# Triggered Message Sequence Charts*

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 propose an extension to Message Sequence Charts called *Triggered Message Sequence Charts* (TMSCs) that are intended to capture system specifications involving nondeterminism in the form of *conditional scenarios*. The visual syntax of TMSCs closely resembles that of MSCs; the semantics allows us to translate a TMSC specification into a framework that supports a notion of refinement based on Denicola's and Hennessy's must preorder. A simple but non-trivial example illustrates the utility of our extension to MSCs.

## Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.10 [**Software Engineering**]: Design

## General Terms

Design, Languages, Verification

## Keywords

Message Sequence Charts; scenario-based requirements; refinement orderings

## 1. INTRODUCTION

Message Sequence Charts (MSCs) [1, 13] are a popular visual formalism for scenario-based specifications of distributed software systems. An MSC depicts one exchange of messages a set of processes are expected to engage in as they execute; sets of MSCs are then used as specifications that eventual system implementations should satisfy.

Semantically, as typified in [4, 9, 10, 12, 13], MSC system specifications are interpreted as *deterministic* and *complete*: mathematically, a set of MSCs completely characterizes all the behavior of correct implementations by requiring that they exhibit exactly the execution sequences the MSCs exhibit. However, scenario-based specification notations like MSCs are also heavily used in the initial stages of system development, when many design-related issues are unresolved and not all eventual implementation scenarios may be known. Such early-stage specifications are thus subject to refinement and elaboration as design proceeds, and requirements that *constrain* rather than just *prescribe* eventual implementation behavior become very useful. For example, a specification may include an erroneous sequence of messages that could arise during system execution together with a recovery action to be taken if the error occurs. It may be possible to devise a more detailed specification that satisfies the original one by preventing the undesirable sequence of actions from occurring in the first place; in this case we would like to say that the new specification *refines* the original one. Such *conditional* specifications ("if the error happens, do this") and refinement-based reasoning are not supported in traditional MSCs.

The main goal of this work is to develop a *structured scenario-based* visual formalism that supports constraint-based and prescriptive system requirements as well as a mathematically precise notion of when one requirement refines another. To that end, we propose an extension to Message Sequence Charts called *Triggered Message Sequence Charts* (TMSCs), a set of operators to combine TMSCs, and a semantic theory that supports a notion of refinement based on the *must preorder* of [8]. The rest of the paper is organized as follows: Section 2 introduces TMSCs and TMSC expressions, along with an example TMSC specification, which we will use to illustrate our formalism. Section 3 contains mathematical preliminaries. The next section describes the semantics of single TMSCs and presents our notion of refinement, and the one following lifts these notions to TMSC expressions. The penultimate section revisits our example specification, and considers some practical refinements of it, to illustrate our theory. The last section contains conclusions and directions for future research.

## 2. TRIGGERED MSCS

Like MSCs, TMSCs describe system scenarios in terms of the sequences of atomic actions (message sends and receives, and local actions) that each parallel process (or *instance*) may engage in. The key novelty of TMSCs is that each instance's action sequence is partitioned into two subsequences: a *trigger* and an *action*. A TMSC scenario stipulates that in any system execution, if the sequence of events performed by an instance constituting its *trigger* occurs, then the subsequent behavior of the instance must include the sequence of events which constitute its *action*. In an implementation an instance is not required to display the behavior described by its trigger, but when it does so, its subsequent behavior is limited to its action. The notions of triggers and actions, when coupled with TMSC composition operators, enrich MSC-based specifications by offering several alternatives for eventual implementations.

*Research supported by ARO grants DAAD190110003 and DAAD190110019 and NSF grant CCR-0098037.


*SIGSOFT2002/FSE-10,* November 18–22,2002,Charleston, SC, USA.

**Related Work.** [9] proposes a similar trigger-like composition operator for joining MSCs. However, the semantic model in [9] is stream-based and inherently deterministic, and the composition operator in question is used to specify liveness properties. The same work 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. Live Sequence Charts (LSCs) [7] include *activation conditions* and *precharts*, which are similar to our notion of triggers; however 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. Thus, assuming that the activation condition/prechart of each universal LSC is actually satisfied in at least some run (the "healthiness check" of [7]), and no run vacuously satisfies all the LSCs by violating all the activation conditions/precharts, an LSC specification describes the intended behavior of the system more or less deterministically. Similarly, scenario networks [2, 3] use sequential and concurrent relationships between scenarios to describe allowable system behavior. Scenarios are labelled with pre- and post-conditions to determine which scenarios can occur next at any given point. However, the purpose of the conditions is to restrict the execution paths of the system to the desirable ones, rather than to offer choices (in the form of non-determinism) in the system behavior. That is, network scenarios are typically expected to occur in at least some runs of the system. In a TMSC specification, however, it may be perfectly "healthy" for a scenario not to arise in any run of an intended implementation, simply because it may have been overruled during system design in favor of some other scenario in the specification.

## 2.1 Syntax of TMSCs

**Visual syntax.** Graphically, we represent TMSCs as in Fig. 1. The partitioning of the sequence of events of an instance into the trigger and action sequences is indicated by a horizontal line running through the instances of the MSC. For each instance, the sequence of events above the line constitutes its trigger, while the sequence below the line constitutes its action. The TMSC in Fig. 1

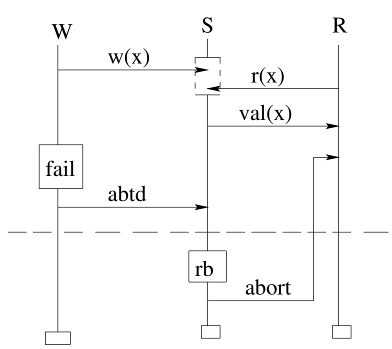


**Figure 1: An example TMSC.**

consists of three instances: a server $S$, a reader process $R$, and a writer process $W$. $S$ is intended to maintain a consistent copy of a variable $x$. $W$ can send a write request, $w(x)$, to $S$ indicating that variable $x$ should be written, while $R$ can send read requests in the form of the message $r(x)$. $S$ can receive the requests in either order (the dashed box at the beginning of $S$ is a *co-region*: co-regions allow incident actions to be ordered arbitrarily) and responds to the $r(x)$ with a message of form $val(x)$ containing the value of $x$. The TMSC depicts a scenario in which $S$ stores the new value of $x$, sends this value to $R$ since it has received a read request, and is then informed by $W$ that it has aborted its write. $S$ rolls back the value of $x$ by performing the local action $rb$, and then asks $R$ to abort. Both $W$ and $R$ have empty actions in the TMSC. This TMSC is part of a larger specification describing the interactions between $W$, $S$ and $R$ that we will consider later. For now, it may be noted that if the cost to abort $R$ is found to be prohibitively expensive, then it may be decided later in the design cycle to disallow $S$ from sending a dirty/uncommitted value to $R$. Hence the above scenario is *conditional*: $S$'s action is only conditionally required.

It may also be noted here that 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 [5].

We also propose another modification to the visual syntax of MSCs: the presence/absence of a small bar at the foot of the vertical line representing each instance. The presence of such a bar (as shown in Fig. 1) indicates that the instance cannot proceed beyond this point in the TMSC, while the absence means that the behavior of this instance beyond the TMSC is left unspecified. The latter feature allows users to describe the behavior of the instances in a step-by-step or cascaded fashion. For example, the trigger-action pair of an instance in a TMSC may be part of the trigger of the same instance in another TMSC that describes its subsequent behavior. It may also be noted that conventional MSCs may be represented as TMSCs with empty triggers.

Apart from the extensions proposed above, and the use of co-regions, TMSCs correspond to the basic MSCs in [11].

