Executable Requirements Specifications Using Triggered Message Sequence Charts

Bikram Sengupta[1] and Rance Cleaveland[1]

[1] IBM India Research Laboratory
Block 1, Indian Institute of Technology
Hauz Khas, New Delhi - 110016
bsengupt@in.ibm.com
[2] Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
rance@cs.sunysb.edu

Abstract. Triggered Message Sequence Charts (TMSCs) are a scenario-based visual formalism for early stage requirements specifications of distributed systems. In this paper, we present a formal operational semantics for TMSCs that allow the simulation of TMSC system descriptions, so that errors and inconsistencies in specification may be detected early on. The semantics is defined in terms of Structured Operational Semantics (SOS) rules that guide the step-wise execution of TMSC specifications. We also consider the equivalence of this semantics and the TMSC denotational semantics that has been presented in previous work.

1 Introduction

Triggered Message Sequence Charts (TMSCs) have been proposed in [13] as an extension of the well-known visual formalism of Message Sequence Charts (MSCs) [1, 11]. Like MSCs, TMSCs describe system *scenarios* in terms of exchange of messages and execution of local-actions that a set of processes (or *instances*) may engage in as they execute. Unlike MSCs, however, TMSCs can specify *conditional scenarios*, which represent requirements that constrain system behavior only when certain "triggering behaviors", are observed; and *partial scenarios*, which permit users to leave aspects of system behavior unspecified. The theory is also equipped with a *refinement* ordering (based on the must preorder of [8]) that determines when one specification is a "correct elaboration of" another, by correctly adhering to prescriptive and conditional-scenario constraints and properly "filling in" unspecified behavior in partial scenarios [12].

TMSCs are thus well-suited for early-stage behavioral descriptions, which may be subject to refinement and elaboration as design proceeds. Accordingly, practitioners will find it useful to be able to simulate the behavior of TMSC-based system descriptions. This will allow early detection of inconsistencies and aberrant scenarios, which may otherwise be very expensive to fix once the system has been constructed. However, as evident in the technical development of [13], the formal semantics of TMSCs, which translates TMSC specifications to *acceptance trees* [8], is *declarative* in nature: it provides a precise definition of *what* the acceptance tree should be for a given TMSC

specification, without describing *how* it may be constructed step-by-step. Thus this semantics does not allow ready simulation of TMSC specifications.

The goal of this paper is to present an alternative (but equivalent) semantics of TMSCs, which is *operational* in nature. This semantics can serve as the basis for tool-support, and help build executable models of TMSC specifications for early simulation. The rest of the paper is organized as follows: in the next section, we introduce TMSCs and the main ideas behind the acceptance tree semantic model. In Section 3, we explain the operational behavior of single instances in a TMSC; this is then extended to enable simulation of complete, single TMSCs. Section 4 outlines how the ideas may be extended to define the executable behavior of structured TMSC specifications. In Section 5, we discuss the equivalence of the declarative and operational semantics of TMSCs. Section 6 considers tool support. Section 7 presents related work, while Section 8 contains conclusions and directions for future research.

2 Background

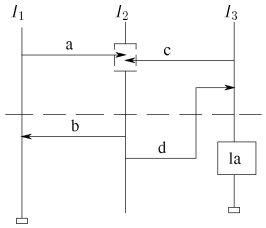

Fig. 1. An Example TMSC

Triggered Message Sequence Charts: Graphically, we represent TMSCs as in Fig. 1. There are two new features in the visual syntax of TMSCs when compared to traditional MSCs. The first is the dashed horizontal line running through the instances, which partitions the sequence of events on an instance's axis into two subsequences: the first, located above the line, constitutes the instance's *trigger*, and the second, below the line, constitutes its *action*. This partition, in effect, forms the basis of a *conditional scenario*: for each instance, the execution of the action is conditional on the occurrence of the trigger. In other words, the behavior of the instance is constrained to its action *only* when it has executed its trigger; otherwise, there are no restrictions. The second new feature in a TMSC is the presence/absence of a small bar at the foot of each instance. The presence of such a bar (as in instance I_1 in Fig 1) indicates that the instance cannot proceed beyond this point in the TMSC, 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. Such a scenario is thus *partial*, and may be extended in future.

