

Triggered Message Sequence Charts

Bikram Sengupta

Rance Cleaveland

Abstract—This paper introduces Triggered Message Sequence Charts (TMSCs), a graphical, mathematically well-founded framework for capturing scenario-based system requirements of distributed systems. Like Message Sequence Charts (MSCs), TMSCs are graphical depictions of scenarios, or exchanges of messages between processes in a distributed system. Unlike MSCs, however, TMSCs are equipped with a notion of *trigger* that permits requirements to be made *conditional*; a notion of *partiality* indicating that a scenario may be subsequently extended; and a notion of *refinement* for assessing whether or not a more detailed specification correctly elaborates on a less detailed one. The TMSC notation also includes a collection of composition operators allowing structure to be introduced into scenario specifications, so that interactions among different scenarios may be studied. In the first part of this paper, TMSCs are introduced, and their use in support of requirements modeling is illustrated via two extended examples. The second part develops the mathematical underpinnings of the language.

Index Terms—Message Sequence Charts, scenarios, requirements modeling, formal semantics, refinement.

I. INTRODUCTION

Message Sequence Charts (MSCs) [3], [40] are a visual formalism for scenario-based specifications of distributed software systems. A single MSC depicts an exchange of messages that the processes in a system should engage in when the system is implemented; such an interaction, or scenario, represents a required facet of eventual system behavior. As a notation for recording scenarios precisely and clearly, MSCs have great appeal, as they permit the temporal relationships among message exchanges to be conveyed graphically. They are thus easier to review and to share with non-implementor system stakeholders than more textual requirements notations. At the same time, by describing scenarios in terms of the individual activities that system entities perform, MSCs convey high-level design guidelines to engineers responsible for the eventual implementation of system components. Other well-known graphical representations of behavioral aspects of systems include SDL [31], Petri Nets [39], Statecharts [23] etc.

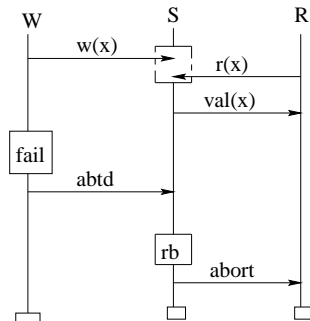


Fig. 1. An example MSC.

As an example, consider the MSC in Fig. 1. This MSC describes one scenario of message exchanges for a system consisting of three processes (also called *instances* in MSC terminology): a writer W , a server S , and a reader R . Each vertical axis represents the life-line of the corresponding instance, while the arrows connecting these lines represent the messages exchanged. In the depicted scenario, W updates the value of a variable x by sending the message $w(x)$ to S and R sends a request of form $r(x)$ to read x . S can receive the messages in either order, as represented by the dashed box at the beginning of S , which is called a *co-region*, and responds to $r(x)$ with a message of form $val(x)$ containing the new value of x . In the mean time W fails for some reason (the box representing a *local action*), and sends an *abtd* message to S indicating that it is aborting. S subsequently rolls back x to its previous value (by performing the local action rb), and sends a message to R asking it to abort, since R has been using a *dirty* value of x . This scenario, which can be delicate to explain clearly in words, becomes much easier to comprehend when depicted as an MSC.

The clarity of MSCs has made them a popular vehicle for conveying requirements of distributed systems, and this popularity has led to the development of standards for representing them [3] and to their inclusion in system modeling notations such as the Unified Modeling Language [7]. Research has also focused on equipping MSCs with a formal semantics, so that they may be reasoned about in a mathematically precise fashion [26], [32], [36], [40].

The evident utility of MSCs has also spurred the study of different extensions to the notation to further enhance its usefulness. One may specifically identify two respects in which the notation could be improved as a requirements notation.

- 1) *Conditionality*. MSCs excel in describing patterns of message exchanges. They are not well-suited, however, to describing conditional scenarios in which one exchange of messages entails the immediate occurrence of another. For example, a typical requirement for a telecommunications system might specify that if a caller activates a telephone connected to the system, then a dial tone will be emitted. MSCs are not equipped to convey such “triggering” behavior; one can only convey that the given sequence of exchanges between caller and handset is one possible behavior of the system. Consequently, MSC specifications must include ancillary commentary relating such causation information.
- 2) *Evolution*. During the elicitation process a requirements specification for a system may undergo several iterations. On the one hand, basic system functionality may be enriched; on the other hand, more abstract, user-level requirements may be translated into more

detailed, developer-level ones. Even after a specification is deemed complete, it often still continues to evolve in response to feedback from system stakeholders. In such cases one should ensure that subsequent requirements specifications are consistent with, or *refine*, previous ones. Such a capability is absent from traditional MSC specifications; as reflected in the technical development of [5], [36], [40], MSC system specifications are viewed as complete elaborations of system behavior that allow no deviation.

Moreover, unstructured collections of MSCs quickly become incomprehensible, leading to the possibility for contradictory specifications on the one hand (because contradicting scenarios have been specified) and incomplete specifications on the other (because the overhead of ensuring that a new MSC does not contradict any pre-existing one can dissuade engineers from making such modifications). It would be useful to have structuring mechanisms for MSC specifications so that related scenarios may be grouped together. We thus need to build on the set of operators offered by the MSC standard and provide practitioners with a rich set of constructs for managing complex specifications.

The goal of the present work is to develop a visual, scenario-based requirements formalism that retains the benefits of MSCs while also supporting conditional requirements, well-defined mechanisms for structuring requirements specifications, and a rigorous framework for reasoning about evolutionary development. At the heart of the resulting formalism, which we call *Triggered Message Sequence Charts* (TMSCs), lie *conditional scenarios*: TMSCs include a facility for representing requirements that dictate system behavior only when certain “triggering behaviors” occur. They also allow users to define *partial scenarios* that leave aspects of system behavior unspecified.

To structure TMSC-based specifications, we introduce composition operators, such as choice, sequential composition, and parallel composition, for combining sub-specifications. Several of the operators have also been studied by MSC researchers (e.g. [40]), and we show how these may be adapted to our framework. We also define a novel *conjunction* operator that, while of little interest in a pure MSC setting, has important uses in TMSC specifications.

We then define a refinement notion that may be used for determining when one TMSC specification is a consistent elaboration of another. The refinement relation we use comes from the process-algebra community, which has long studied notions of refinement and compositionality for distributed systems. The particular relation in our case is called the *must preorder* [22], [27] and defines one process as refining another if the former is “more deterministic than” the latter. The intuition underlying the must preorder is that nondeterminism represents “open development choices”; as choices are resolved, requirements specifications become more definite and thus more refined. The must preorder, and its close relative the failures preorder [29], have been extensively studied and shown to possess a number of desirable characteristics, such as safety and liveness property preservation (i.e. refinement is property-preserving), compositionality (i.e. refinement can

safely be done “in context”), and sensitivity to deadlocking behavior. Also, while the refinement relation is not decidable for the full TMSC theory (for essentially the same reasons that model checking of High-Level MSCs is not [5]), reasonable restrictions may be imposed on the TMSC notation that guarantee that the relation is computable. We discuss these issues in more detail in Section V. Other useful notions of MSC refinement (e.g. message refinement, structural refinement [32], [35]) that have been proposed in the literature, may also be suitably adapted to our framework.

We also show how our theory may be naturally used in support of two different requirements modeling paradigms. In the first, conjunction and conditional TMSCs are used to support the “gluing together” of existing TMSC component specifications into coherent system specifications. More specifically, component sub-specifications, when composed together, may exhibit undesirable interactions that an implementation should eventually remove. We show how additional TMSCs that remove undesired execution may be conjoined with the composition of sub-specifications to yield a specification having only desired behavior. In this case, the ancillary TMSCs may be viewed as “filters” that remove bad computations. The second requirements modeling approach illustrates how partial specifications and refinement may be used to define a principled, feature-by-feature design of a system while avoiding problems with feature interaction. In this case, features are initially modeled very coarsely as “stubs” (single messages activating the given feature, with the remaining computation left completely unspecified). These stubs are then “filled in” with the required high-level behavior; our refinement operator can be used to check that this filling in process yields a refined system that does not introduce any unwanted feature interactions.

Finally, we give an extensive account of the mathematics underpinning our TMSC theory. This treatment serves two purposes. On the one hand, it unambiguously defines the meaning of our notation. On the other, it also provides a platform-independent reference point for future tools that support the design and analysis of TMSC-based requirements documents.

The rest of the paper is divided into two parts. Sections II–IV present the syntax and intuitive meaning of TMSCs and TMSC refinement and describes via two extended examples how the notation may be used in requirements modeling. Section V then gives the precise mathematical foundations of the language. The results in the current paper improve and extend those in [9], [10].

II. DEFINING TRIGGERED MESSAGE SEQUENCE CHARTS

Like MSCs, TMSCs describe system scenarios in terms of the sequence of atomic actions (message sends and receives, and local actions) that each parallel process, or *instance* may engage in. TMSCs, however, enrich the graphical notation of MSCs by introducing capabilities for expressing *conditional* and *partial* behavior. Graphical constructs are also introduced for organizing collections of TMSCs into hierarchical system specifications. This section introduces TMSCs and the operators used to structure TMSC system specifications.

A. Visual Syntax of TMSCs

Fig. 2 gives an example of a TMSC. TMSCs enhance the syntax of MSCs with a *dashed horizontal line* that cuts through the instances and an *optional bar* at the foot of an instance (the bar is always present in an MSC). These simple syntactic extensions are explained below.

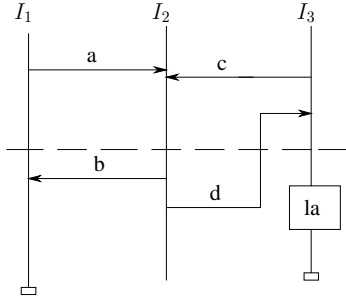


Fig. 2. An example TMSC.

a) Conditional Scenarios: TMSCs partition the sequence of atomic actions for each instance in a scenario into two subsequences: a *trigger* and an *action*. This partition is effected by a dashed horizontal line running through the instances of a TMSC as shown in Fig. 2. A TMSC scenario stipulates that in any system execution, if an instance performs the sequence of events constituting its *trigger*, then the subsequent behavior of the instance must include the sequence of events which constitute its *action*; otherwise there are no constraints on the behavior of the instance. The requirement that the action be performed is thus *conditional* on the occurrence of the trigger, and represents a *constraint* that the instance needs to satisfy in the scenario depicted by the TMSC.

b) Partial Scenarios: TMSCs have the ability to express whether a scenario is complete, or whether it is *partial* and thus *extensible*. This is achieved by the presence/absence of a small bar at the foot of each instance in a TMSC. The presence of a bar (as in instance I_1 in Fig. 2) indicates that the instance cannot proceed beyond this point in this scenario, while the absence (as in instance I_2) means that the behavior of this instance beyond the TMSC is left unspecified, i.e. there are no constraints on its subsequent behavior.

Fig. 2 may now 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.

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

In designing the visual syntax of TMSCs, we were confronted with representing conditionality on the one hand (triggers “imply” actions), and the passage of time on the other (the events within the instances of a TMSC occur in “top-to-bottom” order). These notions are orthogonal; in particular an event (say, a send) occurring in the action of one instance (below the dashed line) may lead to an event (the

corresponding receive) occurring in the trigger of another instance (above the dashed line). This leads to “upward” arrows such as the one labeled d in Fig. 2. The presence of these upward arrows should not be misconstrued as representing “time flowing backward”: they are merely used to record action-trigger dependencies between different instances, with one instance being seen as “triggering” another. The events in individual instances still occur in the usual top-to-bottom order as represented in the syntax.

It should be noted that traditional MSCs may be represented as TMSCs whose trigger is empty (i.e. the dashed trigger line is at the very top of the diagram) and whose instance lines all contain a termination bar at the foot. Thus TMSCs represent an extension of traditional MSCs. It also implies that MSCs and TMSCs may be freely intermixed within TMSC system specifications.

B. TMSC Expressions

Single TMSCs by themselves do not specify entire systems, but rather single aspects of intended system behavior. System specifications comprise collections of TMSCs, one for each scenario the system is expected to engage in. Two questions then naturally arise.

- 1) What is the intended meaning of a “collection” of TMSCs?
- 2) How can collections of TMSCs be structured?

The first question is important because an answer to it will determine when an implementation indeed satisfies a TMSC-based requirements specification. The second question is important for practical reasons: large systems will generally have many requirements, and structuring them so that they can be understood and reasoned about effectively is essential.

In this section we address these questions by defining a set of operators for building system specifications out of collections of TMSCs. The operators are driven by three considerations for organizing system definitions.

- **Structural.** A system might consist of interacting (parallel) components, each with its individual requirements specification.
- **Temporal.** A system might consist of several phases (e.g. initialization, operation, termination), occurring in a given order and specified independently.
- **Alternative.** Multiple specifications might be provided for a given behavior description, with the understanding that some or all of the requirements must be met.

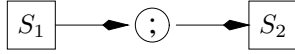
Table 1 lists the TMSC operators, and Fig. 3 defines the graphical form of a TMSC system specification. (For terminological convenience we often use the term “TMSC expression” to refer to TMSC system specifications.) In general, a TMSC expression is a graph consisting of an operator enclosed in a circle that is connected to other TMSC expressions, each enclosed in a rectangular box and which should be thought of as “subexpressions”. In most cases the edges in the graph are undirected and unordered. The sequential composition operator requires directed edges, with exactly one incoming and one outgoing, and the recursion operator has only one edge incident upon it.

TABLE I
TMSC OPERATORS.

