones in the refined specification.

A second difference with LSCs arises from the fact that our base specifications may be non-deterministic, and may be *determinized* during refinement. What that means is that TMSC conditional scenarios are truly “conditional”, they may not occur in any run of an eventual implementation. Thus, in contrast to [9] which proposes “healthiness check” to ensure that the activation condition/prechart of each universal LSC is actually satisfied in some run of the system (the fact that a universal LSC is in a specification itself means that the body of the chart was meant to arise in at least one run), it is perfectly “healthy” for a TMSC condi-

tional scenario not to arise in any run of an implementation, simply because it may have been overruled during system design in favor of some other scenario in the specification. The WSR system, which was used in [16] as a running example to explain the TMSC semantics, illustrates this feature of TMSC specifications.

Scenario networks [3, 4] use sequential and concurrent relationships between scenarios to describe allowable system behavior. Scenarios in such networks are also labelled with pre- and post-conditions; but the purpose of such conditions is to determine which scenarios can occur next at any given point (based on which preconditions evaluate to true), rather than to offer choices (in the form of non-determinism) in the system behavior, as may be the case with TMSCs.

Finally, [17] presents a notation for *negative scenarios* that uses a trigger-like precondition. However, the motivation, as stated in that paper, is primarily “to provide a language for documenting rejected implied scenarios and eliciting further implied scenarios,” and unlike our work, is thus related to determining whether certain system behaviors, not present in a scenario specification, may appear in all possible implementations of the system.

3 Using TMSCs

As pointed out earlier, TMSCs can encode traditional MSC-based scenarios, and hence they may be used wherever MSCs are applicable; but the real usefulness of TMSCs arises from their enhancements when compared to MSCs, namely, *conditional scenarios*, *partial scenarios* and the *refinement relation*. We see two distinct specification methodologies based on TMSCs. The first approach is useful in modeling distributed systems whose overall behavior is derived by composing subsystems; we begin by specifying the behavior of these subsystems, and then compose them in a manner that reflects the structure of the overall system. This allows us to easily determine the global execution traces produced by the unconstrained composition of the sub-systems; some of these sequences may be undesirable (e.g. they may violate a global safety property), and the initial specification is then *refined* by eliminating these sequences using constraints in the form of conditional scenarios. The WSR system of [16] exemplifies this approach. The second approach is based on specifying system behavior *incrementally* by extending partial scenarios in a sequence of steps. It is often difficult, if not impossible, to specify the many different aspects of a system’s behavior all at once; for example, there may be many error conditions that can arise in addition to the normal operation of the system, or the behavior of the system may consist of a series of *phases*, with a different functionality in each phase. For such systems, we may begin by speci-