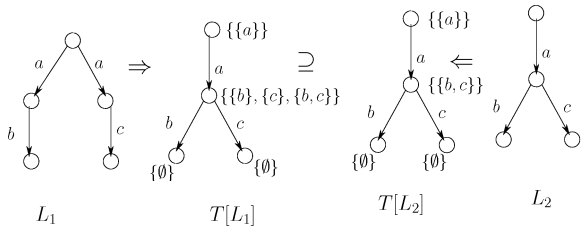

Fig. 2. The *must* preorder: $L_1 \sqsubseteq_{\text{must}} L_2$

The TMSC in 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".

Acceptance Trees: Acceptance trees and the must preorder arise in the theory of testing of concurrent processes given in [8]. In this theory, tests, which may also be thought of as processes that are capable of reporting "success", interact with a process under test. When processes and tests are nondeterministic, a process may be capable both of passing and failing a test, depending on how nondeterministic choices are resolved. A process *must pass* a test if, regardless of how such choices are made, the process passes the test. One process refines another with respect to the must preorder if it must pass every test that the less refined process must. We use an alternative characterization of the must preorder that is given in terms of the processes themselves, rather than tests. Specifically, the must pre-order may be characterized in terms of *acceptance sets* (that are a measure of the non-determinism of a process) when the processes are given as Labeled Transitions Systems (LTSs).

Definition 1. *Let $\mathcal{P} = \langle P, E, \longrightarrow, p_I \rangle$ be a Labeled Transition System (LTS),where P is a set of states, E a set of events, $\longrightarrow \subseteq P \times E \times P$ the transition relation, and $p_I \in P$ the start state. Then, for $p \in P$ and $w \in E^*$. the following may be defined.*

$$L(\mathcal{P}) = \{w \in E^* \mid \exists p' \in P.p_I \xrightarrow{w} p'\} \text{ (Language)}$$
$$S_{\mathcal{P}}(p) = \{a \mid \exists p' \in P.p \xrightarrow{a} p'\} \text{ (Successors)}$$
$$Acc(\mathcal{P}, w) = \{S_{\mathcal{P}}(p') \mid p_I \xrightarrow{w} p'\} \text{ (Acceptance set)}$$

We now define a *saturation operator*, sat, on acceptance sets. Let $\mathcal{A} \subseteq 2^E$; then $sat(\mathcal{A})$ is the least set satisfying:

1. $\mathcal{A} \subseteq sat(\mathcal{A})$.
2. If $A_1, A_2 \in sat(\mathcal{A})$ then $A_1 \cup A_2 \in sat(\mathcal{A})$.
3. If $A_1, A_2 \in sat(\mathcal{A})$ and $A_1 \subseteq A \subseteq A_2$, then $A \in sat(\mathcal{A})$.

The alternative characterization of $\sqsubseteq_{\text{must}}$ can now be given as follows [8].

Theorem 1. *Let $\mathcal{P}_1 = \langle P_1, E_1, \longrightarrow_1, p_{I_1} \rangle$ and $\mathcal{P}_2 = \langle P_2, E_2, \longrightarrow_2, p_{I_2} \rangle$ be two LTSs, and let $E = E_1 \cup E_2$. Then $\mathcal{P}_1 \sqsubseteq_{must} \mathcal{P}_2$ iff for all $w \in E^*$, $sat(Acc(\mathcal{P}_1, w)) \supseteq sat(Acc(\mathcal{P}_2, w))$.*

Intuitively, $\mathcal{P}_2$ refines $\mathcal{P}_1$ if it has "less nondeterminism." This alternative characterization forms the basis for representing processes as *acceptance trees* [8], which map sequences of events to acceptance sets.