**Abstract syntax.** To define TMSCs mathematically we use some auxiliary notions. Let $\mathcal{I}$, $\mathcal{M}$, and $\mathcal{L}$ be finite sets of *instances*, *message types* and *local actions*, respectively. Then a *TMSC event alphabet* $\mathcal{E}[\mathcal{I},\mathcal{M},\mathcal{L}]$ is defined as follows.

$$\begin{aligned}\mathcal{E}[\mathcal{I},\mathcal{M},\mathcal{L}] \quad = \quad & \{\mathsf{out}(I_i,I_j,m) \mid I_i \neq I_j \in \mathcal{I}, m \in \mathcal{M}\}\\ \cup \quad & \{\mathsf{in}(I_i,I_j,m) \mid I_i \neq I_j \in \mathcal{I}, m \in \mathcal{M}\}\\ \cup \quad & \{\mathsf{loc}(I_i,\ell) \mid I_i \in \mathcal{I}, \ell \in \mathcal{L}\}\end{aligned}$$

Intuitively, event $\mathsf{out}(I_i,I_j,m)$ occurs when instance $I_i$ sends message $m$ to $I_j$, while $\mathsf{in}(I_i,I_j,m)$ occurs when a message $m$ sent by $I_i$ is received by $I_j$. Event $\mathsf{loc}(I_i,\ell)$ represents the execution by $I_i$ of local action $\ell$. We also define a function $\mathsf{active} : \mathcal{E}[\mathcal{I},\mathcal{M},\mathcal{L}] \to \mathcal{I}$ indicating which instance in an event descriptor "causes" the event to occur:

$$\mathsf{active}(e) = \begin{cases} I_i & \text{if } e = \mathsf{out}(I_i,I_j,m), \mathsf{loc}(I_i,\ell)\\ I_j & \text{if } e = \mathsf{in}(I_i,I_j,m)\end{cases}$$

In a send event, the sender is active, while in a receive event the receiver is active. Finally, the set of *co-regions* consists of the nonempty subsets of $\mathcal{E}[\mathcal{I},\mathcal{M},\mathcal{L}]$:

$$\mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}] \quad = \quad \{C \subseteq \mathcal{E}[\mathcal{I},\mathcal{M},\mathcal{L}] \mid C \neq \emptyset\}.$$

If $I_i \in \mathcal{I}$ then we write $\mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}]_{I_i}$ for the co-regions in which $I_i$ is active: $\mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}]_{I_i} = \{C \in \mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}] \mid \forall e \in C.\, \mathsf{active}(e) = I_i\}$. We now define TMSCs as follows.

DEFINITION 1. *A* triggered message sequence chart $M$ *is a tuple* $\langle \mathcal{I},\mathcal{M},\mathcal{L}, \mathit{trig}, \mathit{act}, \mathit{term}\rangle$*, where:*

1. $\mathcal{I}$*,* $\mathcal{M}$ *and* $\mathcal{L}$ *are finite sets of* instances*,* messages *and* local action names*, respectively;*

2. Trigger function $\mathit{trig} : \mathcal{I} \to \mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}]^*$ *satisfies:* $\mathit{trig}(I_i) \in \mathcal{C}[\mathcal{I},\mathcal{M},\mathcal{L}]^*_{I_i}$ *all* $I_i \in \mathcal{I}$*;*

3. Action function *act* : $\mathcal{I} \rightarrow \mathcal{C}[\mathcal{I}, \mathcal{M}, \mathcal{L}]^*$ *satisfies:* $act(I_i) \in \mathcal{C}[\mathcal{I}, \mathcal{M}, \mathcal{L}]^*_{I_i}$ *all* $I_i \in \mathcal{I}$;

4. *term* $\subseteq \mathcal{I}$ *is the* terminating set.

The trigger and action functions of a TMSC each map instances to sequences of co-regions. The intended semantics is that when an instance "performs" its trigger, it is then obliged to perform its action. The terminating set indicates which instances are to terminate after completing their actions; those not in this set have no constraints on their behavior after they complete their actions.

## 2.2 TMSC Expressions

Individual TMSCs can only capture single scenarios of systems. In order to provide capabilities for structured TMSC-based system specifications, we also introduce an algebra of operators that permit system specifications to be assembled out of subspecifications. We refer to the resulting terms as *TMSC expressions*. The following BNF-like grammar defines the syntax of TMSC expressions.

| | | | |
|---|---|---|---|
| $S$ | $::=$ | $M$ | (single TMSC) |
| | $\mid$ | $S \parallel S$ | (parallel composition) |
| | $\mid$ | $S \mp S$ | (delayed choice) |
| | $\mid$ | $S \oplus S$ | (internal choice) |
| | $\mid$ | $S; S$ | (sequential composition) |
| | $\mid$ | $S \wedge S$ | (logical and) |

The operators include a selection of "behavioral" and "logical" ones, with $\parallel$, $\mp$ and ; falling into the former category, $\wedge$ into the latter, and $\oplus$ falling into both. $S_1 \parallel S_2$ denotes the "parallel composition" of expressions $S_1$ and $S_2$, while $S_1 \mp S_2$ 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. In this respect $\oplus$ has overtones of logical disjunction. $S_1; S_2$ denotes the sequential composition of $S_1$ and $S_2$, where the composition is at the instance-level rather than at the expression-level and is thus similar to the notion of *asynchronous concatenation* in [4] and [13]. Finally, $S_1 \wedge S_2$ represents the logical conjunction of $S_1$ and $S_2$. The $\parallel$, $\mp$ and ; operators have been studied in the context of MSCs [13]; however the semantics of our $\parallel$ operator differs significantly from that of [13], as will be made clear later.

## 2.3 An Example: The W-S-R System

We now present a TMSC-expression specification of a simple but non-trivial system and use this as a running example. The system we consider consists of three instances: a writer $W$, a reader $R$, and a server $S$. $W$ may update the value of a variable $x$ by sending $w(x)$ to $S$ and then perform a local action (*ok*/$fail$) to decide whether it should *commit* (by sending *com* to $S$) or *abort* (by sending *abtd* to $S$). If $W$ commits, then the new value of $x$ is *clean* and remains so until the next update, whereas the value of $x$ is rolled back to its previous value by $S$ (by performing the local-action *rb*) if $W$ aborts. $R$ may request the value of $x$ by sending $r(x)$ to $S$, and $S$ responds by sending $val(x)$ to $R$. Now, if $x$ is clean or becomes clean, then $S$ allows $R$ to commit by sending *com*, whereas if $S$ sends an uncommitted $x$ value to $R$ which has to be rolled back then $S$ asks $R$ to abort by sending *abort*. The aim is to maintain *atomicity* (each of $W$ and $R$ either runs to completion or has no effect at all), and *isolation* (the concurrent execution of $W$ and $R$ has the same effect as their serial execution in some order).

A TMSC expression for the system may be given as follows.

$$
\begin{aligned}
Spec \quad = \quad & ((M_1 \mp M_2) \parallel (M_3 \oplus M_4)) && (1)\\
& \wedge M_5 \wedge M_6 \wedge (M_7 \mp M_8) && (2)\\
& \wedge M_9 \wedge M_{10} \wedge (M_{11} \mp M_{12}) && (3)
\end{aligned}
$$

where $M_1, \ldots M_{12}$ are as shown in Fig. 2. $Spec$ is structured in two pieces: a "basic interactions" component (1) and a "constraints" component (2 and 3). Each of $M_1$, $M_2$, $M_3$ and $M_4$ represents a possible isolated interaction between $S$ and either $W$ or $R$; note that each trigger is empty, and thus these TMSCs correspond to MSCs. $M_1 \mp M_2$ then captures the entire possible behavior occurring between $W$ and $S$: $\mp$ is used because the choice between committing or aborting cannot be made until after the $w(x)$ action common to both $M_1$ and $M_2$ occurs. Similarly, $M_3 \oplus M_4$ captures the behavior between $S$ and $R$; here $\oplus$ is used because the choice between $M_3$ and $M_4$ depends on factors (namely, $W$) that are outside the purview of the interaction between $S$ and $R$. The over-all unconstrained behavior of the system is then given by $(M_1 \mp M_2) \parallel (M_3 \oplus M_4)$.

The role of the "constraints" component of $M$ is to eliminate undesirable behaviors from the "basic interaction" component by introducing conjuncts involving conditional scenarios $M_5$–$M_{12}$. $M_5$–$M_8$ depict scenarios in which $W$ and $R$ sequentially interact with $S$ in some order, while $M_9$–$M_{12}$ represent cases in which $W$'s and $R$'s interactions with $S$ are interleaved. $M_5$ may be read as follows.

> If $W$ sends $w(x)$, performs *fail*, and sends *abtd*, then it should terminate. If $S$ receives $w(x)$ followed by *abtd*, then it should perform *rb*, receive $r(x)$, send $val(x)$, send *com*, and terminate. If $R$ sends $r(x)$, receives $val(x)$ and receives *com*, it should terminate.

Note how the trigger/action requirements are "localized" to each instance.