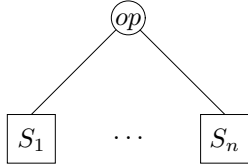
Symbol	Meaning
;	Sequential composition
X	Recursion
\parallel	Parallel composition
\mp	Delayed choice
\oplus	Internal choice
\wedge	Conjunction

1. Single TMSC

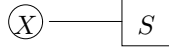
2. Sequence



3. Collection (op is $\parallel, \mp, \oplus, \wedge$)



4. Recursion (X a variable)



5. Variable

Fig. 3. The form of TMSC system specifications. S, S_1, \dots, S_n are TMSC system specifications

TMSC expressions have the following intended meaning. (In what follows, the type / diagram numbers refer to the numbers in Fig. 3.) Type 1 expressions are just single TMSCs, as defined earlier in this section. Diagrams of Type 2 represent “instance-wise” sequential composition; in such a diagram as depicted in Fig. 3 an instance in S_1 is intended to execute to completion before being resumed in S_2 , although an instance that terminates in S_1 may “continue” in S_2 even though other instances in S_1 have not finished. This operator may also be found in [3], [5], [40], and it is sometimes called *asynchronous concatenation*. Type 3 diagrams involve an application of one of the four given operators to the sub-diagrams S_1, \dots, S_n . Operator \parallel denotes parallel composition. The operators \mp and \oplus represent different forms of choice, with the former referred to as “delayed choice” and the latter as “internal choice”. In each case, the over-all specification indicates that a choice can be made among the sub-specifications; the operator indicates whether the choice can be made immediately, before execution begins (as in \oplus), or whether it should be delayed until a difference in execution forces one to be made (as in \mp). More precisely, \mp may be seen as forcing synchronization on shared actions of sub-specifications, with one sub-specification being selected and the others discarded when the former engages in an action that the others cannot. Delayed choice

is sometimes called deterministic, or angelic, choice and may be found in [3], [40] in the context of MSCs, as well as in the setting of refinement-oriented process-algebraic system specifications [27], [29]. Internal choice is also sometimes referred to as demonic choice and has been studied extensively in the context of process algebra [27], [29], where it has been shown to have many of the properties of “logical or.” Operator \wedge denotes conjunction, or “logical and”; a specification built with this operator indicates that each sub-specification must be satisfied independently by an eventual implementation.

Type 4 and Type 5 diagrams are related. In many cases one wishes to specify ongoing (i.e. nonterminating) behavior; an example might be the requirements for a transaction-processing system dictating that the system repeatedly process transactions without halting. Recursion diagrams allow this possibility. Specifically, Type 4 diagrams introduce a variable, X , that can then be used inside its sub-specification S . Whenever such a variable (Type 5 specification) appears, this is an indication that the body of the associated Type 4 diagram should be inserted at this point. Iterative behavior may also be found in [40]. Fig. 4 contains a sample TMSC expression.

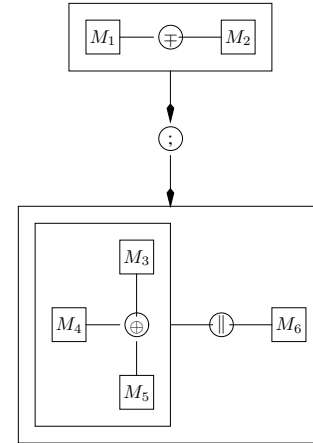


Fig. 4. A sample TMSC expression, where M_1-M_6 are TMSCs.

TMSC expressions have similarities with the graphical notation of High-Level Message Sequence Charts (HMSCs) [3]; in both cases, arrows are used to specify the flow of control through the specification. The main difference lies in how the composition operators are represented. In HMSCs, the operators are implicit: two specifications are composed sequentially if they are connected by an arrow; they are in parallel if they are placed next to each other in an enclosing frame; they represent an alternative choice if there is more than one outgoing arrow from a preceding node. In TMSCs, however, the operators are explicitly mentioned in the connection nodes (Types 2 and 3 in Fig. 2 above). This generalized syntax may be used to represent and distinguish between all the composition operators, including the two kinds of choice operators (\mp and \oplus) in TMSCs, without having to define implicit representations for each. Again, our representation of iterative behavior (Type 4 of Fig.2) is different from that in HMSCs, where loops are used to specify any behavior that is

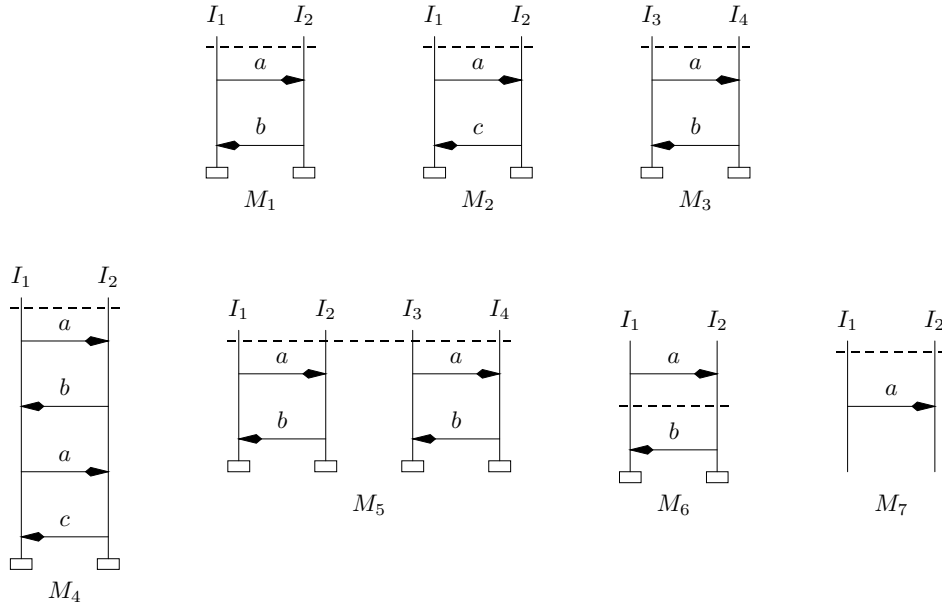


Fig. 5. Sample TMSCs.

repetitive.

Textually Representing TMSC Expressions: Graphical diagrams have great appeal, but referring to them in larger documents can be problematic. To ease reference to diagrams “in-line” we also introduce a textual representation of diagrams using the following BNF-like grammar.

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

Specifically, we write $S_1; S_2$ for diagrams of Type 2 in Fig. 3, $S_1 \text{ op } \dots \text{ op } S_n$ for diagrams of Type 3, and $rec X.S$ for diagrams of Type 4. We also use parentheses when necessary to disambiguate expressions.

C. Examples

In the remainder of this section we present several simple examples illustrating the different TMSC operators defined above. The examples refer to the sample TMSCs contained in Fig. 5. Note that M_1 and M_3 differ only in the names of their instances. Also, all scenarios except M_6 and M_7 are in fact usual MSCs, since they have empty triggers and terminate. (This is primarily because illustrating the intuitive behavior of most of the constructs is simpler with MSCs.)

Sequential composition: Consider the TMSC expressions $M_1; M_2$ and $M_1; M_3$. The behavior of the former may be seen as equivalent to the TMSC M_4 , while the latter mirrors that of M_5 . This is due to the fact that the sequential composition operator does not require all instances in the first

subexpression to terminate before allowing instances in the second subexpression to commence execution. Note that if M_1 were modified to M'_1 by omitting the bar at the foot of I_1 , then $M'_1; M_2$ would no longer be semantically equivalent to M_4 . The reason is that after finishing the “receive b ” event in M'_1 , instance I_1 would then be allowed to nondeterministically perform any number of arbitrary events before resuming execution in M_2 .

Finally consider the TMSC expression $M_1; M_6$. In this case, after completing M_1 instance I_1 has a choice. If it emits a , then it must await receipt of b and then terminate, since in M_6 the trigger for I_1 involves the sending of a . On the other hand, since M_6 is a conditional scenario (“if an a is sent, then await receipt of b ”) I_1 might perform an action other than a after finishing M_1 , or even terminate; in this case, M_6 does not constrain I_1 any further, and I_1 can behave completely nondeterministically.

Parallel: Consider TMSC expression $M_1 \parallel M_3$. The \parallel operator specifies parallel execution; in this case the specification would have behavior equivalent to M_5 .

What about $M_1 \parallel M_2$? Since the instances are shared, the parallel operator in effect requires that each instance make interleaved internal choices among the possibilities offered by M_1 and M_2 . Thus, when I_2 receives a , it can internally choose to send either a b or a c to I_1 in the next step. This is thus another source of non-determinism in the TMSC language, which may be removed during refinement (e.g. by fixing the order of messages b and c).

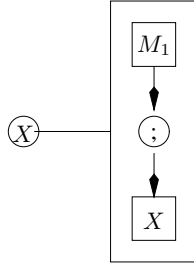
Choice: Consider the TMSC expressions $M_1 \mp M_2$ and $M_1 \oplus M_2$. Note that both M_1 and M_2 start with the delivery and receipt of a message a ; they differ on the behavior afterward. The first TMSC expression requires that I_1, I_2 engage in the send / receipt of a and, afterward, to enable both possibilities present in M_1 and M_2 ; this is because the

\mp operator requires choices to be delayed as long as possible. $M_1 \oplus M_2$, on the other, does not impose this constraint; thus, an implementation that mimics the behavior only of M_1 , for example, would be deemed correct. Note that a traditional MSC specification of a system given as a set $\{M_1, \dots, M_n\}$ of MSCs corresponds to the TMSC expression $M_1 \mp \dots \mp M_n$.

Conjunction: Consider TMSC expression $M_1 \wedge M_2$. A successful implementation of this expression should satisfy both M_1 and M_2 . In fact, no such implementation can execute both a followed only by b and a followed only by c , so this specification is impossible to meet. Indeed, one may show that for any pair of distinct MSCs (as opposed to TMSCs), this is the case.

Now consider TMSC expression $M_6 \wedge M_7$. M_6 indicates that if I_1 sends an a to I_2 , then I_2 must respond with a b , with both instances then terminating. M_7 says that I_1 should send an a to I_2 , and that afterward anything can happen. Conjoining these requirements yields a system in which I_1 sends an a to I_2 (because of M_7), and in response, I_2 sends b to I_1 , and both instances terminate (because the initial send satisfies the trigger of M_6). In other words, this TMSC expression is behaviorally equivalent to M_1 .

Recursion: Consider expression $\text{rec } X.(M_1; X)$, which is represented graphically as:



This expression describes a system $M_1; M_1; M_1; \dots$ consisting of an unending sequence of occurrences of M_1 .

III. REFINING TMSCS

The TMSC theory includes a notion of *refinement* for determining whether one TMSC specification is consistent with another. This section introduces this refinement relation, which is based on the *must preorder* given in [22], [27]. A full mathematical treatment may be found in Section V.

A. Refinement and Single TMSCs

Intuitively, a TMSC specification should refine another if it allows “fewer” implementations: any implementation of a more refined TMSC specification should also be allowed by a less refined one. We elaborate on this idea for single TMSCs in this subsection; TMSC expressions are considered in the next.

One may identify at least two respects in which TMSCs admit multiple implementations. On the one hand, the conditionality of a TMSC allows different implementations, since a system that fails to satisfy the trigger of a TMSC vacuously satisfies the TMSC. On the other hand, TMSCs may also be partial: a designer should then be free to “fill in” the partial aspects of a TMSC in whatever manner he or she wishes.

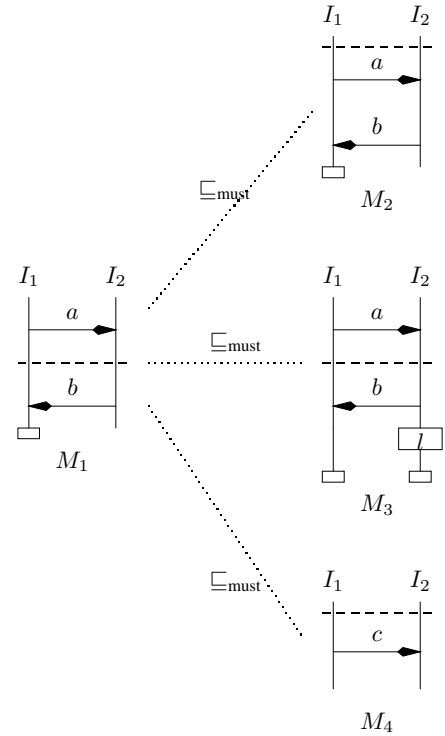


Fig. 6. Refining a TMSC.

To illustrate these points, consider Fig. 6. The requirement that M_1 expresses is: if I_1 sends a to I_2 , then it should wait to receive b from I_2 , and once I_1 receives b , it should terminate; also, if I_2 receives a from I_1 , then it should send b to I_1 , and there are no constraints on its subsequent behavior. One way to satisfy this requirement would be to make the trigger *mandatory*, i.e. we place the trigger line at the very top (as in TMSC M_2), so that I_1 and I_2 satisfy M_1 by actually exchanging the messages a and b as shown. M_2 indeed is a valid refinement of M_1 ; this is reflected in the diagram via the use of the $\sqsubseteq_{\text{must}}$ symbol. Similarly, M_3 also refines M_1 : since I_2 is allowed any behavior once it sends b , I_2 may perform a local-action l and terminate. Finally, M_4 , where I_1 sends c to I_2 , also vacuously refines M_1 by ensuring that the trigger of I_1 or I_2 is not satisfied.