Definition 2. *Let $\mathcal{E}$ be a finite set of events. Then an* acceptance tree *T is a function in $\mathcal{E}^* \to 2^{2^{\mathcal{E}}}$ satisfying:*

1. *For any $w \in \mathcal{E}^*$, $sat(T(w)) = T(w)$.*
2. *For any $w, w' \in \mathcal{E}^*$, if $T(w) = \emptyset$ then $T(w \cdot w') = \emptyset$.*
3. *For any $w \in \mathcal{E}^*$, $e \in \mathcal{E}$, $T(w \cdot e) \neq \emptyset$ iff there exists $A \in T(w)$ such that $e \in A$.*

We say that $T_1 \supseteq T_2$ if for all $w \in \mathcal{E}^*$, $T_1(w) \supseteq T_2(w)$.

For any LTS $\mathcal{P}$ there is an immediate way to construct an acceptance tree $T\lceil\mathcal{P}\rceil$: $T\lceil\mathcal{P}\rceil(w) = sat(Acc(\mathcal{P}, w))$. It immediately follows that $\mathcal{P}_1 \sqsubseteq_{\text{must}} \mathcal{P}_2$ if and only if $T\lceil\mathcal{P}_1\rceil \supseteq T\lceil\mathcal{P}_2\rceil$. For example, in Fig. 2, $L_1 \sqsubseteq_{\text{must}} L_2$ because $T\lceil L_1\rceil \supseteq T\lceil L_2\rceil$.

3 Executing Single TMSCs

Having introduced TMSCs, let us now consider how they may be executed. The operational behavior of single TMSCs will be described by Plotkin-style [7] Structured Operational Semantics (SOS) rules. A SOS rule of the form $\mathsf{R}.\ \dfrac{l}{p \xrightarrow{a} p'}$ denotes a rule R which describes the operational behavior of a process p that can perform an event a to evolve to p' provided each predicate in the list l is true. Since a TMSC consists of a set of *instances* executing asynchronously, we begin by defining SOS rules to model the operational behavior of individual instances in a TMSC.

Notational Convention We fix finite sets $\mathbb{I}$, $\mathbb{M}$ and $\mathbb{A}$ as the set of all instances, message types and local action names, respectively. We write $\mathbb{R} = \{\mathsf{in}(I, J, m) \mid I, J \in \mathbb{I}, m \in \mathbb{M}\}$ for the set of all receive events, and similarly define $\mathbb{S} = \{\mathsf{out}(I, J, m) \mid I, J \in \mathbb{I}, m \in \mathbb{M}\}$ as the set of all send events; in each case, I denotes the sender and J the receiver of message m. We use $\mathbb{L} = \{\mathsf{loc}(I, \ell) \mid I \in \mathbb{I}, \ell \in \mathbb{A}\}$ as the set of all local actions. Our semantics also uses events of form $\mathsf{end}(I)$, where $I \in \mathbb{I}$, which instances emit when they terminate, and "potential events" of form $\mathsf{wait}(r)$, where $r \in \mathbb{R}$, to denote that an instance is capable of performing r once the corresponding send event occurs. $\mathbb{T}$ and $\mathbb{W}$ denote the set of all end events and wait events respectively.

An instance in a TMSC M is initially specified by the term $Ins(I, S, t, p, q)$, where I is the name of the instance, S is the set of all events it may possibly perform, t indicates if the instance terminates on performing its action or not (with t = "yes" if the instance terminates, "no" otherwise), and p and q are sequences of events that constitute respectively, the trigger and action of I in M. In addition to the Ins form, I may also be specified by three other instance terms in course of its execution:(i) it may assume the form $Nondet\_ins(I, S)$, when its behavior is completely non-deterministic (e.g. when the trigger has been violated, or I has performed the trigger, followed by the action, and I is not required to terminate subsequently), (ii) it may be of the form $Term\_ins(I)$, after it has terminated, or (iii) it may move to the form $Restr\_ins(\{e\}, I_t)$, (where only

event e is enabled, and I_t is an instance term) from the Ins and $Nondet\_ins$ forms, once it has non-deterministically chosen to perform event e, from the set of possible events. As we will see, these instance forms are necessary to capture the evolution of an instance in a conditional/partial scenario.