# 3. BACKGROUND

The previous section gave an informal interpretation of TMSC expressions. The remainder of this paper gives a mathematically precise semantics for TMSCs together with a refinement relation that determines when one TMSC expression is a faithful "elaboration" of another. This section presents notations used in what follows.

## 3.1 Sequences

If $A$ is a set then $A^*$ is the set of finite sequences over $A$. We use the following, where $a \in A$ and $w, w_1, w_2 \in A^*$.

| | |
|---|---|
| $-\ \epsilon, a$ | empty, single-element sequence |
| $-\ fst(w), lst(w)$ | first, last elements in $w \neq \epsilon$ |
| $-\ w_1 \cdot w_2$ | sequence concatenation |
| $-\ \lvert w \rvert_a$ | number of occurrences of $a$ in $w$ |
| $-\ a \in w$ | holds if $a$ is an element in $w$ |
| $-\ w_1 \preceq w_2$ | holds if $w_1$ is a prefix of $w_2$ |
| $-\ w_1 \prec w_2$ | holds if $w_1$ is a *proper* prefix of $w_2$ |
| $-\ subseq(w)$ | the set of (not necessarily contiguous) subsequences of $w$ |

Finally, for $w \in A^*$ and $S \subseteq A$, $w - S$ represents the sequence obtained by removing all occurrences of each element in $S$ from $w$.

## 3.2 The Must Preorder

The must preorder arises in the theory of process testing given in [8], where tests, which may also be thought of as processes that

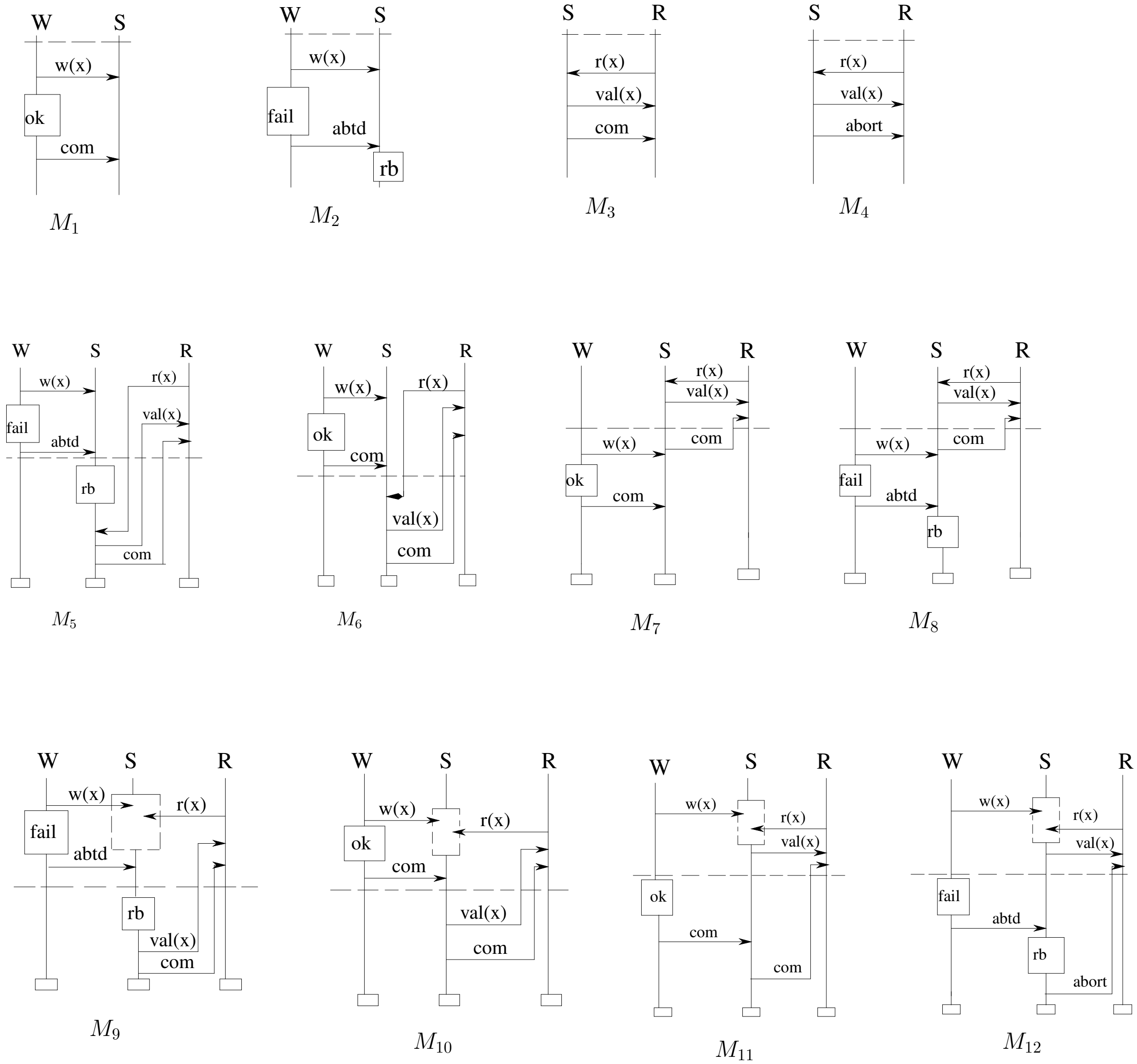


**Figure 2: TMSCs representing the interactions of $W$, $S$ and $R$.**

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. This subsection presents some of the theory of this relation in a simplified setting in which there are no (1) unobservable actions and (2) divergent processes.

The must preorder may be characterized in terms of *acceptance sets*. To define these we first introduce below some of the basic concepts from [8].

DEFINITION 2. *A* labeled transition system (LTS) *is a tuple $\langle P, E, \longrightarrow, p_I \rangle$, 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.*

An LTS describes the execution behavior of a system, with $P$ representing the states a system may be in, $E$ the set of events that may spark state transitions, $\longrightarrow$ the actual transitions that are possible, and $p_I$ the initial system state. We write $p \xrightarrow{e} p'$ in lieu of $\langle p, e, p' \rangle \in \longrightarrow$ and extend $\longrightarrow$ to sequences of events in the usual manner: if $w = a_1 \cdots a_n$ then $p \xrightarrow{w} p'$ if $p \xrightarrow{a_1} \cdots \xrightarrow{a_n} p'$.

The must preorder, $\sqsubseteq_{\text{must}}$, of [8] relates different LTSs in terms of their responses to tests. The acceptance-set characterization of this relation relies on the following notions.

DEFINITION 3. *Let $\mathcal{P} = \langle P, E, \longrightarrow, p_I \rangle$ be an LTS, with $p \in P$ and $w \in E^*$. The following may then be defined.*

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

The language of a system contains its "execution sequences", while the successors of a state are the events enabled in the state. The acceptance set of a system after a sequence is a measure of nondeterminism: for each state reachable via $w$ from the start state of $\mathcal{P}$, $Acc(\mathcal{P}, w)$ contains the events that are enabled in that state. Note that if $w \notin L(\mathcal{P})$ then $Acc(\mathcal{P}, w) = \emptyset$.

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 $\sqsubseteq_{\text{must}}$ is 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 4. *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 $s \in \mathcal{E}^*$, $sat(T(s)) = T(s)$.*
2. *For any $s, s' \in \mathcal{E}^*$, if $T(s) = \emptyset$ then $T(s \cdot s') = \emptyset$.*
3. *For any $s \in \mathcal{E}^*$, $e \in \mathcal{E}$, $T(s \cdot e) \neq \emptyset$ iff there exists $A \in T(s)$ such that $e \in A$.*

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

For any LTS $\mathcal{P}$ there is an immediate way to construct an acceptance tree $T[\mathcal{P}]$: $T[\mathcal{P}](s) = sat(Acc(\mathcal{P}, s))$. It immediately follows that $\mathcal{P}_1 \sqsubseteq_{\text{must}} \mathcal{P}_2$ if and only if $T[\mathcal{P}_1] \supseteq T[\mathcal{P}_2]$.