There are other refinements for M_1 ; in general, unlike a MSC, a TMSC may be satisfied in many ways. But then, what do these refinements have in common? To answer this question we return to the basic intuition behind conditional and partial scenarios. As long as an instance has not satisfied its trigger in a conditional scenario, and after it has performed its trigger, followed by the action in a partial scenario, there are no constraints on its behavior. Semantically, we interpret such a behavior as being completely *non-deterministic*, and each of M_2 , M_3 and M_4 is a refinement of M_1 , as each is at least as deterministic as M_1 .

To sum up, conditional and partial behavior give rise to non-determinism by allowing multiple implementations, and we refine a TMSC by *reducing* the non-determinism inherent in it.

B. Refining TMSC Expressions

TMSC expressions, like single TMSCs, may be non-deterministic, and hence refinable in more than one way. The non-determinism in an expression may arise from a constituent TMSC (conditional/partial) scenario, as well as from the use of operators like \oplus (non-deterministic choice) and \parallel (non-deterministic interleaving). Since scenario-based notations like TMSCs are heavily used for early stage specifications, such non-determinism may be useful in representing a *coarse* initial model, which is successively refined as design progresses.

Given two TMSC expressions S_1 and S_2 , we say that S_2 refines S_1 (written $S_1 \sqsubseteq_{\text{must}} S_2$), if S_2 is *more deterministic* than S_1 . As in the case of single TMSCs, the definition of $\sqsubseteq_{\text{must}}$ will be made precise in Section V; intuitively, however, for any two TMSC expressions S_1, S_2 : $S_1 \oplus S_2 \sqsubseteq_{\text{must}} S_1$, $S_1 \oplus S_2 \sqsubseteq_{\text{must}} S_2$, $S_1 \oplus S_2 \sqsubseteq_{\text{must}} S_1 \mp S_2$, $S_1 \parallel S_2 \sqsubseteq_{\text{must}} S_1; S_2$ etc. One of the useful features of $\sqsubseteq_{\text{must}}$ is that it is *compositional*: if $S_1 \sqsubseteq_{\text{must}} S_2$, then $S_1 \text{ op } S_3 \sqsubseteq_{\text{must}} S_2 \text{ op } S_3$, where *op* maybe any TMSC operator. This allows us to refine a TMSC expression by independently refining its subexpressions.

C. Relating TMSC Expressions and Implementations

TMSC expressions are intended to capture system requirements. As engineers move to design and implementation phases, the notations used will change to more state-oriented ones (state machines, etc.). A virtue of our refinement ordering is that can be used to relate such “downstream” system artifacts to TMSC expressions or even relate heterogeneous specifications involving multiple notations, as has been shown elsewhere [14]. This point is not elaborated on further in this paper, but it relies on the fact that the original definition of the must preorder [22] relies on a very generic semantic model of system behavior, namely *acceptance trees*. As a consequence, any notation that can be given a semantics in terms of acceptance trees can use the must preorder as a refinement relation.

IV. REQUIREMENTS MODELING USING TMSCS

Having introduced TMSCs, TMSC expressions, and our refinement relation, we now illustrate how these concepts can efficiently capture requirements specifications for distributed systems. In particular, we show how the TMSC framework naturally supports two different methodologies in requirements modeling. The first approach, based on conditional scenarios, interweaves *constraint-based* and *prescriptive* system requirements within a single specification; it is useful when global system requirements must be preserved during the composition of sub-systems. The second approach supports *extensible specifications*, in which partial descriptions of requirements may be elaborated on in a succession of steps; it is thus suitable for the incremental development of complex system behavior. Both methodologies are based on the refinement notion of TMSC expressions. In both cases we define the basic methodology and illustrate its application on concrete case studies.

All claimed refinements presented in this section have been checked using the TRIM tool [12], [8], which provides automated support for analyzing system requirements expressed

in the TMSC notation. TRIM is implemented on top of the Concurrency Workbench of the New Century (CWB-NC) [16], [17], an easy-to-retarget verification tool for finite-state systems.

A. Constraint-Based Requirements Elicitation

This methodology is intended for systems that have pre-existing components whose requirements have been developed in isolation. The functionality of the overall system may then be described by appropriate composition of these *sub-specifications*. However, the specification thus obtained may not accurately capture desired over-all system behavior, which may need to preserve additional *global* constraints. In the early phases of the development of such a system, one would not wish to impose specific component-level mechanisms for satisfying these global constraints, but would instead prefer a means merely of recording the global constraints on system behavior.

The TMSC language offers a natural framework for supporting the above methodology. Modeling such systems using TMSCs is a three-step process. We first identify the sub-systems and give a specification of each. Next, we “glue” these specifications together to obtain a *coarse* specification of overall system behavior, which we call the initial, or *base*, specification. The sub-system specifications are composed using TMSC operators (e.g. \parallel , $;$, \oplus , \mp) in a way that reflects the functionality of the whole system. Note that although the scenarios that describe the individual sub-system behavior may be deterministic and look like normal MSCs, a sub-system specification may be non-deterministic (e.g. may involve a non-deterministic choice): how the sub-system behaves during a system run may depend on the *context* i.e. its interaction with the rest of the system. Finally, since the base specification may be overly permissive, we inspect this specification to determine if there are undesirable execution traces that may be generated. If so, we “weed-out” these incorrect execution sequences using conditional scenario constraints, and refine the initial coarse specification by adding these constraints using the \wedge operator.

The methodology outlined above has several appealing features. Firstly, by identifying the architectural components, practitioners can focus on specifying the behavior of smaller sub-systems to start with, and can also re-use existing legacy specifications. Secondly, the conditional scenarios conjoined to the base specification explicitly record global behavior that the eventual system should exhibit without prescribing a specific mechanism (in the form of modifications to sub-specifications) for doing so.

Case Study

In this section we illustrate the utility of constraint-based requirements capture for a version of the Automated Resuscitation and Stabilization System (ARSS) given in [10]. ARSS is intended to be part of the integrated Life Support for Trauma and Transport (LSTAT) patient stretcher system [2] and is responsible for automatically tracking a trauma victim’s blood pressure and transfusing fluids as necessary to stabilize the

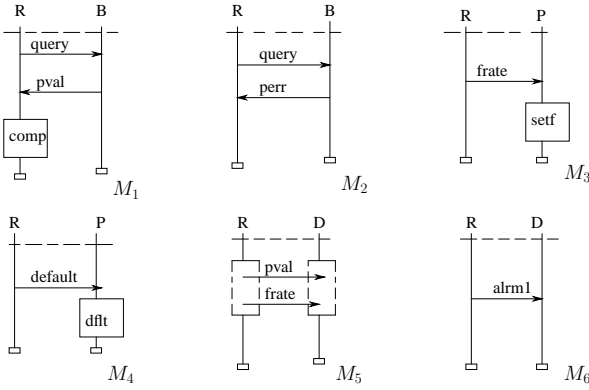


Fig. 7. Basic interaction between R, B, P and D

patient's condition. ARSS consists of a blood pressure measuring device (B), an infusion pump (P), a display and alarm unit (D) and a component R that controls the resuscitation process. R operates in cycles, communicating with the other units during a cycle: with B to poll the current pressure, with P to maintain a suitable flow rate of fluids through the patient's body, and with D to display appropriate messages or sound an alarm, if necessary.

To develop a requirements specification for ARSS, we first model the basic interactions between R and the other units during a single cycle of operation (Fig. 7).

a) R and B: The possible interactions between R and B are depicted in MSC-like TMSCs M_1 and M_2 . R sends a *query* message to B and B responds by either sending the current blood-pressure *pval* to R (as in M_1), or sending an error message *perr* to R (as in M_2) in case there was an error in reading the pressure (e.g. if the pressure source is lost for some reason). If R receives the correct value, it uses the current pressure to determine the rate at which fluids should be supplied to the patient; this is indicated by the local-action *comp*. The possible interactions between R and B are then given by the TMSC expression $RB = M_1 \mp M_2$. We use \mp because the choice between M_1 and M_2 is *delayed* until B sends either *pval* or *perr*.

b) R and P: R may send two types of messages to P :

- 1) under normal operation, it sends the rate (*frate*) at which P should supply fluids to the patient (M_3)
- 2) if R is unable to compute the correct rate, it asks P to set the rate to a pre-determined default value, *dflt* (M_4).

In each case, P responds by performing appropriate local actions to adjust the flow. We specify this behavior by $RP = M_3 \oplus M_4$; here \oplus is used because whether R can correctly compute the flow-rate or not depends on factors (in this case, R 's interaction with B) which are outside the purview of the interaction between R and P .

c) R and D: R may interact with D in the following ways. Under normal operating conditions, R sends the current pressure value *pval* and flow-rate *frate* for display to D . This scenario is depicted in M_5 . If the immediate attention of the care-giver is required, R instructs D to sound an alarm (M_6). The interaction between R and D is thus expressed by $RD = M_5 \oplus M_6$. As before, \oplus is used as the choice will be made

internally by R , depending on its interaction with the rest of the system. Note that M_5 uses *co-regions* in specifying the delivery and receipt of the messages between R and D . These are the dashed boxes on the R and D axes; as explained in the Introduction, a co-region specifies that all the actions incident upon it must occur, but their order is unspecified. In this case, both the *pval* and *frate* messages must be sent by R to D , but they may be sent in either order.

d) Base Specification: We now assemble the TMSCs described above to obtain an initial specification of the whole system. R begins by querying B about the current blood-pressure of the patient. It then sends messages to P and D to control the flow-rate of the pump and to display appropriate messages or warn the care-giver. An initial specification of ARSS for a single cycle of operation is thus given by

$$IS = (M_1 \mp M_2); ((M_3 \oplus M_4) \parallel (M_5 \oplus M_6)).$$

At the end of a cycle, the system has to begin a new cycle to maintain an appropriate flow of fluid to the patient. We specify this iterative behavior by applying recursion to the single-cycle specification IS to obtain the base specification BS :

$$BS = \text{rec } X.(IS; X)$$

e) Constraints: The base specification described above gives a high-level view of how the different sub-systems interact. However, by leaving out low-level details (e.g. what causes R to choose M_3 over M_4 , or vice versa, in any cycle of operation), the specification becomes too permissive; for example, it allows R to sound the alarm even when B has correctly read the pressure. This is because IS was obtained by gluing together several local views and does not properly account for global system requirements.

This is where TMSC based conditional scenarios are useful: we can eliminate the undesirable execution sequences by appropriately constraining system behavior under different *triggering* conditions. There are two such conditions in ARSS, during any cycle of operation.

- 1) B reads the pressure value correctly, which corresponds to the trigger in the conditional scenario C_1 .
- 2) There is an error in reading the pressure which is captured by C_2 .

In the first case, the corresponding action ensures that the correct flow-rate is computed and displayed. Similarly, the action in C_2 ensures that if B has lost the pressure source, then the D unit sounds the alarm to warn the care-giver, while P maintains a safe fluid-infusion rate till help arrives. We then elaborate our initial specification by adding in these constraints to get a detailed specification RS for a single cycle.

$$RS = IS \wedge C_1 \wedge C_2$$

In addition, it may be shown that $IS \sqsubseteq_{\text{must}} RS$, i.e. that RS refines IS and is thus consistent with the original very high-level "architectural" specification of the system.

Final Specification. The final ARSS specification (FS) may now be obtained by considering iteration over the refined single-cycle behavior RS :

$$FS = \text{rec } X.(RS; X)$$

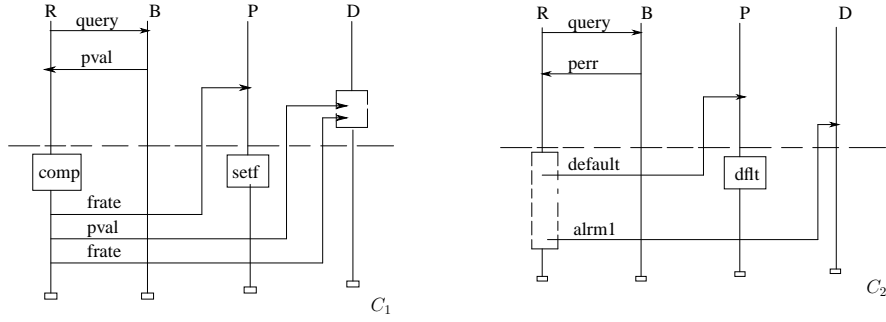


Fig. 8. TMSCs representing constraints on ARSS.

The TMSC semantics being compositional, $IS \sqsubseteq_{\text{must}} RS$ implies that $BS \sqsubseteq_{\text{must}} FS$. Note that since inter-instance communication and TMSC sequential composition are both “asynchronous” and the system behavior is cyclic, we would need to bound the size of the message buffers to compute the above relation in practice.

B. Elaboration-Based Requirements Specification

Elaboration-based requirements specification involves the iterative development of required system behavior in a feature-by-feature manner; while one feature is fully defined, others are left unspecified until later in the process. For example, normative system behavior might be defined, with exception-handling left unspecified until afterwards. The main motivation behind this approach is to facilitate a separation of design concerns, and thereby permit the development of complex systems to be broken down into manageable pieces. It also supports the addition of subsequent new features to existing systems. For these reasons, elaboration-based requirements development is quite routinely employed, albeit without any formal support, leaving open possibilities for unanticipated “feature interactions” to be introduced in different refinements. In this section, we show how TMSCs can be used to “fortify” the elaboration-based specification process by providing precise support for determining when a subsequent iteration preserves the behavior introduced in a previous iteration.

TMSC support for partial scenarios, together with the TMSC refinement relation, are the key ingredients in a formal elaboration-based process. In particular, partial TMSC scenarios describe an incomplete interaction sequence between instances: the behavior of the system beyond that depicted in the scenario is left unspecified. The refinement relation may then be used to compare a TMSC specification with an elaboration of it in which some partiality has been eliminated. Furthermore, because TMSC refinement is *compositional*, we may take an initial specification, replace a partial sub-specification within it by its more elaborate refinement, and thereby obtain a refinement of the overall specification, i.e. we may refine a specification by refining parts of it in isolation. We may repeat this process for any aspect of system behavior that has been partially described, till we reach the final, complete specification.