Table 1. Operational semantics for Instances

I1. $$\frac{p \neq nil, e \in possible\_ev(S, me)}{me : Ins(I, S, t, p, q) \xrightarrow{\tau} Restr\_ins(\{e\}, Ins(I, S, t, p, q))}$$

I2. $$\frac{e = \mathsf{wait}(r), r \in me, p = r \cdot p'}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{r} Ins(I, S, t, p', q)}$$

I3. $$\frac{e = \mathsf{wait}(r), r \in me, p = r' \cdot p', r' \neq r}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{r} Nondet\_ins(I, S)}$$

I4. $$\frac{e = \mathsf{wait}(r), r \notin me}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{e} Restr\_ins(\{e\}, Ins(I, S, t, p, q))}$$

I5. $$\frac{e \in \mathbb{R} \cup \mathbb{S} \cup \mathbb{L}, p = e \cdot p'}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{e} Ins(I, S, t, p', q)}$$

I6. $$\frac{e \in \mathbb{R} \cup \mathbb{S} \cup \mathbb{L}, p = e' \cdot p', e \neq e'}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{e} Nondet\_ins(I, S)}$$

I7. $$\frac{e \in \mathbb{T}}{me : Restr\_ins(\{e\}, Ins(I, S, t, p, q)) \xrightarrow{e} Term\_ins(I)}$$

I8. $$\frac{e \in possible\_ev(S, me)}{me : Nondet\_ins(I, S) \xrightarrow{\tau} Restr\_ins(\{e\}, Nondet\_ins(I, S))}$$

3.1 Operational Semantics for an Instance

We will now present SOS rules that govern the operational behavior of an instance in a TMSC. These rules, defined in Tables 1 and 2, assume the existence of a message environment me which represents the set of enabled in events; an instance J may perform an event $\mathsf{in}(I, J, m)$, only if $\mathsf{in}(I, J, m) \in me$. Thus me corresponds to messages that have been sent but not yet received.

In **I1** (Table 1), instance I begins in its initial state $Ins(I, S, t, p, q)$, and as long as its trigger has not been completely satisfied (i.e. $p \neq nil$), I may non-deterministically

Table 2. Operational semantics for Instances (Cont.)

I9. $$\frac{e = \mathsf{wait}(r), r \in me}{me : Restr\_ins(\{e\}, Nondet\_ins(I,S)) \xrightarrow{\tau} Nondet\_ins(I,S)}$$

I10. $$\frac{e = \mathsf{wait}(r), r \notin me}{me : Restr\_ins(\{e\}, Nondet\_ins(I,S)) \xrightarrow{e} Restr\_ins(\{e\}, Nondet\_ins(I,S))}$$

I11. $$\frac{e \in \mathbb{R} \cup \mathbb{S} \cup \mathbb{L}}{me : Restr\_ins(\{e\}, Nondet\_ins(I,S)) \xrightarrow{e} Nondet\_ins(I,S)}$$

I12. $$\frac{e \in \mathbb{T}}{me : Restr\_ins(\{e\}, Nondet\_ins(I,S) \xrightarrow{e} Term\_ins(I)}$$

I13. $$\frac{(e \in \mathbb{S} \cup \mathbb{L}) \vee (e \in \mathbb{R} \wedge e \in me)}{me : Ins(I,S,t,nil,e.q) \xrightarrow{e} Ins(I,S,t,p,q)}$$

I14. $$\frac{e \in \mathbb{R} \wedge e \notin me}{me : Ins(I,S,t,nil,e \cdot q) \xrightarrow{\mathsf{wait}(e)} Ins(I,S,t,nil,e \cdot q)}$$

I15. $$\frac{t = yes}{Ins(I,S,t,nil,nil) \xrightarrow{\mathsf{end}(I)} Term\_ins(I)}$$

I16. $$\frac{t = no}{Ins(I,S,t,nil,nil) \xrightarrow{\tau} Nondet\_ins(I,S)}$$

choose to perform any event in S that is allowed by me; $possible\_ev(S, me)$ returns the set of such events and may be defined as:

$$\begin{aligned} possible\_ev(S, me) = \{e \mid e \in S \wedge (e \in \mathbb{S} \cup \mathbb{L} \cup \mathbb{T} \\ \vee (e \in \mathbb{R} \wedge e \in me))\} \\ \cup \{\mathsf{wait}(e) \mid (e \in S \cap \mathbb{R}) \wedge e \notin me\} \end{aligned}$$

Thus, an out, loc or end event is always possible. An in event is only possible if it is in me, otherwise the corresponding wait event is possible. For each such possible event e, I non-deterministically moves to a restricted mode represented by $Restr\_ins(\{e\}, Ins(I,S,t,p,q))$ (**I1**).