Acceptance trees are often used in support of proofs of *substitutivity*. In particular, to show that composition operators respect the must preorder, one may show that the operators are monotonic functions on acceptance trees. More concretely, suppose that $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{Q}$ are LTSs. To show that $op$ is substitutive with respect to $\sqsubseteq_{\text{must}}$, it must be shown that if $\mathcal{P}_1 \sqsubseteq_{\text{must}} \mathcal{P}_2$, then $op(\mathcal{P}_1, \mathcal{Q}) \sqsubseteq_{\text{must}} op(\mathcal{P}_2, \mathcal{Q})$. One way to do this is to define a binary operator $f_{op}$ on acceptance trees that is monotonic with respect to $\supseteq$ and has the property that $T[op(\mathcal{P}, \mathcal{Q})] = f_{op}(T[\mathcal{P}], T[\mathcal{Q}])$.

# 4. SEMANTICS OF TMSCS

We will now develop a formalism for TMSC expressions that is based on acceptance trees and the must preorder. We first show how individual TMSCs may be interpreted using variants of acceptance trees in which triggers are handled via nondeterminism: a TMSC is essentially treated as a nondeterministic choice of all behaviors violating the trigger together with those in which the trigger is satisfied and "progress" is made on performing the action. We then use this treatment of TMSCs as a basis for interpreting the TMSC expression operators introduced in Section 2.2. The net effect is a scheme for translating TMSC expressions into our specialized acceptance trees. As a byproduct of this approach, we immediately obtain a refinement ordering based on the must preorder. In this framework, TMSC expression $S_2$ is more refined than TMSC expression $S_1$ if $S_2$ in effect includes a subset of the nondeterministic behaviors of $S_1$.

In the remainder of this section we fix finite sets $\mathbb{I}, \mathbb{M}$ and $\mathbb{L}$ of instances, message types and local actions and require that all TMSCs draw their instance sets, message sets and local action sets from these. We write $\mathbb{R} = \{\mathsf{in}(I_i, I_j, m) \mid I_i, I_j \in \mathbb{I}, m \in \mathbb{M}\}$ for the set of all receive events. Our semantics also uses events of form $\mathsf{end}(I_i)$, where $I_i \in \mathbb{I}$, 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. The $\mathsf{end}(I_i)$ events, as we will see, are necessary in the definition of the ; operator. We write $\mathbb{T} = \{\mathsf{end}(I_i) \mid I_i \in \mathbb{I}\}$; $\mathbb{E} = \mathcal{E}[\mathbb{I}, \mathbb{M}, \mathbb{L}] \cup \mathbb{T}$ for the set of all events; and $\mathbb{W} = \{\mathsf{wait}(r) \mid r \in \mathbb{R}\}$ for the set of all "potential receive events". We extend the domain of $\mathsf{active}$ to $\mathbb{E} \cup \mathbb{W}$ by defining $\mathsf{active}(\mathsf{end}(I_i)) = I_i$, and $\mathsf{active}(\mathsf{wait}(r)) = \mathsf{active}(r)$. For any $E \subseteq \mathbb{E}$, $I \subseteq \mathbb{I}$, we write $E_I$ for $\{e \in E \mid \mathsf{active}(e) \in I\}$, the subset of $E$ in which some instance in $I$ is active. We use $\mathbb{C} = \mathcal{C}[\mathbb{I}, \mathbb{M}, \mathbb{L}]$ for the set of all co-regions.

In what follows we fix TMSC $M = \langle \mathcal{I}, \mathcal{M}, \mathcal{L}, \mathit{trig}, \mathit{act}, \mathit{term}\rangle$ (note that $\mathcal{I} \subseteq \mathbb{I}, \mathcal{M} \subseteq \mathbb{M}$ and $\mathcal{L} \subseteq \mathbb{L}$).

**Interpreting instances.** We begin our development by showing how individual instances within $M$ may be equipped with an acceptance-tree semantics. The first definitions associate languages with co-regions and co-region-sequences.

DEFINITION 5. *Let $C \in \mathbb{C}$.*

1. *$L(C) \subseteq \mathbb{E}^*$ is defined as:*

$$L(C) = \begin{cases} \{e\} & \text{if } C = \{e\} \\ \bigcup_{e \in C}\{e \cdot w \mid w \in L(C - \{e\})\} & \\ \quad \text{otherwise} & \end{cases}$$

2. *Let $W, W' \in \mathbb{C}^*$. Then $L(W) \subseteq \mathbb{E}^*$ is defined as:*

$$L(W) = \begin{cases} \{\epsilon\} & \text{if } W = \epsilon \\ \bigcup_{w \in L(C)}\{w \cdot w' \mid w' \in L(W')\} & \\ \quad \text{if } W = C \cdot W' & \end{cases}$$

$L(C)$ contains all sequences in which every element in $C$ appears once. If $W = C_1 \cdot \cdots \cdot C_n$ then $L(W)$ consists of words of the form $w_1 \cdot \cdots \cdot w_n$ where each $w_i \in L(C_i)$.

The next definition introduces the notions of *trigger* and *action* languages of instances in $M$.

DEFINITION 6. *Let $I_i \in \mathcal{I}$. Then $\mathit{trigL}_M(I_i)$, the* trigger language*, of $I_i$ in $M$ is given by $\mathit{trigL}_M(I_i) = L(\mathit{trig}(I_i))$. The* action language*, $\mathit{actL}_M(I_i)$, of $I_i$ in $M$ is defined as $\mathit{actL}_M(I_i) = L(\mathit{act}(I_i))$.*

The *language*, $L_M(I_i)$, of an instance $I_i$ records the possible sequences of events the instance might generate as it executes. Intuitively, if a sequence does not "satisfy" the trigger of $I_i$, then it will be admitted as a sequence. Otherwise, it will be constrained to "satisfy" the action.

DEFINITION 7. *Let $I_i \in \mathcal{I}$. Then $L_M(I_i) \subseteq \mathbb{E}^*_{\{I_i\}}$ is defined as follows.*

$$\begin{aligned} L_M(I_i) \;=\; \{ & w \in \mathbb{E}^*_{\{I_i\}} \mid (|w|_{\mathit{end}(I_i)} \leq 1) \\ & \wedge(|w|_{\mathit{end}(I_i)} = 1 \Rightarrow \mathit{lst}(w) = \mathit{end}(I_i)) \\ & \wedge(\forall w_1 \in \mathit{trigL}_M(I_i).\, w_1 \preceq w \Rightarrow (\exists w_2 \in \mathit{actL}_M(I_i). \\ & \quad (w \preceq w_1 \cdot w_2) \vee (I_i \in \mathit{term} \wedge w = w_1 \cdot w_2 \cdot \mathit{end}(I_i)) \\ & \quad \vee(I_i \notin \mathit{term} \wedge w_1 \cdot w_2 \preceq w)))\} \end{aligned}$$

*The* maximal language *of $I_i$ is given by:*
$\mathit{maxL}_M(I_i) = \{w \in L_M(I_i) \mid \mathit{lst}(w) = \mathit{end}(I_i)\}$.

To understand this definition, it should first be noted that $L_M(I_i)$ contains sequences of events in which $I_i$ is active (that is, sequences in $\mathbb{E}^*_{\{I_i\}}$). These sequences must also be *well-terminated*: they contain at most one occurrence of $\mathsf{end}(I_i)$, and this event, if present, must be the last event in the sequence. Finally, if the sequence includes an element of $I_i$'s trigger language as a prefix, then the sequence must also include at least part of a sequence in the action language. If the sequence includes a complete action sequence and $I_i$ is terminating, the sequence can also include an $\mathsf{end}(I_i)$ action; if $I_i$ is nonterminating, any sequence after the completed action sequence is allowed.

To define acceptance sets for instances we need the following operation on languages. Let $L \subseteq A^*$ and $w \in A^*$. Then the *next set*, $next(L, w) \subseteq A$, of $L$ after w is given by: $next(L, w) = \{a \in A \mid \exists w' \in L.\ w \cdot a \preceq w'\}$. We now define how to associate an acceptance set with an instance $I_i$ in a TMSC $M$. The acceptance set construction differs from the traditional one for LTSs given previously in that it is given relative to a set of "enabled inputs". An instance can only emit an input event if another instance has emitted the corresponding output; otherwise, this input event is not enabled. To capture this behavior, we introduce an additional parameter, $eR \subseteq \mathbb{R}$, into the acceptance-set function. An input event in $eR$ is deemed enabled; otherwise, it is defined to be disabled. We also define the *nondeterminism set* of $E \subseteq \mathbb{E}$ and enabled inputs $eR \subseteq \mathbb{R}$ as follows.

$$\begin{aligned} ND(E, eR) \quad = \quad & sat(\{\{e\} \mid e \in E \wedge (e \in \mathbb{R} \Rightarrow e \in eR)\} \\ & \cup \{\{\mathsf{wait}(r)\} \mid r \in ((E \cap \mathbb{R}) - eR)\}) \end{aligned}$$

$ND(E, eR)$ represents an acceptance set of a system that can nondeterministically decide to perform any event in $E$ that is *enabled*, where any output or local event, and any input in $eR$, is enabled, or $\mathsf{wait}$ for any input event in $E$ that is not yet enabled.