Such a step-wise development supports a natural separation of concerns during the derivation of a specification, while the

refinement relation ensures that desired behavior introduced in earlier refinements is preserved in later stages. Moreover, by constructing a chain of refinements, each refinement elaborating on a particular aspect of system behavior, designers have a record of what changes were made when, thereby facilitating subsequent changes and maintenance.

Case Study

We illustrate the application of TMSCs in elaboration-based requirements capture using the example of the Center Tracon Automation System (CTAS) [1][11]. Another case study involving the specification of a steam boiler may be found in [10].

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

In what follows, the interaction between CM and a generic *weather-aware client* CL will be specified incrementally via TMSCs, each step adding more detail to the previous one through an elaboration and refinement scheme. The TMSCs we use are shown in Fig. 9.

First Stage. CL initiates the communication by sending a CONNECTION (*con*) message to CM (M_1). On reception, CM sets the Weather Cycle status and CL’s weather status to “pre-initializing”, (for simplicity, we model this by a common local-action *pre-init*). It then *disables* a manual weather control button so that no manual changes are made. Next, it sets the Weather Cycle status and the CL’s weather status to “initializing” (local-action *init*), and sends a CTAS-GET-NEW-WTHR (*getnw*) message to CL (M_1). This message contains CL’s initial weather state. On reception, CL will send back a message indicating what the GET-STATUS is: we use *get-fail* for failure and *get-succ* for success. In case of failure, CM sets the Weather Cycle status to “done” and sends a CLOSE-CONNECTION (*close-con*) message to CM, which effectively terminates the communication (M_2). In case of success, the

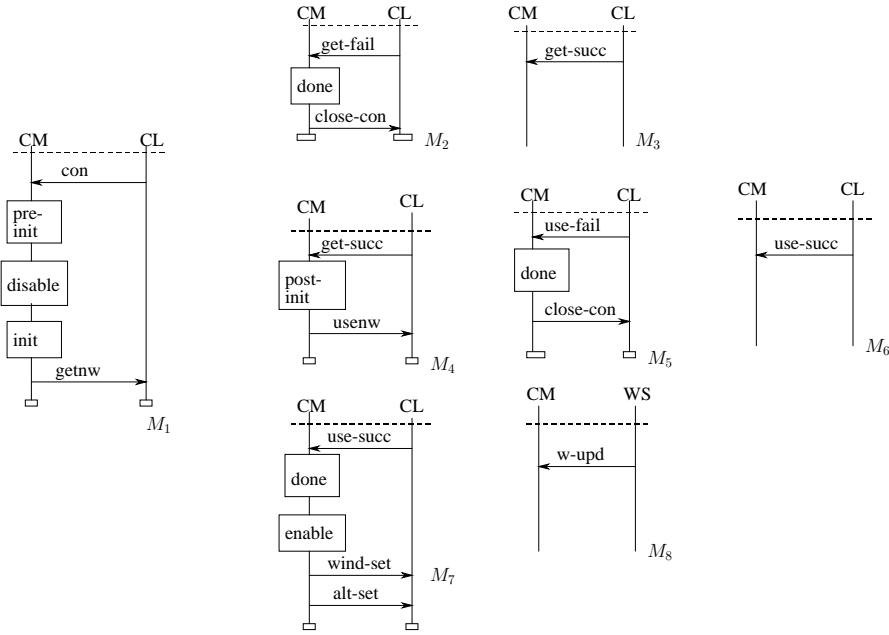


Fig. 9. TMSCs for CTAS

interaction proceeds to the next phase. We do not model this now, but indicate it through the partial scenario M_3 . Our initial specification is thus given by the TMSC expression

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

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

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

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

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

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

Third Stage. If CM receives a *use-succ* message, then it indicates the successful completion of the post-initialization phase. CM responds by setting both the Weather Cycle status and CL’s weather status to *done*, enables the manual weather control button, and sends a GROUND-WIND-SETTING (*wind-set*) message and an ALTIMETER-SETTING (*alt-set*) message to CL, to convey the client’s initial surface-winds and altimeter readings (M_7). The next phase (“updating”) would start only when CM receives a weather update (*w-upd*) from a weather source (represented here by the generic

component WS). CM would then need to convey the new weather information to CL; this may be done by subsequently extending the partial scenario (M_8) as appropriate, following the same approach. We have

$$R_2 = M_7; M_8$$

as an elaboration of M_6 . We now get

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

as a refinement of R_1 , and $S_3 = M_1; (M_2 \mp R'_1)$, as our new specification. We would thus get a refinement sequence $S_1 \sqsubseteq_{\text{must}} S_2 \sqsubseteq_{\text{must}} S_3 \dots$ as we build the specification in a stepwise manner, extending the behavior of the system one phase at a time.

V. FORMAL SEMANTICS OF TMSCS

The purpose of this section is to develop the mathematical foundations of TMSCs, TMSC expressions, and TMSC refinement.

MSC semantic accounts often rely on *traces* as the basis for defining the semantics of MSC system specifications [5]. A trace is a sequence of possible actions (message sends and receives, and local actions); an MSC system specification may then be seen as a set of traces that can arise in any eventual implementation. One may find variations of this basic framework; single MSCs naturally define a partial order on events [6] and thus sets of MSCs also define partial order languages that determine valid traces. However, a defining characteristic of trace-based approaches and associated notions of refinement (based on set inclusion, equality or reverse containment), is that in some sense, all traces in the specification are deemed “equal”; the specified behavior may be considered either optional (i.e. allowed) or required (i.e. mandatory) in the eventual implementation (depending on the refinement notion

chosen), but there is no easy way to specify behavior that is a heterogeneous mix of mandatory and optional content using pure trace-based semantics.

In contrast, the TMS language provides rich facilities for interweaving required and optional behavior. TMSCs with empty triggers may serve as the basis for defining behavior that is mandatory in an implementation, but TMSCs with non-empty triggers may be safely avoided by an implementation whenever possible; TMS expressions with internal choice can be refined by selecting one choice and omitting others, while a refinement is required to offer all alternatives in a delayed choice till execution forces a choice to be made. Again, a terminating scenario embodies exact behavior that the system is required to perform on satisfaction of the trigger, while a partial scenario allows several options for further extension. If we interpret TMS expressions as simply sets of traces, then conveying this mix of mandatory and optional behavior at each stage of execution becomes difficult, sometimes impossible (for example trace-based semantics is unable to distinguish between internal and delayed choice). On the other hand, the testing-based framework for process refinement [22], [27] does allow distinctions between required vs. allowed behavior to be easily made, and we use this work to define TMSCs precisely. The testing framework models systems as *acceptance trees*, which, in addition to recording trace behavior, also reflect the nondeterministic choices available to the system after such traces. These choices are encoded as *acceptance sets*, which represent the different choices for next actions that the system can select from. Acceptance trees include a notion of refinement, the *must preorder*, reflecting a notion of “more refined is more deterministic”; the must preorder ensures that a refinement satisfy all required behavior, but gives freedom in the choice of optional behavior. As a byproduct of our approach, we can easily adapt this notion to define refinement of TMS specifications.

We therefore use acceptance trees and the must preorder as the formal framework on which the semantics of the TMS language will be based. We show how to interpret individual TMSCs as acceptance trees, and how operators on TMSCs can be seen as functions over acceptance trees that respect the refinement ordering. We also include some remarks on decidability of this theory.

The semantics presented here both extends (by defining a recursive operator) and modifies (by improving the definition of \mp and \wedge) the technical development in [9].

A. Background

a) *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

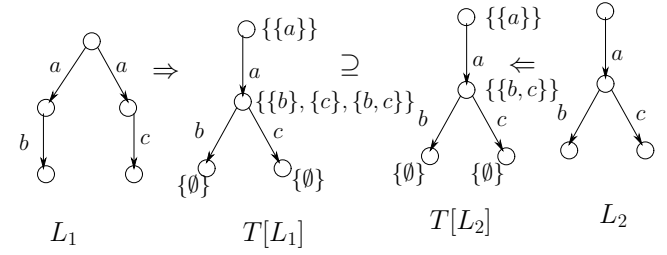


Fig. 10. The *must* preorder: $L_1 \sqsubseteq_{\text{must}} L_2$

$w, w_1, w_2 \in A^*$.

- ϵ, a empty, single-element sequence
- $\text{fst}(w), \text{lst}(w)$ first, last elements in $w \neq \epsilon$
- $w_1 \cdot w_2$ sequence concatenation
- $|w|_a$ 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
- $\text{subseq}(w)$ the set of (not necessarily contiguous) subsequences of w

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

b) *The Must Preorder*: The must preorder arises in the theory of process testing given in [22], [27], where 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. 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 [27].

Definition 1: A *labeled transition system (LTS)* is a tuple $\langle P, E, \longrightarrow, p_I \rangle$, where P is a state set, E an event set, $\longrightarrow \subseteq P \times E \times P$ 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 [27] relates different LTSs in terms of their responses to tests. The acceptance-set characterization of this relation relies on the following notions.

Definition 2: 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.

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

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 definition of $\sqsubseteq_{\text{must}}$ may now be given as follows [27].

Definition 3: 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 LTSSs, and let $E = E_1 \cup E_2$. Then $\mathcal{P}_1 \sqsubseteq_{\text{must}} \mathcal{P}_2$ 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 definition forms the basis for representing processes as *acceptance trees* [27], 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 $\mathcal{E}^* \rightarrow 2^{2^{\mathcal{E}}}$ 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 $s \in \mathcal{E}^*$, $T_1(s) \supseteq T_2(s)$.

It should also be noted that acceptance trees have strong connections to the theory of failure sets developed for CSP [29].

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 $T[\mathcal{P}_1] \supseteq T[\mathcal{P}_2]$. For example, in Fig. 10, $T[L_1]$ is the acceptance tree extracted from L_1 ; since L_1 can non-deterministically choose to perform b or c after execution of a , $T[L_1](a) = \{\{b\}, \{c\}, \{b, c\}\}$. In contrast, L_2 is deterministic, and $T[L_2](a) = \{\{b, c\}\}$. It follows that $L_1 \sqsubseteq_{\text{must}} L_2$ because $T[L_1] \supseteq T[L_2]$.

Acceptance trees are often used in support of proofs of *substitutivity*. Suppose that $\mathcal{P}_1, \mathcal{P}_2$ and \mathcal{Q} are LTSSs. 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 $T[op(\mathcal{P}, \mathcal{Q})] = f_{op}(T[\mathcal{P}], T[\mathcal{Q}])$.

B. From Single TMSCs to Acceptance Trees

In this section we first formally define TMSCs and then show how they may be assigned meanings in the form of acceptance trees. In what follows we fix finite sets \mathcal{I}, \mathcal{M} and \mathcal{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. (In practice this restriction poses no problem: any TMSC specification will contain a finite number of TMSCs, each with a finite set of instances.) The set \mathbb{E} of

events that can appear in TMSCs is defined to be

$$\mathbb{E} = \mathbb{S} \cup \mathbb{R} \cup \mathbb{L} \cup \mathbb{T}, \text{ where}$$

$$\mathbb{S} = \{\text{out}(I_i, I_j, m) \mid I_i, I_j \in \mathcal{I}, m \in \mathcal{M}\}$$

$$\mathbb{R} = \{\text{in}(I_i, I_j, m) \mid I_i, I_j \in \mathcal{I}, m \in \mathcal{M}\}$$

$$\mathbb{L} = \{\text{loc}(I_i, \ell) \mid I_i \in \mathcal{I}, \ell \in \mathcal{L}\}$$

$$\mathbb{T} = \{\text{end}(I_i) \mid I_i \in \mathcal{I}\}$$

Intuitively, \mathbb{S}, \mathbb{R} and \mathbb{L} contain the set of all send, receive and local events, respectively. Event $\text{out}(I_i, I_j, m)$ represents the sending of message m by instance I_i to I_j , while $\text{in}(I_i, I_j, m)$ denotes the receive of message m from I_i by I_j and $\text{loc}(I_i, \ell)$ reflects the execution of local event ℓ by I_i . Our semantics also uses events of form $\text{end}(I_i)$, which instance I_i emits when it terminates. Finally, we use “potential events” of form $\text{wait}(r)$, where $r \in \mathbb{R}$, to denote that an instance is capable of performing r once the corresponding send event occurs. By $\mathbb{W} = \{\text{wait}(r) \mid r \in \mathbb{R}\}$, we denote the set of all wait events. We also define a function $\text{active} : \mathbb{E} \rightarrow \mathcal{I}$ as follows; this function records which instance “performs” a given event e .

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

For any $E \subseteq \mathbb{E}$, $I \subseteq \mathcal{I}$, we write E_I for $\{e \in E \mid \text{active}(e) \in I\}$, the subset of E in which some instance in I is active. We call a set $C \subseteq \mathbb{E}$ a *co-region* if C is nonempty and every event in C involves the same active instance — formally, if for every $e_1, e_2 \in C$, $\text{active}(e_1) = \text{active}(e_2)$. We write \mathbb{C} for the set of all co-regions and extend the function active to co-regions in the obvious manner: $\text{active}(C) = I_i$ if for every $e \in C$, $\text{active}(e) = I_i$. Also, if $I_i \in \mathcal{I}$ is an instance and $\mathcal{C} \subseteq \mathbb{C}$ is a set of co-regions, we use \mathcal{C}_{I_i} for the set of co-regions C such that $\text{active}(C) = I_i$. We now define the mathematical representation of TMSCs as follows.