If e is a $\mathsf{wait}(r)$ event and r subsequently becomes enabled (because the corresponding message has been sent), then there are two possibilities: (i) r is the next event in the trigger, and the instance performs r, and evolves to the mode $Ins(I,S,t,p',q)$, where p' represents the suffix of the trigger that is left to be performed; this is shown in **I2** (ii) r is not the next event in the trigger, in which case performing r violates the trig-

ger, and the instance moves to a totally unconstrained (non-deterministic) mode given by $Nondet\_ins(I, S)$ (**I3**). If e is a wait(r) event and r is currently not allowed, then the instance stays in the same mode as indicated by **I4**. (Note that wait events are only *potential* events, not actual ones, hence, they do not cause a change of state).

If e is an in, out or loc event, and is also the next event in the trigger, then the instance moves to a mode $Ins(I, S, t, p', q)$ (**I5**), where p' represents the suffix of the trigger yet to be performed, as in **I2**. If e is not the next event in the trigger, then performing e causes the instance to move to the (unconstrained) mode $Nondet\_ins(I, S)$ (**I6**). If the end is enabled in the restricted mode, then the instance terminates by performing e, as it moves to the mode $Term\_ins(I)$ from which no transitions are enabled (**I7**).

If I is in the mode $Nondet\_ins(I, S)$, then it may non-deterministically choose to perform any event e that is possible, and move to a restricted mode where e is enabled. This is shown in **I8**. If e is a wait(r) event and r gets enabled, then r is performed, and the instance returns to the non-deterministic state (**I9**, Table 2); however, if r is not allowed, then the instance *waits* in the same mode (**I10**). If e is an in, out or loc event, then it may be immediately performed, and the instance returns to the non-deterministic mode (**I11**). If e is an end event, then the instance terminates on performing e (**I12**).

I13, I14, I15 and **I16** describe the behavior of I once it has satisfied its trigger. If the next event e in the action is enabled, then it is performed (**I13**), else the corresponding wait event is performed (**I14**). Once the action has been completely executed, the parameter t comes into play. If I has to terminate immediately ($t = yes$), then I performs the end(I) event (**I15**), else it moves to the nondeterministic mode (**I16**).

3.2 A single TMSC M = me:IL

A single TMSC M consists of a set of instance terms IL, having a common message environment me, which represents the set of enabled receive events. Initially me is empty. Table 3 presents the execution behavior of TMSC M in terms of the execution of its instances. In rule **M1**, we first compute the set of instance names that are explicitly mentioned in M, using the function $Ins\_name\_list(IL) = \{Ins\_name(I_t) \mid I_t \in IL\}$, where $Ins\_name(I_t)$ returns the instance name of I_t. We then get $T = \mathbb{I} - Ins\_name\_list(IL)$ as the set of instances which are not explicitly mentioned in the TMSC M. According to the TMSC semantics [13], these instances are assumed to terminate immediately. As rule **M1** shows, IL is annotated with T, to record this set of instances.

As long as an instance term I_t in IL is *unstable* i.e. may perform a τ transition, M is also unstable: if I_t evolves to I'_t on performing τ, M also performs τ and moves to a new state where me remains unchanged, but within IL, I_t is replaced by I'_t (**M2**). The $update\_msc$ function may be written simply as $update\_msc(IL, I_t, I'_t) = (IL - I_t) \cup I'_t$.