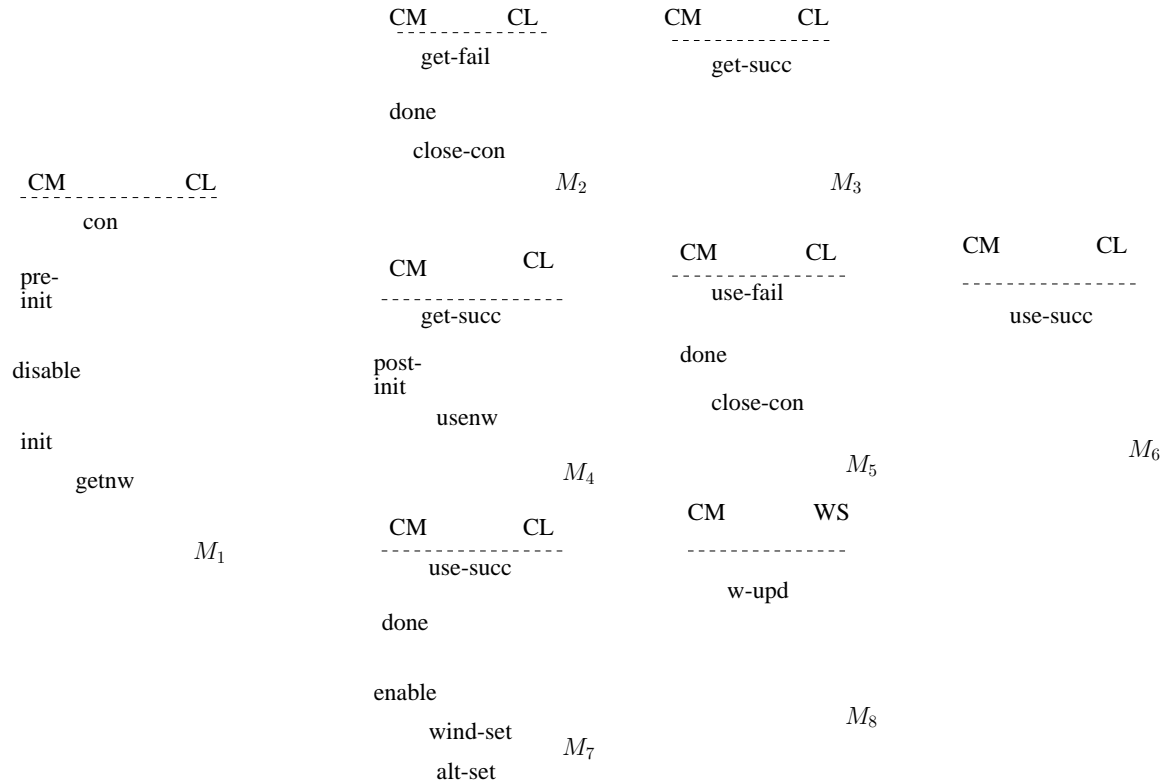


Figure 2. TMSCs for CTAS

ifying the normal behavior of the system (or its first phase), and *partially* specify the different error conditions (or the subsequent phase). Each of these partial specifications may then be elaborated (*refined*) in isolation following the same approach. The *compositionality* of our semantics ensures that when we replace a partial description (occurring within the global specification) with a refinement, we generate a refinement of the overall specification. This approach thus provides us with a robust way of building a large specification by focusing on only a part of it at a time.

We now illustrate this second approach using the Center TRACON Automation System.

Center TRACON Automation System (CTAS)

CTAS [1] is a set of tools designed to help air-traffic controllers manage complex air-traffic flows at large airports. The central process in the CTAS is the Communications Manager (CM). Various other processes act as clients and communicate with the CM through sockets. The communication begins with the client establishing a socket connection to the CM, and one aspect of the subsequent interaction is the notification of weather forecast updates to the client. For each update, a Weather Cycle is invoked; it begins with

a pre-initialization phase, and proceeds through initialization, post-initialization, updating and post-updating phases.

The interaction between CM and a generic *weather-aware client* CL will be specified incrementally via TMSCs in what follows, each step adding more detail to the previous one through an elaboration and refinement scheme. In the following description, each actual message and local-action in the system is followed by the abbreviation we use in the corresponding scenarios in Fig. 2. Also, for simplicity, we do not model socket communication, but use direct messages. The TMSCs we use are shown in Fig. 2.

First Stage. CL initiates the communication by sending a CONNECTION (*con*) message to CM (M_1). On reception, CM sets the Weather Cycle status and CL’s weather status to “pre-initializing”, (for simplicity, we model this by a common local-action *pre-init*). It then *disables* a manual weather control button so that no manual changes are made. Next, it sets the Weather Cycle status and the CL’s weather status to “initializing” (local-action *init*), and sends a CTAS-GET-NEW-WTHR (*getnw*) message to CL (M_1). This message contains CL’s initial weather state. On reception, CL will send back a message indicating what the GET-STATUS is: we use *get-fail* for failure and *get-succ* for success. In case of failure, CM sets the Weather Cycle status

to “done” and sends a CLOSE-CONNECTION (*close-con*) message to CM, which effectively terminates the communication (M_2). In case of success, the interaction proceeds to the next phase. We do not model this now, but indicate it through the partial scenario M_3 . Our initial specification is thus given by the TMS expression

$$S_1 = M_1; (M_2 \mp M_3)$$

Second Stage. We now specify the behavior once the first stage has successfully completed. CM sets the Weather Cycle status and CL’S weather status to “post-initializing” (*post-init*), and sends a CTAS-USE-NEW-WEATHER (*usenw*) to CL (M_4). On reception of this message, CL responds with a message indicating what the USE-STATUS is: we represent failure by *use-fail* and success by *use-succ*. As before, in case of failure, the Weather Cycle status is set to *done*, and communication is terminated (M_5); otherwise, the partial scenario (M_6) points to possible future communication. The behavior at this stage is then given by

$$R_1 = M_4; (M_5 \mp M_6)$$

R_1 represents an elaboration of the partial scenario M_3 , and it may be shown that $M_3 \sqsubseteq_{\text{must}} R_1$. We substitute R_1 for M_3 in S_1 to obtain

$$S_2 = M_1; (M_2 \mp R_1)$$

as our new specification. The compositionality of our semantics ensures that $S_1 \sqsubseteq_{\text{must}} S_2$.