Definition 5: A *Triggered Message Sequence Chart* M is a tuple $\langle I, \text{trig}, \text{act}, \text{term} \rangle$, where:

- 1) $I \subseteq \mathcal{I}$ is the set of instances;
- 2) *Trigger function* $\text{trig} : I \rightarrow \mathbb{C}^*$ satisfies: $\text{trig}(I_i) \in (\mathbb{C}_{I_i})^*$ for all $I_i \in I$, and similarly for *action function* $\text{act} : I \rightarrow \mathbb{C}^*$.
- 3) *term* $\subseteq I$ is the *terminating set*.

Mathematically, a TMSC consists of a set of instances, functions trig and act assigning sequences of co-regions (sets of events that can occur in any order, recall) to each instance, and a set of instances that are intended to terminate. The restriction on trig and act ensure that an instance can only be assigned co-regions in which that instance is active. Any graphical TMSC can be translated into this formalism, although some mathematical TMSCs cannot be represented graphically.

In the remainder of this section, we show how to construct an acceptance tree from a single TMSC. In what follows we fix TMSC $M = \langle I, \text{trig}, \text{act}, \text{term} \rangle$. We begin by introducing notions of language and acceptance set for TMSCs; these rely on a notion of language of co-regions and co-region sequences.

Definition 6: Let $C \in \mathbb{C}$.

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

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

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

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

$L(C)$ contains all sequences in which every element in C appears once. If $W = C_1 \cdots C_n$ then $L(W)$ consists of words of the form $w_1 \cdots w_n$ where each $w_i \in L(C_i)$. The next definition introduces the notions of *trigger* and *action* languages of instances in M .

Definition 7: Let $I_i \in I$. Then $\text{trig}L_M(I_i)$, the *trigger language*, of I_i in M , is given by $\text{trig}L_M(I_i) = L(\text{trig}(I_i))$. The *action language*, $\text{act}L_M(I_i)$, of I_i in M is defined as $\text{act}L_M(I_i) = L(\text{act}(I_i))$.

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 8: Let $I_i \in I$. Then $L_M(I_i) \subseteq \mathbb{E}^*$ is defined as follows.

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

The *maximal language* of I_i is given by:

$$\text{max}L_M(I_i) = \{w \in L_M(I_i) \mid \text{lst}(w) = \text{end}(I_i)\}.$$

To understand these definitions, it should first be noted that $L_M(I_i)$ contains sequences of events in which I_i is active. These sequences must also be *well-terminated*: they contain at most one occurrence of $\text{end}(I_i)$, 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 $\text{end}(I_i)$ 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*, $\text{next}(L, w) \subseteq A$, of L after w is given by: $\text{next}(L, w) = \{a \in A \mid \exists w' \in L. w \cdot a \preceq w'\}$. 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 receives”. An instance can only emit a receive event if another instance has emitted the corresponding send; otherwise, this receive event is not

enabled. To capture this behavior, we introduce an additional parameter, $eR \subseteq \mathbb{R}$, into the acceptance-set function. A receive 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 receives $eR \subseteq \mathbb{R}$ as follows.

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

$ND(E, eR)$ is the acceptance set of a system that can non-deterministically decide to perform any *enabled* event in E , where any send, local or terminating event, and any receive in eR , is enabled, or *wait* for any receive event in E that is not enabled.

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

$$\begin{aligned} \text{Acc}(I_i, M, w, eR) = & \\ & \begin{cases} \emptyset & \text{if } w \notin L_M(I_i) \\ ND(\text{next}(L_M(I_i), w), eR) & \text{o.w.} \end{cases} \end{aligned}$$

$\text{Acc}(I_i, M, w, eR)$ is the acceptance set of instance I_i in TMSC M after w . The first clause handles the case when I_i is incapable of performing w , whereas the second clause computes the acceptance set based on events that are possible “next” after w , together with any potential receive events that are not enabled. This definition simplifies the corresponding one in [9], although it is semantically equivalent.

The above definition associates acceptance sets with the set of instances in TMSC M . This definition, however, does not say anything about the behavior of instances not mentioned in M (i.e. instances in $\mathcal{I} - I$). We assume that any such instance 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 10: Let $I_i \in \mathcal{I} - I$, $w \in \mathbb{E}^*$ and $eR \subseteq \mathbb{R}$. Then $\text{Acc}(I_i, M, w, eR)$ is defined as follows.

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

Having given the semantics of individual instances within TMSC M we now give an account of M itself by composing the acceptance trees of individual instances. We first introduce an operator \sqcup on acceptance sets: $\mathcal{A} \sqcup \mathcal{B} = \text{sat}(\mathcal{A} \cup \mathcal{B})$. 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

$$\mathcal{A} \bowtie \mathcal{B} = \bigsqcup_{A \in \mathcal{A}, B \in \mathcal{B}} \{A \cup B\}$$

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 also introduce the following definitions on event sequences.

Definition 11: Let $w \in \mathbb{E}^*$.

1) The *projection*, $w \lfloor I_i$, of w onto $I_i \in \mathcal{I}$ is the longest (not necessarily contiguous) subsequence of w containing

only events in which I_i is active:

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

- 2) The receive event $\text{in}(I_i, I_j, m)$ is called *enabled* by w if $|w|_{\text{out}(I_i, I_j, m)} > |w|_{\text{in}(I_i, I_j, m)}$. We use $e\mathcal{R}(w)$ to stand for all receive events enabled by w .
- 3) w is called *well-balanced* if for every $w' \preceq w$ such that $w' = w'' \cdot \text{in}(I_i, I_j, m)$, $\text{in}(I_i, I_j, m) \in e\mathcal{R}(w'')$.

In a well-balanced sequence of events, every receive is preceded by an “outstanding” send. We now define the acceptance set $\text{Acc}(M, w)$ as follows.

Definition 12: The acceptance set $\text{Acc}(M, w)$, of M after $w \in \mathbb{E}^*$ is defined as:

$$\text{Acc}(M, w) = \begin{cases} \emptyset & \text{if } w \text{ is not well-balanced} \\ \bowtie_{I_i \in \mathcal{I}} \text{Acc}(I_i, M, w \lfloor I_i, e\mathcal{R}(w)_{\{I_i\}}) & \text{o.w.} \end{cases}$$

Example: In Fig. 8 TMSC C_1 , let us consider

$$\begin{aligned} w = & \text{out}(R, B, \text{query}) \cdot \text{in}(R, B, \text{query}) \cdot \text{out}(B, R, \text{pval}) \\ & \cdot \text{in}(B, R, \text{pval}) \cdot \text{loc}(R, \text{comp}) \cdot \text{out}(R, P, \text{frate}) \\ & \cdot \text{in}(R, P, \text{frate}) \end{aligned}$$

Then,

$$\begin{aligned} w \lfloor R &= \text{out}(R, B, \text{query}) \cdot \text{in}(B, R, \text{pval}) \cdot \text{loc}(R, \text{comp}) \\ & \cdot \text{out}(R, P, \text{frate}) \\ w \lfloor B &= \text{in}(R, B, \text{query}) \cdot \text{out}(B, R, \text{pval}) \\ w \lfloor P &= \text{in}(R, P, \text{frate}) \\ w \lfloor D &= \epsilon \end{aligned}$$

We define

$$\begin{aligned} \mathcal{A}_R &= \text{Acc}(R, C_1, w \lfloor R, e\mathcal{R}(w)_{\{R\}}) \\ &= \{\{\text{out}(R, D, \text{pval})\}\} \\ \mathcal{A}_B &= \text{Acc}(B, C_1, w \lfloor B, e\mathcal{R}(w)_{\{B\}}) \\ &= \{\{\text{end}(B)\}\} \\ \mathcal{A}_P &= \text{Acc}(P, C_1, w \lfloor P, e\mathcal{R}(w)_{\{P\}}) \\ &= \{\{\text{loc}(P, \text{setf})\}\} \\ \mathcal{A}_D &= \text{Acc}(D, C_1, w \lfloor D, e\mathcal{R}(w)_{\{D\}}) \\ &= \text{sat}\{\{\text{wait}(\text{in}(R, D, \text{pval}))\}\} \{\{\text{wait}(\text{in}(R, D, \text{frate}))\}\} \\ & \quad \{\{\text{end}(D)\}\} \end{aligned}$$

where, we assume that

$$\mathbb{E}_{\{D\}} = \{\text{in}(R, D, \text{pval}), \text{in}(R, D, \text{frate}), \text{end}(D)\}$$

Then, we have

$$\begin{aligned} T[C_1](w) &= \mathcal{A}_R \bowtie \mathcal{A}_B \bowtie \mathcal{A}_P \bowtie \mathcal{A}_D \\ &= \text{sat}\{\{\text{out}(R, D, \text{pval}), \text{end}(B), \text{loc}(P, \text{setf}), \\ & \quad \text{wait}(\text{in}(R, D, \text{pval}))\}\} \{\{\text{out}(R, D, \text{pval}), \\ & \quad \text{end}(B), \text{loc}(P, \text{setf}), \text{wait}(\text{in}(R, D, \text{frate}))\}\} \\ & \quad \{\{\text{out}(R, D, \text{pval}), \text{end}(B), \text{loc}(P, \text{setf}), \\ & \quad \text{end}(D)\}\} \end{aligned}$$

C. The Must Preorder for TMSCs

Given Acc we may now define a version of the must preorder on TMSCs. This must preorder is a variant of the one presented in Section V-A.

Definition 13: An acceptance tree is a function $T : \mathbb{E}^* \rightarrow 2^{2^{\mathbb{E} \cup \mathbb{W}}}$ satisfying the following.

- 1) For any $w \in \mathbb{E}^*$, $\text{sat}(T(w)) = T(w)$.
- 2) For any $w, w' \in \mathbb{E}^*$, if $T(w) = \emptyset$ then $T(w \cdot w') = \emptyset$.
- 3) For any $w \in \mathbb{E}^*$, $e \in \mathbb{E}$, $T(w \cdot e) \neq \emptyset \Rightarrow \exists A \in T(w)$ 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 $\text{wait}(r)$, $r \in \mathbb{R}$. 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 [22], [27].

It is easy to associate an acceptance tree $T[M]$ to TMSC M : $T[M](w) = \text{Acc}(M, w)$. 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 14: Let M_1, M_2 be TMSCs. Then $M_1 \sqsubseteq_{\text{must}} M_2$ iff $T[M_1] \supseteq T[M_2]$.

D. Semantics of TMSC Expressions

To define the semantics of TMSC expressions we show how to interpret \mp , \oplus , \wedge , $;$ and $\text{rec}X$ as operations on acceptance trees. For brevity, we do not include the formalization of the \parallel operator here, but this definition may be found in [9]. The TMSC semantics presented here enhances the formal semantics in [9] in several respects. Firstly, the semantics for \mp in [9] was incomplete in the sense that it did not take the possible non-determinism of the sub-specifications into account. Our definition here removes that shortcoming. Secondly, for \wedge , we have added a constraint to ensure that the generated acceptance trees satisfy Definition 13. Most importantly, the TMSC expressions considered in [9] did not contain the recursion operator and hence could only model finite, terminating systems.

Technically, we show how a TMSC expression S may be converted into an acceptance tree $T_\sigma[S]$, where σ is an environment mapping variables to acceptance trees. The need for σ arises because TMSC expressions may contain variables. We consider each form of TMSC expression in turn.

a) *Single TMSC M :* $T_\sigma[M] = T[M]$ as defined in the previous section. For example, the acceptance tree of TMSC M_1 in Fig. 7 is shown in Fig. 11, where we assume $\mathcal{I} = \{R, B\}$.

b) *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. However, TMSCs may be non-deterministic, and hence, when assigning an acceptance tree to $S_1 \mp S_2$, (where S_1 and S_2 are TMSC expressions) we need to account for non-determinism in S_1 and S_2 . To do this, given a TMSC expression S and environment σ , we first define predicate $\text{may_reject}(S, \sigma, w)$ to hold when

$$\exists w_1, e, w_2. w = w_1 \cdot e \cdot w_2 \wedge \exists A \in T_\sigma[S](w_1). e \notin A.$$

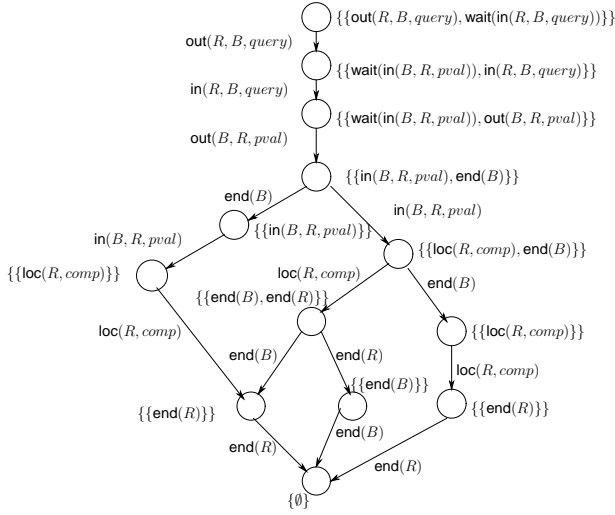


Fig. 11. Acceptance Tree of TMSC M_1 in Fig. 7

Intuitively, S in the context of σ may reject w if, after executing a proper prefix of w , S may reach a state where the next event in w is not enabled. Note that, $may_reject(S, \sigma, w) = true$ does not mean that $T_\sigma[S](w) = \emptyset$.