Once all the instances have resolved their internal non-determinism and stabilized ($I_t \stackrel{\tau}{\not\longrightarrow}$ is taken to mean I_t is unable to perform a transition labeled by τ), and an instance term I_p may perform an event e to become I'_p, M may also perform e, as shown in **M3**. In this case, me may get updated (if e is an in or out), and the instance

Table 3. Operational semantics for TMSC M = me: IL

M1.	$\dfrac{T = \mathbb{I} - Ins\_name\_list(IL)}{me : IL \xrightarrow{\tau} IL_T}$
M2.	$\dfrac{I_t \in IL, me : I_t \xrightarrow{\tau} I_t'}{me : IL_T \xrightarrow{\tau} me : update\_msc(IL, I_t, I_t')_T}$
M3.	$\dfrac{\forall I_t \in IL, me : I_t \not\xrightarrow{\tau}, \exists I_p \in IL.me : I_p \xrightarrow{e} I_p'}{me : IL_T \xrightarrow{e} update\_env(me, e) : update\_msc(IL, I_p, I_p')_T}$
M4.	$\dfrac{\forall I_t \in IL, me : I_t \not\xrightarrow{\tau}, J \in T}{me : IL_T \xrightarrow{\mathsf{end}(J)} IL_{T-\{J\}}}$

term list IL now contains I_p' in place of I_p. The $update\_env$ function may be written as

$$\begin{aligned} update\_env(me, e) &= me \cup \mathsf{in}(J, I, m), && \text{if } e = \mathsf{out}(J, I, m) \\ &= me - \mathsf{in}(I, J, m), && \text{if } e = \mathsf{in}(I, J, m) \\ &= me, && \text{otherwise} \end{aligned}$$

Thus, if e is an out event, the corresponding in event becomes enabled in me, whereas if e is an in event, it is removed from me upon execution. Finally, **M4** indicates that when the system reaches a stable state, any instance not explicitly mentioned in M may terminate immediately; the parameter T is then updated to reflect all remaining instances that are still *active*.

4 TMSC Expressions

Single TMSCs can express single scenarios of systems. In order to provide capabilities for structured system specifications, the TMSC language also includes a suite of operators for assembling sub-specifications. The resulting terms, called *TMSC expressions*, are defined by the following grammar:

$S ::= M$		(single TMSC)
$\mid$	X	(variable)
$\mid$	$S_1 \parallel S_2$	(interleaving parallel composition)
$\mid$	$S_1 \mp S_2$	(delayed choice)
$\mid$	$S_1 \oplus S_2$	(internal choice)
$\mid$	$S_1; S_2$	(sequential composition)
$\mid$	$S_1 \wedge S_2$	(logical and)
$\mid$	$recX.S$	(recursive operator)

$S_1 \parallel S_2$ denotes the "interleaving" parallel composition of expressions S_1 and S_2: it allows the interleaving of events from S_1 and S_2 while the expressions execute independently. $S_1 \mp S_2$ 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. $S_1 \oplus S_2$ is the nondeterministic choice between S_1 and S_2; a successful refinement can choose either. $S_1; S_2$ denotes the "instance-level" (asynchronous) sequential composition [11]; $S_1 \wedge S_2$ represents logical conjunction, and is primarily used in our framework to weave together individual constraints on system behavior Finally, the recursive operator $recX$ allows us to model infinite behavior of processes, where a new execution cycle starts whenever there is a reference to the variable used in the recursive definition (say X).

4.1 Executing TMSC Expressions

The operational behavior of TMSC expressions may also be defined in terms of SOS rules. Due to space constraints, we are unable to provide the entire semantics here. However, to provide some illustrative examples, we present the semantics of the two choice operators, $\oplus$ and $\mp$. This will also help bring out the difference between these two operators.

$S_1 \oplus S_2$: The SOS rules for the TMSC expression $S_1 \oplus S_2$ are shown in Table 4. The rules simply state that $S_1 \oplus S_2$ may non-deterministically (internally) choose between S_1 (**O1**) or S_2 (**O2**).