DEFINITION 8. *Let $I_i \in \mathcal{I}$, $w \in \mathbb{E}^*$ and $eR \subseteq \mathbb{R}_{\{I_i\}}$. Then $Acc(I_i, M, w, eR)$ is defined as follows.*

$$Acc(I_i, M, w, eR) = \begin{cases} \emptyset \\ \quad \text{if } w \notin L_M(I_i) \\ \{\emptyset\} \\ \quad \text{if } w \in maxL_M(I_i) \\ \{\{\mathsf{end}(I_i)\}\} \\ \quad \text{if } I_i \in term \text{ and } \exists w_1 \in trigL_M(I_i), w_2 \in \\ \quad actL_M(I_i).\ w = w_1 \cdot w_2 \\ ND(next(L_M(I_i), w), eR) \\ \quad \text{if } \exists w_1 \in trigL_M(I_i), w_2 \in actL_M(I_i).\ w_1 \preceq w \prec \\ \quad w_1 \cdot w_2 \\ ND(\mathbb{E}_{\{I_i\}}, eR) \\ \quad \text{otherwise} \end{cases}$$

$Acc(I_i, M, w, eR)$ is the acceptance set of instance $I_i$ in TMSC $M$ after $w$. The first three clauses above handle the case when $I_i$ is incapable of performing $w$ or when $I_i$ has terminated, or is about to terminate, after $w$. The fourth clause occurs when $w$ consists of a complete word in $I_i$'s trigger language and an incomplete word in it's action language. In this case, the acceptance set includes only events that are possible "next", after $w$, together with any potential receive events that are not enabled. The final case holds either if $w$ does not contain an element in $I_i$'s trigger as a prefix, or if $w$ has a prefix that is an element in $I_i$'s trigger followed by an element in its action and $I_i$ does not terminate.

The above definition associates acceptance sets with the set of instances $\mathcal{I}$ participating in the scenario depicted by the TMSC $M$. This definition, however, does not say anything about the behavior of instances in $\mathbb{I} - \mathcal{I}$. We assume that any such instance, whose behavior is not explicitly described in $M$, has empty trigger and action languages in $M$ and must terminate. This assumption simplifies the semantics of the ; operator while still ensuring compositionality, as will be noted later.

DEFINITION 9. *Let $I_i \in \mathbb{I} - \mathcal{I}$, $w \in \mathbb{E}^*$ and $eR \subseteq \mathbb{R}_{\{I_i\}}$. Then $Acc(I_i, M, w, eR)$ is defined as follows.*

$$Acc(I_i, M, w, eR) \quad = \quad \begin{cases} \{\{\mathsf{end}(I_i)\}\} & \text{if } w = \epsilon \\ \{\emptyset\} & \text{if } w = \mathsf{end}(I_i) \\ \emptyset & \text{otherwise} \end{cases}$$

Thus, any instance $I_i \in \mathbb{I} - \mathcal{I}$ can only perform the event $\mathsf{end}(I_i)$ in $M$, after which it terminates.

**Interpreting TMSCs.** To define the semantics of a TMSC we define a "parallel composition" operator on acceptance trees and apply it to the acceptance trees of individual instances implied by the previous two definitions. If $E \subseteq \mathbb{E}$, then

$$\mathsf{activeset}(E) \quad = \quad \{I \mid \exists e \in E.\mathsf{active}(e) = I\}.$$

Thus $\mathsf{activeset}(E)$ contains the set of instances which are "active" in $E$.

Let $A, B \subseteq \mathbb{E}$. We define the *parallel composition*, $A * B$, of $A$ and $B$, as $A * B = sat(\{E \subseteq A \cup B \mid C_1(E) \wedge C_2(E) \wedge C_3(E)\})$; the $C_i(E)$ conditions are given below. Note that $I_A = \mathsf{activeset}(A)$, $I_B = \mathsf{activeset}(B)$, and we use $I_A \sim I_B$ for the *symmetric difference* of $I_A$ and $I_B$: $I_A \sim I_B = (I_A - I_B) \cup (I_B - I_A)$.

$$\begin{aligned} C_1(E) \quad : \quad & (A \cup B)_{I_A \sim I_B} \subseteq E \\ C_2(E) \quad : \quad & \forall I_i \in I_A \cap I_B.\ \mathsf{end}(I_i) \in A \cap B \Rightarrow \\ & A_{\{I_i\}} \subseteq E \vee B_{\{I_i\}} \subseteq E \\ C_3(E) \quad : \quad & \forall I_i \in I_A \cap I_B.\ \mathsf{end}(I_i) \notin A \cap B \Rightarrow \\ & \begin{cases} E_{\{I_i\}} = A_{\{I_i\}} & \text{if } B_{\{I_i\}} = \{\mathsf{end}(I_i)\} \\ E_{\{I_i\}} = B_{\{I_i\}} & \text{if } A_{\{I_i\}} = \{\mathsf{end}(I_i)\} \\ (A_{\{I_i\}} - \{\mathsf{end}(I_i)\}) \subseteq E & \\ \quad \vee (B_{\{I_i\}} - \{\mathsf{end}(I_i)\}) \subseteq E & \text{otherwise} \end{cases} \end{aligned}$$

The basic idea behind the definition of $*$ is this: $I_A$ ($I_B$) is a set of instances, and $A$ ($B$) represents a possible "state" the instances may be in, in terms of the events they can perform next. $*$ defines the set of "combined" states that $I_A \cup I_B$ can be in, where the instance sets $I_A$ and $I_B$ are running in parallel. Any instance which is in one of $I_A$ or $I_B$, but not in the other, is in only one possible state: its state in $A$, if it is in $I_A$, or its state in $B$, otherwise. This is reflected in $C_1(E)$. However, if an instance is in both $I_A$ and $I_B$, then it can nondeterministically choose to be either in its state in $A$ or its state in $B$. Moreover, it can only terminate if both its states allow the terminating event. These notions are captured by $C_2(E)$ and $C_3(E)$.

We now use $*$ in defining a notion of parallel composition for acceptance sets.We first introduce an operator $\sqcup$ on acceptance sets: $\mathcal{A} \sqcup \mathcal{B} = sat(\mathcal{A} \cup \mathcal{B})$. We define an indexed version $\bigsqcup_{k \in K}$ in the obvious manner, with $\bigsqcup_{k \in K} \mathcal{A}_K = \emptyset$ if $K = \emptyset$. Let $\mathcal{A}$ and $\mathcal{B}$ be acceptance sets. Then

$$\mathcal{A} \bowtie \mathcal{B} \quad = \quad \bigsqcup_{A \in \mathcal{A}, B \in \mathcal{B}} (A * B)$$

Intuitively, $\mathcal{A}$ and $\mathcal{B}$ represent the set of "states" of two different systems, in terms of their acceptance sets. $\mathcal{A} \bowtie \mathcal{B}$ computes the set of states that the parallel composition of the two systems may be in, by applying the $*$ operator defined above in a "point-wise" manner. It can be shown that $\bowtie$ is commutative and associative. We use $\bowtie_{k \in K}$ as the "indexed" version of $\bowtie$, with $\bowtie_{k \in K} \mathcal{A}_K = \emptyset$ if $K = \emptyset$ .

We now show how to associate an acceptance set with TMSC $M$ after a sequence $w$. The definition will use the $\bowtie$ operator just defined. First, however, we introduce the following definitions on event sequences.