To define \mp formally in our setting, let us assume inductively that we have constructed $T_1 = T_\sigma[S_1]$ and $T_2 = T_\sigma[S_2]$. We now set:

$$T_\sigma[S_1 \mp S_2](w) = \text{sat} \left(\begin{aligned} & \{A_1 \cup A_2 \mid A_1 \in T_1(w), A_2 \in T_2(w)\} \\ & \cup \{A_1 \in T_1(w) \mid may_reject(S_2, \sigma, w)\} \\ & \cup \{A_2 \in T_2(w) \mid may_reject(S_1, \sigma, w)\} \end{aligned} \right)$$

$T_\sigma[S_1 \mp S_2](w)$ is computed in three parts: the first part, given by the pairwise union of $T_\sigma[S_1](w)$ and $T_\sigma[S_2](w)$, applies when both $T_\sigma[S_1](w)$ and $T_\sigma[S_2](w)$ are non-empty; since the choice between S_1 and S_2 is *delayed*, we take the pairwise-union, rather than the simple union, of the acceptance sets. However, if S_2 may reject w , then the choice between S_1 and S_2 may already have been resolved in favor of S_1 during the execution of w , and hence the acceptance set of S_1 after w may represent possible internal states of $S_1 \mp S_2$ on execution of w ; the second part in the definition of $T_\sigma[S_1 \mp S_2](w)$ captures this intuition. The third part captures the complementary case, when S_1 may reject w .

Example: Let us consider $RB = M_1 \mp M_2$, where M_1 and M_2 are as shown in Fig. 7, and let $w_1 = \text{out}(R, B, query) \cdot \text{in}(R, B, query)$, $w_2 = w_1 \cdot \text{out}(B, R, pval)$. Then,

$$\begin{aligned} T[M_1](w_1) &= \{\{\text{wait}(\text{in}(B, R, pval)), \text{out}(B, R, pval)\}\} \\ T[M_2](w_1) &= \{\{\text{wait}(\text{in}(B, R, perr)), \text{out}(B, R, perr)\}\} \\ T[M_1](w_2) &= \{\{\text{in}(B, R, pval), \text{end}(B)\}\} \\ T[M_2](w_2) &= \emptyset \end{aligned}$$

$$\begin{aligned} T[RB](w_1) &= \{\{\text{wait}(\text{in}(B, R, pval)), \text{wait}(\text{in}(B, R, perr)), \text{out}(B, R, pval), \text{out}(B, R, perr)\}\} \\ T[RB](w_2) &= \{\{\text{in}(B, R, pval), \text{end}(B)\}\} \end{aligned}$$

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

d) *Conjunction:* The \wedge operator allows the definition of composite specifications in which both sub-specifications must be satisfied. If S_1, S_2 are TMSC expressions, we define

$$T_\sigma[S_1 \wedge S_2](w) = \begin{cases} \emptyset & \text{if } \exists w_1, e, w_2. w = w_1 \cdot e \cdot w_2, T_\sigma[S_1 \wedge S_2](w_1) = \emptyset \\ \text{sat}(T_\sigma[S_1](w) \cap T_\sigma[S_2](w)) & \text{o.w.} \end{cases}$$

The first condition in the definition states that if there is a proper prefix w_1 of w such that $T_\sigma[S_1 \wedge S_2](w_1) = \emptyset$, then $T_\sigma[S_1 \wedge S_2](w) = \emptyset$; this ensures that acceptance trees generated for $S_1 \wedge S_2$ satisfy the second clause in the definition of acceptance trees (Definition 13). In case there is no such proper prefix of w , $T_\sigma[S_1 \wedge S_2](w)$ is given by the intersection of $T_\sigma[S_1](w)$ and $T_\sigma[S_2](w)$. Note that $T_\sigma[S_1 \wedge S_2](\epsilon) = T_\sigma[S_1](\epsilon) \cap T_\sigma[S_2](\epsilon)$ by the above definition.

e) *Sequential Composition:* The $;$ operator connects two TMSC expressions sequentially. Given a TMSC expression $S_1; S_2$, any instance $I \in \mathcal{I}$ 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 $A \# B$, of A and B as $A \# B = E$, where, $\forall I_i \in \mathcal{I}$, $E_{\{I_i\}}$ is given as follows.

$$E_{\{I_i\}} = \begin{cases} A_{\{I_i\}} & \text{if } I_i \in \text{activeset}(A) \wedge \text{end}(I_i) \notin A_{\{I_i\}} \\ (A_{\{I_i\}} - \{\text{end}(I_i)\}) \cup B_{\{I_i\}} & \text{o.w.} \end{cases}$$

The $\#$ 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. $A \# B$ 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 $\text{end}(I_i)$ 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 $\mathcal{A}; \mathcal{B}$, the sequential composition of \mathcal{A} and \mathcal{B} as

$$\mathcal{A}; \mathcal{B} = \bigsqcup_{A \in \mathcal{A}, B \in \mathcal{B}} A \# B$$

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

$$\text{activeseq}(w) = \{I_i \mid \exists e \in w. \text{active}(e) = I_i\}$$

We now define the following.

$$seq(w_1, w_2, I) = \left\{ \begin{array}{l} \emptyset \text{ if } \exists I_p \in \text{activeseq}(w_2). \text{end}(I_p) \notin w_1 \\ \{a \cdot w' \mid w' \in seq(w'_1, w_2, I)\} \\ \quad \text{if } w_1 = a \cdot w'_1, \nexists I_p.a = \text{end}(I_p), w_2 = b \cdot w'_2, \\ \quad \text{active}(b) \notin I \\ \{a \cdot w' \mid w' \in seq(w'_1, w_2, I)\} \\ \cup \{b \cdot w' \mid w' \in seq(w_1, w'_2, I)\} \\ \quad \text{if } w_1 = a \cdot w'_1, \nexists I_p.a = \text{end}(I_p), w_2 = b \cdot w'_2, \\ \quad \text{active}(b) \in I \\ seq(w'_1, w_2, I \cup \{I_p\}) \\ \quad \text{if } w_1 = a \cdot w'_1, \exists I_p.a = \text{end}(I_p), w_2 = b \cdot w'_2, \\ \quad \text{active}(b) \notin I \\ seq(w'_1, w_2, I \cup \{I_p\}) \\ \cup \{b \cdot w' \mid w' \in seq(w_1, w'_2, I)\} \\ \quad \text{if } w_1 = a \cdot w'_1, \exists I_p.a = \text{end}(I_p), w_2 = b \cdot w'_2, \\ \quad \text{active}(b) \in I \\ \{w_1 - \{\text{end}(I_p) \mid I_p \in \mathcal{I}\} \text{ if } w_2 = \epsilon \\ \{w_2\} \text{ if } w_1 = \epsilon, I = \mathcal{I} \end{array} \right.$$

Given two TMSC expressions S_1 and S_2 , we now define $T_\sigma[S_1; S_2](w)$ as follows:

$$T_\sigma[S_1; S_2](w) = \bigsqcup_{\langle w_1, w_2 \rangle \in \mathbf{L}} (T_\sigma[S_1](w_1) \# T_\sigma[S_2](w_2))$$

where

$$\mathbf{L} = \{ \langle w_1, w_2 \rangle \mid T_\sigma[S_1](w_1) \neq \emptyset, T_\sigma[S_2](w_2) \neq \emptyset, \\ w \in seq(w_1, w_2, \emptyset) \}$$

\mathbf{L} contains tuples of the form $\langle w_1, w_2 \rangle$, representing all possible ways in which w may have been produced by the sequential execution of S_1 and S_2 . For each such tuple, we use the $\#$ operator to compute the corresponding acceptance set of $S_1; S_2$. The overall acceptance set is then given by the non-deterministic choice of all the possible acceptance sets.

Example: Let us consider $S = M_1; M_3$, where M_1 and M_3 are as shown in Fig. 7, and let $w = \text{out}(R, B, \text{query}) \cdot \text{in}(R, B, \text{query})$, $\mathcal{I} = \{R, B, P, D\}$. Then we have $\mathbf{L} = \{ \langle w_1, w_2 \rangle \}$, where $w_1 = w$, $w_2 = \epsilon$,

$$\begin{aligned} T[M_1](w_1) &= \{ \{ \text{wait}(\text{in}(B, R, \text{pval})), \text{out}(B, R, \text{pval}), \\ &\quad \text{end}(P), \text{end}(D) \} \} \\ T[M_3](\epsilon) &= \{ \{ \text{out}(R, P, \text{frate}), \text{end}(B), \\ &\quad \text{wait}(\text{in}(R, P, \text{frate})), \text{end}(D) \} \} \\ T[M_1; M_2](w) &= \{ \{ \text{wait}(\text{in}(B, R, \text{pval})), \\ &\quad \text{out}(B, R, \text{pval}), \text{wait}(\text{in}(R, P, \text{frate})), \\ &\quad \text{end}(D) \} \} \end{aligned}$$

f) Recursion: The recursive TMSC operator $rec X.S$ may be used to represent nonterminating behavior. The variable X may occur freely within S , and if it does so, every reference to it results in a new cycle of operation (i.e. an *unrolling* of $rec X.S$). Instances that are not explicitly mentioned in S are only allowed to terminate, while those in S take part in the actual recursion.

To characterize $rec X.S$, we follow a traditional approach used in programming-language semantics: we define a series of acceptance trees that approximate the meaning of $T_\sigma[rec X.S]$ to greater and greater levels of detail, and then

take the *least upper bound* of these approximations. In order for this approach to work we first must show that such *chains* of acceptance trees ordered by the must preorder indeed have least upper bounds. We do this by proving that the set of acceptance trees \mathcal{T} ordered by $\sqsubseteq_{\text{must}}$ form a *complete partial order (cpo)*. (It should be noted that we need to establish this even though [27] also establishes that acceptance trees are a cpo. This is because our notion of acceptance tree differs in certain key respects.)

Theorem 1: $\langle \mathcal{T}, \sqsubseteq_{\text{must}} \rangle$ is a cpo.

Proof: We first show that $\langle \mathcal{T}, \sqsubseteq_{\text{must}} \rangle$ contains a least element. The least acceptance tree within our framework corresponds to the TMSC where each instance has an empty trigger and an empty action sequence, and none of the instances terminate in the TMSC. We use \perp to denote the acceptance tree for this TMSC; it is defined as follows.

$$T[\perp](w) = \begin{cases} \bowtie_{I_i \in \{I_i \mid \text{end}(I_i) \notin w\}} ND(\mathbb{E}_{\{I_i\}}, e\mathcal{R}(w)) \\ \quad \text{if } w \text{ is well-balanced and well-terminated} \\ \emptyset \quad \text{o.w.} \end{cases}$$

To see that $\perp \sqsubseteq_{\text{must}} T$ for any $T \in \mathcal{T}$, observe that for any other acceptance tree T , and any well-balanced and well-terminated sequence of events w , if $A \in T(w)$, then $A_{\{I_i\}} \in ND(\mathbb{E}_{\{I_i\}}, e\mathcal{R}(w))$ for any I_i for which $\text{end}(I_i) \notin w$. Thus $A \in T[\perp](w)$.

Next, we prove that any chain of acceptance trees $T_1 \sqsubseteq_{\text{must}} T_2 \sqsubseteq_{\text{must}} T_3 \sqsubseteq_{\text{must}} \dots$, has a least upper bound (*lub*) written as $\bigsqcup_i T_i$. Consider $\bigsqcup_i T_i$ defined by $(\bigsqcup_i T_i)(w) = \bigcap_i (T_i(w))$. It is easy to see that $(\bigcap_i T_i)(w) \subseteq T_i(w)$ for all $i = 1, 2, \dots$; hence, it is indeed an upper bound. To prove that it is a lub, assume that T' is another upper bound such that $T' \sqsubseteq_{\text{must}} \bigsqcup_i T_i$ and $\bigsqcup_i T_i \not\sqsubseteq_{\text{must}} T'$. Then there are w and $A \in T'(w)$ such that $A \notin (\bigsqcup_i T_i)(w)$. This implies that $A \notin T_i(w)$ for some i , and that $T_i \not\sqsubseteq_{\text{must}} T'$, which is a contradiction. ■

For technical reasons, we also need to be able to “relativize” the approximation process just described to a subset of the set of all instances. We begin with a few definitions. Given a TMSC expression S , $Ins(S)$ returns the set of instances that are explicitly mentioned in S and is defined as follows.

$$Ins(S) = \begin{cases} I & \text{if } S = \langle I, \text{trig}, \text{act}, \text{term} \rangle \\ Ins(S_1) \cup Ins(S_2) & \text{if } S = S_1 \text{ op } S_2, \text{ where } op \in \\ & \{ \oplus, \mp, \wedge, \parallel, ; \} \\ Ins(S') & \text{if } S = \text{rec } X.S' \\ \emptyset & \text{o.w.} \end{cases}$$

For any acceptance tree T^I that has been computed with respect to an universe of instances I , we may *lift* T^I to the set of instances J , with $J \supseteq I$, as $\text{lift}(J, T^I) = T^J$, where

$$T^J(w) = \begin{cases} \emptyset & \text{if (i) } \exists e \in w.\text{active}(e) \notin J, \text{ or (ii) } \exists e \in \\ & w.\text{active}(e) = I_p \wedge e \neq \text{end}(I_p) \wedge I_p \in J - I, \text{ or} \\ & \text{(iii) } w \text{ is not well-terminated} \\ \{S\} \bowtie T^I(w \upharpoonright I) & \text{o.w.} \end{cases}$$

and $S = \{ \text{end}(I_i) \mid I_i \in J - I \wedge \text{end}(I_i) \notin w \}$. Intuitively, *lift* “lifts” an acceptance tree defined using a restricted universe

of instances I to the larger universe of instances J by adding in $\text{end}(I_p)$ events $\forall I_p \in J - I$.