Third Stage. If CM receives a *use-succ* message, then it indicates the successful completion of the post-initialization phase. CM responds by setting both the Weather Cycle status and CL’s weather status to *done*, enables the manual weather control button, and sends a GROUND-WIND-SETTING (*wind-set*) message and a ALTIMETER-SETTING (*alt-set*) message to CL, to convey the client’s initial surface-winds and altimeter readings (M_7). The next phase (“updating”) would start only when CM receives a weather update (*w-upd*) from a weather source (represented here by the generic component WS). CM would then need to convey the new weather information to CL; this may be done by subsequently extending the partial scenario (M_8) as appropriate, following the same approach. We have

$$R_2 = M_7; M_8$$

as an elaboration of M_6 . We now get

$$R'_1 = M_4; (M_5 \mp R_2)$$

as a refinement of R_1 , and $S_3 = M_1; (M_2 \mp R'_1)$, as our new specification. We would thus get a refinement sequence

$S_1 \sqsubseteq_{\text{must}} S_2 \sqsubseteq_{\text{must}} S_3 \dots$ as we build the specification in a stepwise manner, extending the behavior of the system one phase at a time.

The above example illustrates a number of useful features of our approach. Firstly, if we replace R'_1 within S_3 by the the corresponding TMS expression (in terms of M_4 , M_5 , M_7 and M_8), we see that the overall specification increases in complexity with each step. This indicates that the final specification, once all the phases have been described, is going to be quite complex and large, and thus it would have been difficult to describe the complete system behavior all at once. This naturally calls for an elaboration and refinement scheme, which the TMS framework provides. Secondly, a stepwise development method like ours reduces the risk of errors being introduced into the specification. Moreover, since the different stages in the above example mirrors the different phases of the system, if we wish to change the behavior in a particular phase, it is immediately clear which part of the specification needs to be modified. Finally, a modification of any step only means a different refinement of the previous step, not an entirely new specification having no relationship to the original refinement sequence. This leads to specifications that are easy to maintain.

4 Conclusions

In this paper we have argued for a scenario-based language that is more flexible than traditional MSCs but has a sound semantic underpinning. We have then presented the TMS formalism as such an alternative for scenario-based specification. The argument is based on the ability of TMSs to capture partial and conditional scenarios, the simplicity of TMS syntax, the careful choice of a suite of operators to build structured specifications, a robust semantic model based on acceptance trees, and a refinement notion that allows a systematic development of specifications. We have also illustrated a particular specification methodology supported naturally by TMSs using a specification of part of the Center TRACON system as an example, and discussed the benefits of this approach.

For future work, we would like to investigate the generation of state-machine based models for individual instances from scenario-based TMS specifications, and also consider how state-based system descriptions that are also equipped with an acceptance-tree semantics may be integrated with TMSs into single, heterogeneous, yet mathematically coherent, specifications.

References

- [1] Center-tracon automation system (ctas) for air traffic control. URL: <http://ctas.arc.nasa.gov/>.

- [2] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
- [3] T. Alspaugh and A. Antón. Scenario networks: A case study of the enhanced messaging system. *Seventh International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'01)*, 2001.
- [4] T. Alspaugh and A. Antón. Scenario networks for software specification and scenario management. Technical Report TR-2001-15, North Carolina State University, 2001.
- [5] R. Alur and M. Yannakakis. Model checking of message sequence charts. *Proceedings of the Tenth International Conference on Concurrency Theory*, LNCS 1664, Springer:82–97, 1999.
- [6] K. M. Chandi and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM Transactions of Computer Systems*, 3(1):63–75, Feb. 1985.
- [7] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
- [8] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, Jan. 2002.
- [9] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
- [10] G. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
- [11] M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
- [12] I. Kruger. Distributed system design with message sequence charts. *PhD Thesis, Technical University of Munich*, 2000.
- [13] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
- [14] R. Cleaveland and S. Sims. The ncsu concurrency workbench. *Computer Aided Verification (CAV)*, 1996, LNCS volume 1102:394–397.
- [15] M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, Eindhoven University of Technology*, 1998.
- [16] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *Proceedings of ACM SIGSOFT 2002, 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 167–176.
- [17] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. *Proceedings of ACM SIGSOFT 2002, 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 109–118.