DEFINITION 10. *Let $w \in \mathbb{E}^*$.*

1. *The* projection*, $w\lfloor I_i$, of $w$ onto $I_i \in \mathbb{I}$ is the longest (not necessarily contiguous) subsequence of $w$ containing only events in which $I_i$ is active:*

$$w\lfloor I_i \quad = \quad \begin{cases} \epsilon & \text{if } w = \epsilon \\ e \cdot (w'\lfloor I_i) & \text{if } w = e \cdot w', \textsf{active}(e) = I_i \\ w'\lfloor I_i & \text{if } w = e \cdot w', \textsf{active}(e) \neq I_i \end{cases}$$

2. *The receive event $\textsf{in}(I_i, I_j, m)$ is called* enabled *by $w$ if $|w|_{\textsf{out}(I_i,I_j,m)} > |w|_{\textsf{in}(I_i,I_j,m)}$. We use $e\mathcal{R}(w)$ to stand for all receive events enabled by $w$.*

3. *$w$ is called* well-balanced *iff for every $w' \preceq w$ such that $w' = w'' \cdot \textsf{in}(I_i, I_j, m)$, $\textsf{in}(I_i, I_j, m) \in e\mathcal{R}(w'')$.*

In a well-balanced sequence of events, every receive is preceded by an "outstanding" send. We also call $w$ *feasible* for $M$ if $w$ is well-balanced and $Acc(I_i, M, w\lfloor I_i, eR_{\{I_i\}}) \neq \emptyset$ for any set of enabled receives $eR$, $I_i \in \mathbb{I}$. We now define the acceptance set $Acc(M, w)$ as follows.

DEFINITION 11. *The acceptance set $Acc(M, w)$, of $M$ after $w \in \mathbb{E}^*$ is defined as:*

$$Acc(M, w) \quad = \quad \begin{cases} \emptyset \\ \quad \text{if } w \text{ is not feasible for } M \\ \bowtie_{I_i \in \mathbb{I}} Acc(I_i, M, w\lfloor I_i, e\mathcal{R}(w)_{\{I_i\}}) \\ \quad \text{otherwise} \end{cases}$$

It may be noted that as the acceptance set of a TMSC is given in terms of the parallel composition of the acceptance sets of individual (and distinct) instances, only condition $C_1(E)$ in the definition of the $*$ operator needs to be used. The remaining conditions arise when we consider the parallel composition of two TMSC expressions having some instances in common.

*Example:* In $M_1$ of Fig.2, let us consider:

$$\begin{aligned} \mathbb{I} &= \{W, S, R\} \\ w &= \textsf{out}(W, S, w(x)) \cdot \textsf{loc}(W, ok). \end{aligned}$$

Then:

$$\begin{aligned} Acc(M_1, w) &= \{\{\textsf{out}(W, S, com)\}\} \bowtie \{\{\textsf{in}(W, S, w(x))\}\} \\ &\quad \bowtie \{\{\textsf{end}(R)\}\} \\ &= \{\{\textsf{out}(W, S, com), \textsf{in}(W, S, w(x), \textsf{end}(R)\}\}. \end{aligned}$$

## The Must Preorder for TMSCs

With our definition of $Acc$ we may now define a version of the must preorder on TMSCs. The must preorder is based on an ordering on a variant of the acceptance trees as presented in Section 3. The differences between this variant (henceforth referred to simply as "acceptance trees" in our framework), and the traditional acceptance trees will be clear from the following definition.

DEFINITION 12. *An* acceptance tree *is a function $T : \mathbb{E}^* \rightarrow 2^{2^{(\mathbb{E} \cup \mathbb{W})}}$ satisfying the following.*

1. *For any $s \in \mathbb{E}^*$, $sat(T(s)) = T(s)$.*

2. *For any $s, s' \in \mathbb{E}^*$, if $T(s) = \emptyset$ then $T(s \cdot s') = \emptyset$.*

3. *For any $s \in \mathbb{E}^*$, $e \in \mathbb{E}$, $T(s \cdot e) \neq \emptyset \Rightarrow \exists A \in T(s)$ such that $e \in A$.*

Thus, an element in an acceptance set may contain not only events as in Definition 4, but also "potential events" of form $\textsf{wait}(r)$, $r \in \mathbb{R}$. Also, Clause 3 in the above definition relaxes the corresponding clause in Definition 4; this is necessitated by the fact that TMSC expressions may contain the $\wedge$ operator, which is absent in traditional testing theory.

It is easy to associate an acceptance tree $T[M]$ to a TMSC $M$: $T[M](w) = Acc(M, w)$. We say $T1 \supseteq T2$ when for all $w \in \mathbb{E}^*$, $T1(w)) \supseteq T2(w)$. We now define the must preorder on TMSCs.

DEFINITION 13. *Let $M_1, M_2$ be TMSCs. Then $M_1 \sqsubseteq_{must} M_2$ iff $T[M_1] \supseteq T[M_2]$.*

# 5. SEMANTICS OF TMSC EXPRESSIONS

In this section we give interpretations of the TMSC composition operators introduced in Section 2 and establish several properties of our semantics.

## 5.1 Acceptance Tree Constructions

To define the semantics of TMSC composition operators we rely on the strategy outlined at the end of Section 3: we interpret $\mp$, $\oplus$, $\wedge$, $\|$ and ; in terms of acceptance trees. The construction of $T[S]$ proceeds inductively on the structure of $S$.

**Delayed choice**: The operator $\mp$ was originally designed as a deterministic choice construct for MSCs. Traditional MSCs are deterministic; $\mp$ was intended to preserve this property. To define it formally in our setting, let $S_1, S_2$ be TMSC expressions, and assume inductively that we have constructed $T[S_1]$ and $T[S_2]$. We set $T[S_1 \mp S_2](w)$ to $sat\{A_1 \cup A_2 \mid A_1 \in \mathcal{A}_1, A_2 \in \mathcal{A}_2\}$, where $T[S_1](w) = \mathcal{A}_1$ and $T[S_2](w) = \mathcal{A}_2$.

**Internal Choice**: Our $\oplus$ operator, in contrast to the delayed choice operator $\mp$, offers non-deterministic choice. Given two TMSC specifications $S_1$ and $S_2$, and a sequence of events $w$, let $T[S_1](w) = \mathcal{A}_1$ and $T[S_2](w) = \mathcal{A}_2$. We define $T[S_1 \oplus S_2](w, eR)$ as $sat(\mathcal{A}_1 \cup \mathcal{A}_2)$.

**Conjunction**: The $\wedge$ operator is usually thought of as a logical rather than a behavioral construct. In TMSC expressions, however, one often wants to require that an implementation satisfy two TMSC expressions, each specifying a different requirement. The $\wedge$ operator allows such specifications to be made. If $S_1, S_2$ are TMSC expressions, we define $T[S_1 \wedge S_2](w) = sat(\mathcal{A}_1 \cap \mathcal{A}_2)$ where $T[S_1](w) = \mathcal{A}_1$ and $T[S_2](w) = \mathcal{A}_2$

**Parallel composition**: The $\|$ operator "runs" two TMSC expressions in parallel. Any sequence $w$ of events that is generated by the execution of the combined specification will consist of some events that are generated by one subexpression (say, a sequence $w_1$) and the others by the other subexpression (say, a sequence $w_2$). The acceptance set for the parallel expression after $w$ is then given in terms of the acceptance sets of the subexpressions after $w_1$ and $w_2$, for all such $w_1$ and $w_2$ that may together generate $w$.

To formalize these intuitions, for $w_1, w_2 \in \mathbb{E}^*$ and $I \subseteq \mathbb{I}$, we let *shuffle*$(w_1, w_2, I)$ denote the set of all possible ways in which $w_1$ and $w_2$ may be interleaved, with each instance $I_i \in I$ terminating in a resulting sequence iff it terminates in both $w_1$ and $w_2$:

$$\mathit{shuffle}(w_1, w_2, I) = \begin{cases} \{\epsilon\} \\ \quad \text{if } w_1, w_2 = \epsilon \\ \mathit{shuffle}(w_p, w_2, I - \{I_i\}) \\ \quad \text{if } w_1 = end(I_i).w_p, I_i \in I \\ \mathit{shuffle}(w_1, w_q, I - \{I_i\}) \\ \quad \text{if } w_2 = end(I_i).w_q, I_i \in I \\ \{a \cdot w | w \in \mathit{shuffle}(w_p, \epsilon, I)\} \\ \quad \text{if } w_1 = a \cdot w_p, w_2 = \epsilon, \nexists I_i \in I.a = end(I_i) \\ \{b \cdot w | w \in \mathit{shuffle}(\epsilon, w_q, I)\} \\ \quad \text{if } w_2 = b \cdot w_q, w_1 = \epsilon, \nexists I_i \in I.b = end(I_i) \\ \{a \cdot w' | w' \in \mathit{shuffle}(w_p, b \cdot w_q, I)\} \\ \quad \cup \{b \cdot w' | w' \in \mathit{shuffle}(a \cdot w_p, w_q, I)\} \\ \quad \text{if } w_1 = a \cdot w_p \text{ and } w_2 = b \cdot w_q, \nexists I_i, I_j \in I.a = \\ \quad end(I_i), b = end(I_j) \end{cases}$$

We now define $T[S_1 \parallel S_2](w)$.

$$T[S_1 \parallel S_2](w) = \begin{cases} \emptyset \\ \quad \text{if } \mathsf{L} = \emptyset \\ \bigsqcup_{\langle w_1, w_2 \rangle \in \mathsf{L}} (T[S_1](w_1) \bowtie T[S_2](w_2)) \quad \text{o.w.} \end{cases}$$

where,
$\mathsf{L} = \{\langle w_1, w_2 \rangle \mid T[S_1](w_1), T[S_2](w_2) \neq \emptyset, w \in \mathit{shuffle}(w_1, w_2, \mathbb{I})\}$.