We are now in a position to define $T_\sigma[\text{rec } X.S]$. As in [27], we first compute successive approximations T_0, T_1, T_2, \dots of $\text{rec } X.S$, but in the universe of instances $I = \text{Ins}(S)$, and with \perp^I the bottom element in this restricted setting. We use $T_\sigma^I[S]$ to emphasize that I is the restricted set of instances.

$$\begin{aligned} T_0 &= \perp^I \\ &\vdots \\ T_{i+1} &= T_{\sigma[X \mapsto T_i]}^I[S] \\ &\vdots \end{aligned}$$

Since \perp^I is the least acceptance tree in the universe of instances I , $T_0 \sqsubseteq_{\text{must}} T_1$. Also, since the TMSC composition operators are substitutive with respect to the must preorder (Theorem 2 in Section V-E), and $T_0 \sqsubseteq_{\text{must}} T_1$, it follows that $T_1 \sqsubseteq_{\text{must}} T_2$, and, by induction, that $T_i \sqsubseteq_{\text{must}} T_{i+1}$ for any i . We thus have a chain $T_0 \sqsubseteq_{\text{must}} T_1 \sqsubseteq_{\text{must}} T_2 \sqsubseteq_{\text{must}} \dots$

Instances that are not explicitly mentioned in S are assumed to terminate in $\text{rec } X.S$. To define this in our framework, consider the function $Q(T) = \text{lift}(\mathcal{I}, T)$. We then have $Q_0 = \text{lift}(\mathcal{I}, T_0)$, $Q_1 = \text{lift}(\mathcal{I}, T_1)$, \dots , $Q_i = \text{lift}(\mathcal{I}, P_i)$, \dots , as the successive approximations $\text{rec } X.S$. Intuitively, each Q_i is the same as T_i for instances in I , while instances in $\mathcal{I} - I$, which are absent in T_i , are allowed to terminate in Q_i . Since $T_0 \sqsubseteq_{\text{must}} T_1 \sqsubseteq_{\text{must}} \dots \sqsubseteq_{\text{must}} T_i \sqsubseteq_{\text{must}} \dots$, it follows that $Q_0 \sqsubseteq_{\text{must}} Q_1 \sqsubseteq_{\text{must}} \dots \sqsubseteq_{\text{must}} Q_i \sqsubseteq_{\text{must}} \dots$. We now define $T_\sigma[\text{rec } X.S] = \bigsqcup_i Q_i$.

E. The Must Preorder and Properties of the Semantics

We may lift $\sqsubseteq_{\text{must}}$ to TMSC expressions as follows.

Definition 15: Let S_1, S_2 be TMSC expressions. Then $S_1 \sqsubseteq_{\text{must}} S_2$ if and only if $T_\sigma[S_1] \supseteq T_\sigma[S_2]$ for any σ .

We now show several properties of our semantics of TMSC expressions. The first shows that the must preorder is substitutive, and hence that our semantics is compositional.

Theorem 2: Let S_1, S_2 and S_3 be TMSC expressions such that $S_1 \sqsubseteq_{\text{must}} S_2$. Then the following hold:

- 1) $S_1 \mp S_3 \sqsubseteq_{\text{must}} S_2 \mp S_3$
- 2) $S_1 \oplus S_3 \sqsubseteq_{\text{must}} S_2 \oplus S_3$
- 3) $S_1 \wedge S_3 \sqsubseteq_{\text{must}} S_2 \wedge S_3$
- 4) $S_1 \parallel S_3 \sqsubseteq_{\text{must}} S_2 \parallel S_3$
- 5) $S_1; S_3 \sqsubseteq_{\text{must}} S_2; S_3$ and $S_3; S_1 \sqsubseteq_{\text{must}} S_3; S_2$
- 6) If $\text{Ins}(S_1) = \text{Ins}(S_2)$ then $\text{rec } X.S_1 \sqsubseteq_{\text{must}} \text{rec } X.S_2$.

Proof: We prove case 3; the others may be established as in [27], and we omit the details here.

$S_1 \sqsubseteq_{\text{must}} S_2$ means $\forall w, \sigma, T_\sigma[S_2](w) \subseteq T_\sigma[S_1](w)$. If $w = \epsilon$, then $T_\sigma[S_2 \wedge S_3](w) = T_\sigma[S_2](w) \cap T_\sigma[S_3](w)$, and $T_\sigma[S_1 \wedge S_3](w) = T_\sigma[S_1](w) \cap T_\sigma[S_3](w)$, so $T_\sigma[S_2 \wedge S_3](w) \subseteq T_\sigma[S_1 \wedge S_3](w)$. Otherwise, if $T_\sigma[S_2 \wedge S_3](w) = \emptyset$, then obviously $T_\sigma[S_2 \wedge S_3](w) \subseteq T_\sigma[S_1 \wedge S_3](w)$. Else, if $T_\sigma[S_2 \wedge S_3](w) \neq \emptyset$, then, $\forall w_1, e, w_2. w = w_1 \cdot e \cdot w_2$, $T_\sigma[S_2](w_1) \cap T_\sigma[S_3](w_1) \neq \emptyset$. Thus, $\forall w_1, e, w_2. w = w_1 \cdot e \cdot w_2$, $T_\sigma[S_1](w_1) \cap T_\sigma[S_3](w_1) \neq \emptyset$ (as, $T_\sigma[S_2](w_1) \subseteq T_\sigma[S_1](w_1)$)

This implies that $T_\sigma[S_1 \wedge S_3](w) = T_\sigma[S_1](w) \cap T_\sigma[S_3](w)$. Thus, $T_\sigma[S_2 \wedge S_3](w) \subseteq T_\sigma[S_1 \wedge S_3](w)$. ■

The next theorem establishes properties of the operators.

Theorem 3: Operators \mp, \oplus, \wedge and \parallel are commutative and associative, $;$ is associative, and \mp, \oplus and \wedge are idempotent.

Proof: We will prove the theorem for the \wedge operator; the arguments for the other operators are similar.

Commutativity follows immediately, since $T_\sigma[S_1](w) \cap T_\sigma[S_2](w) = T_\sigma[S_2](w) \cap T_\sigma[S_1](w)$.

Regarding associativity, we must show that for any w , $T_\sigma[(S_1 \wedge S_2) \wedge S_3](w) = T_\sigma[S_1 \wedge (S_2 \wedge S_3)](w)$. We prove the result by induction on w . When $w = \epsilon$ we argue as follows.

$$\begin{aligned} T_\sigma[(S_1 \wedge S_2) \wedge S_3](\epsilon) &= T_\sigma[S_1 \wedge S_2](\epsilon) \cap T_\sigma[S_3](\epsilon) \\ &= T_\sigma[S_1](\epsilon) \cap T_\sigma[S_2](\epsilon) \cap T_\sigma[S_3](\epsilon) \\ &= T_\sigma[S_1 \wedge (S_2 \wedge S_3)](\epsilon) \end{aligned}$$

Now assume $T_\sigma[(S_1 \wedge S_2) \wedge S_3](w) = T_\sigma[S_1 \wedge (S_2 \wedge S_3)](w) = A$; we must show the result for $w \cdot e$. If $A = \emptyset$, then $\forall e, T_\sigma[(S_1 \wedge S_2) \wedge S_3](w \cdot e) = T_\sigma[S_1 \wedge (S_2 \wedge S_3)](w \cdot e) = \emptyset$. If $A \neq \emptyset$, we know $T_\sigma[S_1 \wedge S_2](w) \neq \emptyset$, $T_\sigma[S_3](w) \neq \emptyset$. Then,

$$\begin{aligned} T_\sigma[(S_1 \wedge S_2) \wedge S_3](w \cdot e) &= T_\sigma[S_1 \wedge S_2](w \cdot e) \cap T_\sigma[S_3](w \cdot e) \\ &= T_\sigma[S_1](w \cdot e) \cap T_\sigma[S_2](w \cdot e) \cap T_\sigma[S_3](w \cdot e) \\ &= T_\sigma[S_1 \wedge (S_2 \wedge S_3)](w \cdot e) \end{aligned}$$

The argument for idempotence is similar and is omitted. ■

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

Theorem 4: Let S_1, S_2 and S_3 be TMSC expressions. Then $S_1 \wedge S_2 \sqsubseteq_{\text{must}} S_3$ if and only if $S_1 \sqsubseteq_{\text{must}} S_3$ and $S_2 \sqsubseteq_{\text{must}} S_3$.

Proof: We first prove that $S_1 \wedge S_2 \sqsubseteq_{\text{must}} S_3$ implies $S_1 \sqsubseteq_{\text{must}} S_3$ and $S_2 \sqsubseteq_{\text{must}} S_3$. So assume $\forall w, \sigma, T_\sigma[S_1 \wedge S_2](w) \supseteq T_\sigma[S_3](w)$. If $T_\sigma[S_1 \wedge S_2](w) = \emptyset$, then $T_\sigma[S_3](w) = \emptyset$, i.e. $T_\sigma[S_3](w) \subseteq T_\sigma[S_1](w)$ and $T_\sigma[S_3](w) \subseteq T_\sigma[S_2](w)$. If $T_\sigma[S_1 \wedge S_2](w) \neq \emptyset$, then $T_\sigma[S_1 \wedge S_2](w) = T_\sigma[S_1](w) \cap T_\sigma[S_2](w)$, and this implies, $T_\sigma[S_1](w) \cap T_\sigma[S_2](w) \supseteq T_\sigma[S_3](w)$, i.e. $T_\sigma[S_3](w) \subseteq T_\sigma[S_1](w)$ and $T_\sigma[S_3](w) \subseteq T_\sigma[S_2](w)$. Thus, $S_1 \sqsubseteq_{\text{must}} S_3$ and $S_2 \sqsubseteq_{\text{must}} S_3$.

We now prove, $S_1 \sqsubseteq_{\text{must}} S_3$ and $S_2 \sqsubseteq_{\text{must}} S_3 \Rightarrow S_1 \wedge S_2 \sqsubseteq_{\text{must}} S_3$. If $T_\sigma[S_1 \wedge S_2](w) \neq \emptyset$, then $T_\sigma[S_1 \wedge S_2](w) = T_\sigma[S_1](w) \cap T_\sigma[S_2](w)$. Thus, $T_\sigma[S_1 \wedge S_2](w) \supseteq T_\sigma[S_3](w)$. If $T_\sigma[S_1 \wedge S_2](w) = \emptyset$, there are two possibilities:

- $T_\sigma[S_1 \wedge S_2](\epsilon) = \emptyset$: This implies that $T_\sigma[S_1](\epsilon) \cap T_\sigma[S_2](\epsilon) = \emptyset$, i.e. $T_\sigma[S_3](\epsilon) = \emptyset$. Thus, $\forall w, T_\sigma[S_3](w) = \emptyset$.
- $T_\sigma[S_1 \wedge S_2](\epsilon) \neq \emptyset$. Then, $\exists w_1, e, w_2. w = w_1 \cdot e \cdot w_2$. $T_\sigma[S_1 \wedge S_2](w_1) \neq \emptyset \wedge T_\sigma[S_1 \wedge S_2](w_1 \cdot e) = \emptyset$, i.e. $T_\sigma[S_1](w_1 \cdot e) \cap T_\sigma[S_2](w_1 \cdot e) = \emptyset$. Hence, $T_\sigma[S_3](w_1 \cdot e) = \emptyset$, and thus $T_\sigma[S_3](w) = \emptyset$. ■

Finally, we consider the notion of decidability of the refinement ordering on TMSCs and TMSC expressions. For individual TMSCs, the relation is computable, using algorithms like the one given in [15]. For arbitrary TMSC expressions, the relation is not computable, for essentially the same reasons that model checking of High-Level MSCs is not [5]: the possibility of cyclic behavior in the presence of asynchronous sequential composition allows message buffers to grow without bound. In the case of TMSCs, another complicating factor is that recursive expressions involving parallel composition also opens

up the possibility for unboundedly many instances of parallel TMSCs to be active (this problem has been well-studied in the process algebra community [37]). On the other hand, reasonable restrictions can be placed on TMSC expressions that ensure decidability: buffers may be bounded, variables inside recursive expressions may be guarded, and the use of sequential and parallel composition restricted. The goal is to ensure that the language of the resulting system is regular, in the automata-theoretic sense of the word; it may then be shown that our refinement relation is also computable. We have implemented a tool, TRIM [12][13], that computes this relation for TMSC expressions that are restricted in this manner.

VI. RELATED WORK

TMSCs, and MSCs, formalize scenario-based approaches for requirements modeling, an active area of research in software engineering (e.g. [30], [24], [38], [43], [42]). A framework for classifying these approaches has been proposed in [20]. Several researchers have explored the use of pre- and post-conditions to relate scenarios [18], [19], [4]. In particular, scenario networks [4] use sequential and concurrent relationships between textual scenarios to describe allowable system behavior. Scenarios in such networks are labeled with pre- and post-conditions, which have superficial similarities with the TMSC notion of trigger/action, but the purpose of such conditions (typically state predicates) is to determine which scenarios can occur next at any given point, rather than to constrain and refine coarse specifications, as in TMSCs.