Table 4. Operational semantics for $S = S_1 \oplus S_2$

O1.	$\dfrac{}{S_1 \oplus S_2 \xrightarrow{\tau} S_1}$	O2.	$\dfrac{}{S_1 \oplus S_2 \xrightarrow{\tau} S_2}$

$S_1 \mp S_2$: In $S_1 \mp S_2$, the choice between S_1 and S_2 is delayed till a point is reached where their behaviors differ; at that point, a choice is made. Thus initially, S_1 and S_2 are allowed to resolve their internal non-determinism through sequences of τ transitions, till they both reach a *stable* state (**D1** and **D2** in Table 5); if S_1 and S_2 can then both perform an event a to evolve to S_1' and S_2' respectively, then $S_1 \mp S_2$ can also make an a transition to the state $S_1' \mp S_2'$ (**D5**). The choice between S_1 and S_2 is thus delayed. However, if S_1 (S_2) can make an a transition and S_2 (S_1) cannot, then $S_1 \mp S_2$ may perform an a transition thereby resolving the choice in favor of S_1 (S_2). This is shown in **D3** (and **D4**).

Table 5. Operational semantics for $S = S_1 \mp S_2$

D1. $$\frac{S_1 \xrightarrow{\tau} S_1'}{S_1 \mp S_2 \xrightarrow{\tau} S_1' \mp S_2}$$

D2. $$\frac{S_2 \xrightarrow{\tau} S_2'}{S_1 \mp S_2 \xrightarrow{\tau} S_1 \mp S_2'}$$

D3. $$\frac{S_1 \xrightarrow{a} S_1', S_2 \not\xrightarrow{a}, S_1 \not\xrightarrow{\tau}, S_2 \not\xrightarrow{\tau}}{S_1 \mp S_2 \xrightarrow{a} S_1'}$$

D4. $$\frac{S_2 \xrightarrow{a} S_2', S_1 \not\xrightarrow{a}, S_1 \not\xrightarrow{\tau}, S_2 \not\xrightarrow{\tau}}{S_1 \mp S_2 \xrightarrow{a} S_2'}$$

D5. $$\frac{S_1 \xrightarrow{a} S_1', S_2 \xrightarrow{a} S_2', S_1 \not\xrightarrow{\tau}, S_2 \not\xrightarrow{\tau}}{S_1 \xrightarrow{a} S_1' \mp S_2'}$$

5 Operational vs. Declarative Semantics

The TMSC declarative semantics is given as a set of equations [13] that define what the acceptance tree should be for a given TMSC expression. For a single TMSC M, this computation considers if an execution sequence w satisfies the triggers of the instances, and then either selects the next events from the action sequences (if an instance's trigger is complete), or non-deterministically chooses any enabled event (since there are no constraints on the instance's behavior). Acceptance tree computation for a TMSC expression S then proceeds by induction on the structure of S. For example, $T[S_1 \oplus S_2](w)$ = $sat(T[S_1](w) \cup T[S_2](w))$. The interested reader is referred to [13] for more details.

Although the declarative semantics gives a precise definition of the acceptance tree for a TMSC expression, it does not support ready simulation through the construction of executable models. This motivates the need for the operational semantics we presented in this paper. Given a TMSC expression S, we may apply the SOS rules repeatedly to generate a behavioral model of S in the form of a labeled transition system, from which an acceptance tree may be extracted, as outlined in Section 2. We make a reasonable assumption that variables inside recursive TMSC expressions are *guarded* i.e. preceded by a sub-expression in which no constituent instance may terminate immediately. This ensures that a TMSC expression may not be infinitely unrolled without making any actual progress. The aim is to constrain the TMSC language to those expressions S for which $LTS(S)$ is actually constructible using the TMSC SOS rules.

We will now relate the TMSC operational semantics with the declarative semantics. To distinguish the two semantics, for a TMSC expression S, we will denote by $T^D[S]$, the acceptance tree of S that is derived from the declarative semantics, and by $T^O[S]$, the acceptance tree that is derived from the transition system of S, generated by the operational semantics, i.e. SOS rules. The following theorem captures the equivalence of the two semantics. The result may be proved by induction on the structure of S; for brevity, we do not include the proof details here.