The tuples $\langle w_1, w_2 \rangle$ in $\mathsf{L}$ represent the different ways in which $w$ may have been produced by the parallel execution of $S_1$ and $S_2$. For each such way, we use the $\bowtie$ operator to compute the corresponding acceptance set of $S_1 \parallel S_2$; the overall acceptance set is then given by the non-deterministic choice of all the possible acceptance sets.

*Example:* Let $M_1, M_2, M_3, M_4$ be as in Fig.2, and consider:

$$\begin{aligned} S &= S_1 \parallel S_2 \\ S_1 &= (M_1 \mp M_2) \\ S_2 &= (M_3 \oplus M_4) \\ \mathbb{I} &= \{W, S, R\} \\ w &= \mathsf{out}(W, S, w(x)) \cdot \mathsf{out}(R, S, r(x)) \cdot \mathsf{loc}(W, ok). \end{aligned}$$

One possible way in which $w$ may have been generated is:

$$\begin{aligned} w_1 &= \mathsf{out}(W, S, w(x)) \cdot \mathsf{loc}(W, ok) \\ w_2 &= \mathsf{out}(R, S, r(x)) \end{aligned}$$

Then we have the following.

$$\begin{aligned} T[S_1](w_1) &= \{\{\mathsf{out}(W, S, com), \mathsf{in}(W, S, w(x)), \mathsf{end}(R)\}\} \\ T[S_2](w_2) &= \{\{\mathsf{in}(R, S, r(x)), \mathsf{wait}(\mathsf{in}(S, R, val(x)), \mathsf{end}(W)\}\} \end{aligned}$$

Finally,

$$\begin{aligned} &T[S_1](w_1) \bowtie T[S_2](w_2) = \\ &\{\{\mathsf{out}(W, S, com), \mathsf{in}(W, S, w(x)), \mathsf{wait}(\mathsf{in}(S, R, val(x)))\}, \\ &\ \{\mathsf{out}(W, S, com), \mathsf{in}(R, S, r(x)), \mathsf{wait}(\mathsf{in}(S, R, val(x))\}, \\ &\ \{\mathsf{out}(W, S, com), \mathsf{in}(W, S, w(x)), \mathsf{in}(R, S, r(x)), \\ &\quad \mathsf{wait}(\mathsf{in}(S, R, val(x))\}\} \end{aligned}$$

There are other ways in which $w$ may have been generated, e.g.

$$\begin{aligned} w_1' &= \mathsf{out}(W, S, w(x)) \cdot \mathsf{end}(R) \cdot \mathsf{loc}(W, ok) \\ w_2' &= \mathsf{out}(R, S, r(x)) \cdot \mathsf{end}(W) \end{aligned}$$

However, since any instance may terminate only when it may do so in both $S_1$ and $S_2$, the acceptance sets corresponding to these other possibilities will be the same as $T[S_1](w_1) \bowtie T[S_2](w_2)$, so we do not show them here. Thus, $T[S](w) = T[S_1](w_1) \bowtie T[S_2](w_2)$.

**Sequential Composition:** The ; operator connects two TMSC expressions sequentially. Given a TMSC expression $S_1; S_2$, any instance $I \in \mathbb{I}$ may perform an event in $S_2$ only if it has terminated in $S_1$. Let $A, B \subseteq \mathbb{E}$. We define the sequential composition $A \sharp B$, of $A$ and $B$ as $A \sharp B = E$, where, $\forall I_i \in \mathbb{I}$,

$$E_{\{I_i\}} = \begin{cases} A_{\{I_i\}} \\ \quad \text{if } I_i \in \mathsf{activeset}(A) \wedge \mathsf{end}(I_i) \notin A_{\{I_i\}} \\ (A_{\{I_i\}} - \{\mathsf{end}(I_i)\}) \cup B_{\{I_i\}} \\ \quad \text{otherwise} \end{cases}$$

The $\sharp$ operator may be explained as follows: we may think of $A$ as one possible state a TMSC expression $S_1$ is in, while $B$ represents a state of an expression $S_2$, where $S_2$ follows $S_1$ sequentially. $A \sharp B$ then defines the state $E$ which represents one possible state of $S_1; S_2$. Any instance $I_i$, which is active in $A$ but not yet ready to terminate in $A$, is in one possible state: its state in $A$. If $I_i$ has already terminated in $S_1$, then its current state is its state in $B$. Finally, if $I_i$ is in a position to terminate in $A$, then it may choose to continue in $S_1$ (if possible) without terminating, or to proceed as in $B$. Note that an $\mathsf{end}(I_i)$ event in $A$ will not be emitted as $I_i$ can terminate in $S_1; S_2$ only when it terminates in $S_2$.

We now lift the notion of sequential composition to acceptance sets. If $\mathcal{A}$ and $\mathcal{B}$ are acceptance sets, we define $\mathcal{A} \colon \mathcal{B}$, the sequential composition of $\mathcal{A}$ and $\mathcal{B}$ as

$$\mathcal{A} \colon \mathcal{B} \quad = \bigsqcup_{A \in \mathcal{A}, B \in \mathcal{B}} (A \sharp B)$$

Finally, let $\mathsf{activeseq}(w)$ be the set of instances that have performed at least one event in $w \in \mathbb{E}^*$:

$$\mathsf{activeseq}(w) \quad = \quad \{I \mid \exists e \in w.\mathsf{active}(e) = I\}$$

We now define the sequential composition of TMSC expressions. Given two TMSC expressions $S_1$ and $S_2$, we define $T[S_1; S_2](w)$ as follows:

$$T[S_1; S_2](w) = \begin{cases} \emptyset \\ \quad \text{if } \mathsf{L}' = \emptyset \\ \bigsqcup_{\langle w_1, w_2 \rangle \in \mathsf{L}'} (T[S_1](w_1) \colon T[S_2](w_2)) \quad \text{o.w.} \end{cases}$$

where $\mathsf{L}' = \{\langle w_1, w_2 \rangle \mid C_1' \wedge C_2' \wedge C_3'\}$. The $C_i'$ conditions are as follows:

$$\begin{aligned} C_1' &: \quad T[S_1](w1) \neq \emptyset, T[S_2](w_2) \neq \emptyset \\ C_2' &: \quad \forall I_i \in \mathbb{I}, I_i \in \mathsf{activeseq}(w_2) \Rightarrow \mathit{lst}(w_1 \lfloor I_i) = \mathsf{end}(I_i) \\ &\quad\ \wedge (w_1 \lfloor I_i - \{\mathsf{end}(I_i)\}) \cdot w_2 \lfloor I_i = w \lfloor I_i \\ C_3' &: \quad w_1 - \{\mathsf{end}(I_i) \mid I_i \in \mathbb{I}\} \in \mathit{subseq}(w) \\ &\quad\ \wedge w_2 \in \mathit{subseq}(w) \end{aligned}$$

$\mathsf{L}'$ contains tuples of the form $\langle w_1, w_2 \rangle$, and represents all possible ways in which $w$ may have been produced by the sequential execution of $S_1$ and $S_2$. Thus $w_1$ and $w_2$ should be execution sequences of $S_1$ and $S_2$, respectively ($C_1'$); also, the execution of each instance in $S_2$ follows its execution (and termination) in $S_1$. This is ensured by $C_2'$. Finally, $w_1$, with the terminating events removed, (as instances in $S_1; S_2$ explicitly terminate only in $S_2$) and $w_2$ should also be subsequences of $w$, so that events of different instances have the same relative order in $w_1$ and $w_2$, as in $w$. This explains the necessity of $C_3'$.