Theorem 2. *Let S be a TMSC expression such that $LTS(S)$ is constructible. Then, for any execution sequence w, $T^D[S](w) = T^O[S](w)$.*

The above theorem implies that the executable models we build using the TMSC SOS rules described in this paper are "correct" i.e. they conform to the original (declarative) TMSC semantics. Note that we may define a refinement notion on TMSC expressions, based on the *must* preorder, in terms of the LTSs obtained through the operational semantics as follows:

Definition 3. *Let S_1 and S_2 be TMSC expressions such that $LTS(S_1)$ and $LTS(S_2)$ are constructible. Then, $S_1 \sqsubseteq_{must} S_2$ iff $T^O[S_1] \supseteq T^O[S_2]$.*

6 Tool Support for TMSCs

The TMSC operational semantics outlined in this paper provides the basis for automated analysis of TMSC expressions through the TRIM tool. TRIM provides a simulator for executing TMSC expressions (according to the SOS rules) and also includes routines for checking refinement ordering between TMSC expressions and for returning diagnostic information when refinement fails to hold. TRIM is built on top of the Concurrency Workbench (CWB-NC) [10] an easy-to-retarget verification tool for finite-state sysems. A brief description of the TRIM architecture and some of the implementation considerations may be found in [4].

7 Related Work

The formal MSC language appears in a recommendation of the ITU [1]. An executable semantics for this language has been defined in [11] in a process algebraic setting, where the system behavior is interpreted in terms of SOS rules. However, basic MSCs in the ITU standard are expressively weak, offering only a visual partial ordering of events. The technical development in [1, 11] does not provide a natural way for expressing conditional/partial behavior and for the step-wise refinement of behavior. TMSCs were motivated by a need to enhance the MSC language along these directions. Accordingly, our operational semantics had to account for aditional considerations that arise from the specification of conditional and partial behavior, as also new structuring constructs (like $\oplus$ and $\wedge$) that TMSCs support. Note that [5], [9] have also proposed MSC extensions that support variations of trigger/action-like behavior as captured by TMSCs.

Several tools have been developed to support the use of scenarios in practice. MESA [3] allows certain properties, such as process divergence to be efficiently checked on MSCs. UBET ([2]) detects potential race conditions and timing violations in an MSC, and also provides automatic test case generation over HMSCs. The *play-in/play-out* approach of [6] is based on LSCs and has been implemented via a tool called the *play engine*. [15] shows how Constraint Logic Programming (CLP) may be used to support symbolic execution of LSC requirements. LTSA-MSC [14] supports synthesis of behavior models from MSC-based specifications and implied-scenario detection.

8 Conclusions

In this paper, we presented an operational semantics of the TMSC language in terms of SOS rules. This semantics complements the denotational TMSC semantics presented in earlier work [13] and provides the basis for simulation and analysis of TMSC-based specifications. We also considered the relation between the operational and denotational semantics of TMSCs, and discussed related work. In future, we intend to study how to generate test cases from TMSC specifications.

References

1. Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
2. R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
3. H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. *Proc. of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, LNCS volume 1384:118–135.
4. B.Sengupta and R.Cleaveland. TRIM: A tool for triggered message sequence charts. *Proceedings of 15TH Computer Aided Verification Conference (CAV'03) (tool paper)*, 2003.
5. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
6. D.Harel and R.Marelly. Specifying and executing behavioral requirements: The play-in/ play-out approach. *Software and System Modeling (SoSym)*, 2003.
7. G.Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
8. M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
9. I. Kruger. Distributed system design with message sequence charts. *PhD Thesis, Technical University of Munich*, 2000.
10. R.Cleaveland and S.Sims. The ncsu concurrency workbench. *Computer Aided Verification (CAV), 1996*, LNCS volume 1102:394–397.
11. M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, Eindhoven University of Technology*, 1998.
12. B. Sengupta and R. Cleaveland. Refinement-based requirements modeling using triggered message sequence charts. 11th IEEE Int'l Requirements Engineering Conference, 2003.
13. B. Sengupta and R. Cleaveland. Triggered message sequence charts. *ACM SIGSOFT 2002, 10th Int'l Symposium on the Foundations of Software Engineering (FSE-10)*, pages 167–176.
14. J.Kramer S.Uchitel and J.Magee. Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios. *TACAS'03*.
15. T.Wang, A.Roychoudhury, R.Yap, and S.C.Choudhary. Symbolic execution of behavioral requirements. *PADL 2004*, LNCS vol. 3057.