## 5.2 The Must Preorder and Properties of the Semantics

We may lift $\sqsubseteq_{\text{must}}$ to TMSC expressions in the obvious way.

DEFINITION 14. *Let $S_1, S_2$ be TMSC expressions. Then $S_1 \sqsubseteq_{must}$ $S_2$ if and only if $T[S_1] \supseteq T[S_2]$.*

Our semantics for TMSC expressions may be shown to have the following properties, which we state without proof.

THEOREM 2. *The operators $\mp, \oplus, \wedge$ and $\|$ are commutative and associative, the operator ; is associative, and $\mp$, $\oplus$ and $\wedge$ are idempotent.*

The next theorem shows that our semantics for $\wedge$ coincides with logical conjunction.

THEOREM 3. *Let $S_1, S_2$ and $S_3$ be TMSC expressions. Then $S_1 \wedge S_2 \sqsubseteq_{must} S_3$ if and only if $S_1 \sqsubseteq_{must} S_3$ and $S_2 \sqsubseteq_{must} S_3$.*

We may additionally show that our semantics for $\mp$ conservatively extends the one given in [13], and our semantics for $\|$ conservatively extends the one given in [13] when instance-sets are disjoint. For lack of space the formal statement is not given here.

The last result establishes that $\sqsubseteq_{\text{must}}$ is substitutive.

THEOREM 4. *Let $S_1, S_2$ and $S_3$ be TMSC expressions such that $S_1 \sqsubseteq_{must} S_2$. Then the following hold:*

1. $S_1 \mp S_3 \sqsubseteq_{must} S_2 \mp S_3$;
2. $S_1 \oplus S_3 \sqsubseteq_{must} S_2 \oplus S_3$;
3. $S_1 \wedge S_3 \sqsubseteq_{must} S_2 \wedge S_3$;
4. $S_1 \| S_3 \sqsubseteq_{must} S_2 \| S_3$
5. $S_1; S_3 \sqsubseteq_{must} S_2; S_3$ *and* $S_3; S_1 \sqsubseteq_{must} S_3; S_2$.

This theorem establishes that our semantics is compositional in the following sense: a TMSC expression may be refined by refining one of its subexpressions in isolation.

# 6. THE W-S-R SYSTEM REVISITED

From the discussion of the TMSC semantics, it is clear that the specification *Spec* of the W-S-R system, described in Section 2.3, offers several alternatives for eventual implementation. Let us consider two refinements of this specification, $Ref_1$ and $Ref_2$, which we also present as TMSC expressions.

$$\begin{aligned} Ref_1 &= M_{13} \mp M_{14} \mp M_{15} \mp M_{16} \\ Ref_2 &= M_{13} \mp M_{14} \mp M_{15} \mp M_{16} \mp M_{17} \mp M_{18} \end{aligned}$$

$M_{13}, M_{14}, \ldots, M_{18}$ are as shown in Fig.3. One may show that $Spec \sqsubseteq_{\text{must}} Ref_1$ and $Spec \sqsubseteq_{\text{must}} Ref_2$ both hold.

In $Ref_1$, $S$ interacts with $R$ and $W$ in only some sequential order. It may be noted that the conditional scenarios depicted in $M_{11}$ and $M_{12}$ do not appear in this refinement. In $Ref_2$, the read and the write operations are allowed to interleave. If $S$ receives $w(x)$ and $r(x)$ in any order, then the new value of $x$ is sent to $R$; if $W$ then commits, then $R$ is also allowed to commit, but if $W$ aborts, then the value of $x$ is rolled back, and $R$ is also asked to abort.

The specification of the system that we have presented consists of a single write and a single read. We are currently working on an operator for expressing iteration within our framework, so that we may capture repetitive behavior of systems. These, and other possible extensions to the TMSC specification language will be explored in future.

# 7. CONCLUSIONS

We have proposed a framework for describing specifications containing conditional scenarios using Triggered Message Sequence Charts, which extend traditional Message Sequence Charts with the concepts of *triggers* and *actions*. The visual syntax of TMSCs closely follows that of MSCs; the semantics of TMSC expressions allows us to translate a specification to a general framework based on acceptance trees, which may be used to relate TMSC specifications with specifications expressed in other formalisms with a similar semantics. We also showed how our semantics supports different composition operators on TMSCs. We have presented a notion of refinement that is based on the must preorder; a refinement is thus obtained by making the specification more deterministic. Finally, we have presented the TMSC specification of a simple but non-trivial system, and presented two of the possible refinements of the specification.

We are currently developing a process algebra for TMSC expressions so that we may automate the translation of a TMSC specification to an acceptance tree, using the Concurrency Workbench [6], and check the validity of proposed refinements of a specification. We are also incorporating an iterative operator into the TMSC framework. For future work, we would like to consider other useful extensions to the TMSC language, and investigate ways to generate deterministic, state-machine based descriptions of individual instances from a given TMSC specification.

# 8. REFERENCES


[1] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.

[2] 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.

[3] T. Alspaugh and A. Antón. Scenario networks for software specification and scenario management. Technical Report TR-2001-15, North Carolina State University, 2001.

[4] 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.

[5] K. M. Chandi and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM Transactions of Computer Systems*, 3(1):63–75, February 1985.

[6] 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.

[7] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.

[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] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[11] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal, 37(4)*, pages 269–277, 1994.

[12] S. Mauw and M. A. Reniers. Operational semantics for MSC'96. *Tutorials of the Eigth SDL Forum SDL'97: Time*

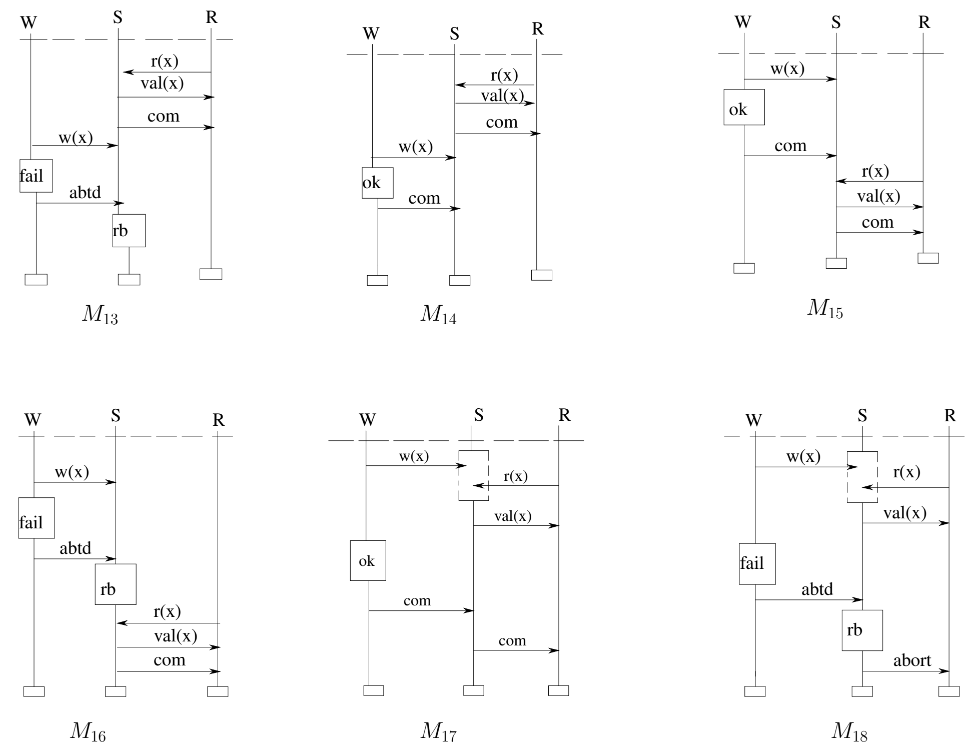


**Figure 3: TMSCs used for possible refinements.**

*for Testing - SDL, MSC and Trends*, pages 135–152, 1997.

[13] M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, Eindhoven University of Technology*, 1998.