In the rest of this section, we will discuss related work mainly in the context of MSCs. The formal MSC language appears in a recommendation of the ITU [3]. A detailed description of the history, syntax and semantics of the language may be found in [40]. The account first presents the graphical syntax of the language, starting with basic MSCs, and describes ways to compose MSCs into High Level Message Sequence Charts or HMSCs. The formal semantics of the language is then defined in a deterministic process algebraic setting. While the formalization in [3], [40] is a very crucial step in the evolution of the MSC language, it also presented opportunities for further enhancement. For example, basic MSCs in the ITU standard are expressively weak, offering only a visual partial ordering of events. As mentioned in the Introduction, the technical development there does not provide a natural way for expressing conditional/partial behavior, and for the step-wise refinement of system behavior. TMSCs were motivated by a need to extend the MSC language along these directions; in particular, we chose the acceptance tree framework because of its natural support for representing non-deterministic system behavior (inherent in conditional and partial scenarios) and the precise notion of refinement encapsulated within the *must* pre-order. Note that, since process algebra lies at the heart of the acceptance tree model, there is a natural synergy between the TMSC semantics and the standardized MSC semantics. Also, since the TMSC language includes the structural constructs (e.g. weak sequential, parallel, delayed choice etc.) found in the ITU standard (with only a slightly different semantics

for the parallel operator), it is straightforward to encode a HMSC as a TMSC expression. Section II-B contrasted the graphical syntax of these languages. Some of the other models of concurrency that have been proposed as a semantic domain for MSCs include Petri Nets [26], Büchi automata [33], etc.

Harel et. al [21], [28] and Krueger [32] have also proposed MSC variants that support trigger/action-like behavior. More precisely, [21] introduces Live Sequence Charts (LSCs), which primarily seek to distinguish between “live” or mandatory behavior and provisional behavior. The work is motivated by the view that a designer’s interpretation of behavior often undergoes a shift from “existential” to “universal”, as system design proceeds and knowledge about the system evolves; accordingly, the authors propose the notions of “universal” charts (behavior that must be satisfied in all “runs” of a system) and “existential” charts (behavior that must be satisfied in at least one run), and also associate with these, trigger-like activation messages or pre-charts. A LSC specification is then defined as a set of such LSCs, and separate notions of satisfaction are defined for universal and existential charts. A system “satisfies” a specification by satisfying each individual chart therein. [28] proposes a trace-based semantics for LSCs, and also shows how an equivalent semantics may be expressed using temporal logic.

Krueger [32] proposes a variant of MSCs that is based on the mathematical model of “timed streams”. The work has several features that go beyond the standard language e.g. guarded MSCs, guarded alternatives and loops, preemption, “trigger” composition etc. It also proposes three notions of refinement: property, message and structural. Property refinement addresses the reduction of possible behaviors of the overall system (e.g. by removing alternatives), message refinement denotes the substitution of a whole interaction sequence for a particular message, and structural refinement occurs when a single instance in a MSC is replaced with a set of other instances. Property refinement is shown to be compositional, with the exception of the “trigger” operator.

Although the notions of trigger/actions in TMSCs have similarities with the notions of pre-chart/chart body in LSCs and trigger composition in Krueger’s work, our perspective on distributed system modeling is different, and this is reflected both in the way we interpret triggering behavior, as also the semantic model we adopt to formalize TMSCs. First, the chart-body in [21] and [32] needs to be activated only after the pre-chart/triggering chart has been completely executed. This corresponds to “strong” triggering, in analogy with the MSC terminology of “strong” sequential composition [32] of the pre- and post charts. In contrast, the trigger/action requirements in TMSCs are localized to individual instances, and this may be interpreted as “weak” triggering, in line with the “weak” sequential composition we support. Our decision was motivated by the fact that scenario-based specifications are most heavily used for distributed systems, where instances have localized control and execute asynchronously. Note that this approach is synergistic with the MSC standard [3] that prescribes the use of the weak sequential composition operator.

Again, trace-based semantics (used in LSCs) is weaker than testing semantics (adopted in TMSCs); in particular, trace-

based semantics is not sensitive to deadlock (a critical property for distributed systems) unlike testing semantics (which, in fact, is the coarsest semantics, as compared to trace semantics, that has this property [34]). As such, our semantics is well-tailored for extensions like Compositional MSCs [25], and also for other notations like Communicating State Machines, where such properties may become relevant. This, in fact, also makes the semantics very suitable for exploring *heterogeneous specifications* of such notations, a concept we have begun to explore. Moreover, LSCs do not support HMSCs, a point also noted in [32]. The TMSC language, on the other hand, provides strong support for developing structured specifications, and this makes it useful when dealing with large requirements specifications. Methodologically, LSCs seem more suited for temporal-logic style satisfaction-based approaches, while the TMSC framework, as evident from the technical development in this paper, is geared more towards refinement-based approaches.

The MSC language defined in Krueger’s work [32] does provide operators for developing structured specifications, and also considers the refinement of behavior. However, the timed-stream theory in [32] is similar in essence to trace-based semantics in the sense that it also treats MSCs as representations of execution *sequences*. The theory is thereby unable to support the notion of “delayed” choice prescribed by the MSC standard, which requires the branching behavior of execution *trees* to be considered. The TMSC theory, in contrast, is able to support both internal (non-deterministic) and “delayed” (deterministic) choice, and this brings flexibility to the development process. Also, the basic “building block” of the timed stream theory of MSCs is a single message occurring on a channel, as opposed to a basic MSC, and appropriate operators are used to specify temporal order, or the lack thereof, between messages occurrences; however, no explicit distinction is made between message sends and receives (as in TMSCs, or standard MSCs), and as such, expressing fine-grained ordering between send and receive events associated with different messages is difficult.

Another approach to MSC refinement may be found in the refinement ordering for interworkings [35]. Interworkings may be seen as a close variant of MSCs, with the key difference being that interworkings support synchronous instead of asynchronous communication. [35] defines a refinement notion based on selective hiding of communications and collapsing of several instances into a single one. Intuitively, one system is said to refine another if it is possible to collapse the former into an interworking that is semantically equivalent to the latter, with bisimulation [37] being the base equivalence used in that paper. The intuition behind this refinement ordering is that refining a specification may involve adding more structure; ensuring consistency with an earlier specification amounts to showing that when the new structure is “ignored”, the behavior is the same. This notion differs significantly from ours, since its focus is on structural changes to MSCs rather than the modification of behavior of instances within an MSC. Similar notions could easily be adapted to our framework, with the equivalence induced by the must preorder used in lieu of bisimulation.

Another recent approach that supports the use of assume-guarantee mechanisms within scenario-specifications is Template MSCs [6]. This extends the TMSC notation with the notion of “gaps” that can be interleaved with message sends and receives, and allow an arbitrary but finite amount of communication to be expressed. This extension can allow more concise specifications to be written, through the use of only those events that are strictly needed to identify a scenario, and the use of gaps as placeholders for other messages that remain unspecified. [6] also illustrates how Template MSCs may be used in the TMSC setting, where a system description is obtained through the combination of MSCs depicting local behaviors of directly interacting processes, and then superimposing assume-guarantee template formulas to restrict the combination of local behaviors.

Finally, [41] presents a notation for *negative scenarios* that uses trigger-like preconditions. However, the motivation, as stated in that paper, is primarily “to provide a language for documenting rejected implied scenarios and eliciting further implied scenarios,” and unlike our work, is for determining whether certain system behaviors, not present in a scenario specification, may appear in possible implementations.

VII. CONCLUSIONS

We have presented Triggered Message Sequence Charts (TMSCs) as a flexible formal basis for scenario-based requirements modeling. TMSCs introduce notions of conditionality and partiality through simple syntactic extensions to MSCs. The language also offers a suite of operators to structure requirements specifications. The semantics is defined by translating TMSC expressions to acceptance trees, which yields a precise notion of refinement. A useful feature of the TMSC semantics is that it is compositional; a TMSC expression may thus be refined by refining its sub-expressions.

TMSCs naturally support two different styles of requirements modeling. In the first, a coarse base specification of the overall system is initially developed by gluing sub-system models; this is then refined by adding conditional scenario constraints to eliminate undesirable execution sequences. In the second, partial scenarios are used in support of an incremental style of requirements specification, in which features are incrementally elaborated on in a requirements specification.

The TMSC framework offers a number of possibilities for future research. For example, it would be interesting to investigate how our notion of refinement may be used to extract specifications for the individual instances from a TMSC expression. Another natural extension of our work would be to generate test suites from a TMSC specification that may be used to check its proposed refinements. It would be especially useful if these tests could be derived in a compositional manner. Another avenue for research involves the adaptation of other operators from process algebra to the MSC / TMSC setting. In particular, process algebras typically include a *hiding* operator that permits the hiding of implementation details. Introducing a hiding operator into the TMSC theory, for example would allow more flexibility in determining refinement relationships among TMSC expressions, since “extraneous” elements (such as “local instances”,

“unobservable actions”, etc.) could be hidden. Finally, the semantic framework presented here is highly suited for the study of heterogeneous specifications, where scenario-based TMSC specifications may be seamlessly integrated with other modeling notations that may be given a similar semantics. While we investigate such a heterogeneous framework for TMSCs and state-machines in [14], it would be interesting to study how to extend the framework to cater to other notations as well.

REFERENCES

- [1] Center-tracon automation system (ctas) for air traffic control. URL: <http://ctas.arc.nasa.gov/>.
- [2] Integrated medical systems inc.-lstat. URL: <http://www.lstat.com/lstat.html>.
- [3] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
- [4] 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*, 2001.
- [5] R. Alur and M. Yannakakis. Model checking of message sequence charts. *Tenth International Conference on Concurrency Theory*, LNCS 1664, Springer:82–97, 1999.
- [6] B.Genest, M.Minea, A.Muscholl, and D.Peled. Specifying and verifying partial order properties using template mscs. *International Conference on Foundations of Software Science and Computation Structures*, LNCS vol. 2987:195–210, 2004.
- [7] Grady Booch, Ivar Jacobson, and James Rumbaugh. The unified modeling language user guide.
- [8] B.Sengupta. Triggered message sequence charts. *Ph.D Thesis, State University of New York (SUNY) Stony Brook*, 2003.
- [9] B.Sengupta and R.Cleaveland. Triggered message sequence charts. *ACM SIGSOFT 2002 (FSE-10)*, pages 167–176.
- [10] B.Sengupta and R.Cleaveland. Refinement-based requirements modeling using triggered message sequence charts. *IEEE International Requirements Engineering Conference*, 2003.
- [11] B.Sengupta and R.Cleaveland. Towards formal but flexible scenarios. *2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.
- [12] B.Sengupta and R.Cleaveland. TRIM: A tool for triggered message sequence charts. *Proceedings of 15TH Computer Aided Verification Conference (CAV'03)*, 2003.
- [13] B.Sengupta and R.Cleaveland. Executable requirements specifications using triggered message sequence charts. *Second International Conference on Distributed Computing and Internet Technology (ICDCIT)*, LNCS vol.3816, 2005.
- [14] B.Sengupta and R.Cleaveland. An integrated framework for scenarios and state-machines. *Fifth International Conference on Integrated Formal Methods (IFM)*, LNCS vol.3771:366–385, 2005.
- [15] R. Cleaveland and M.C.B. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.
- [16] 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), 1993.
- [17] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, January 2002.
- [18] C.Rolland and C.Ben Achour. Guiding the construction of textual use case specifications. *Data and Knowledge Engineering Journal*, 25(1-2):125–160, 1998.
- [19] C.Rolland, C.Souveyet, and C.Ben Achour. Guiding goal modeling using scenarios. *IEEE Transactions on Software Engineering*, 24(12), 1998.
- [20] C.Rolland and C.Ben Achour et al. A proposal for a scenario classification framework. *Requirements Engineering journal*, 3, No. 1:23–47, 1998.
- [21] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
- [22] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.
- [23] D.Harel. Statecharts:a visual formalism for complex systems. *Science of Computer Programming*, 8, pages 231–274, 1987.
- [24] J.C.S. do Prado Leite and G.Rossi et al. Enhancing a requirements baseline with scenarios. *Third IEEE International Symposium on Requirements Engineering*, 1997.
- [25] E.Gunter, A.Muscholl, and D.Peled. Compositional message sequence charts. *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, LNCS vol. 2031:496–511, 2001.
- [26] J. Grabowski, P. Graubmann, and E. Rudolph. Towards a petri net based semantics for message sequence charts. *SDL'93 Using Objects, Proc. of the Sixth SDL Forum*, pages 179–190.
- [27] M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
- [28] H.Kugler, D.Harel, A.Pnueli, Y.Lu, and Y.Bontemps. Temporal logic for scenario-based specifications. *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.
- [29] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
- [30] I.Jacobson. The use case construct in object-oriented software engineering. *Scenario-Based Design: Envisioning Work and Technology in System Development*, pages 309–336, 1995.
- [31] ITU-T. Recommendation z.100: Specification and description language (sdl). 1994.
- [32] I. Kruger. Distributed system design with message sequence charts. *PhD Thesis, Technical University of Munich*, 2000.
- [33] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
- [34] M. Main. Trace, failure and testing equivalences for communicating processes. *International Journal of Parallel Processing*, 16(5):383–400, 1987.
- [35] S. Mauw and M. Reniers. Refinement in interworkings. In U. Montanari and V. Sassone, editors, *CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 671–686, Pisa, Italy, August 1996. Springer-Verlag.
- [36] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), pages 269–277, 1994.
- [37] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.
- [38] N.Maiden, S.Minocha, K.Manning, and M.Ryan. CREWS-SAVRE: Systematic scenario generation and use. *International Conference on Requirements Engineering*, pages 148–155, 1998.
- [39] W. Reisig. Petri nets: An introduction. vol. 4 of *EATCS Monographs in Theoretical Computer Science*, 1985.
- [40] M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, Eindhoven University of Technology*, 1998.
- [41] S.Uchitel, J.Kramer, and J.Magee. Negative scenarios for implied scenario elicitation. *ACM SIGSOFT 2002 (FSE-10)*.
- [42] J.Kramer S.Uchitel and J.Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2), 2003.
- [43] T.Alspaugh, A. Antón, T.Barnes, and B.Mott. An integrated scenario management strategy. *IEEE International Symposium on Requirements Engineering*, pages 142–149, 1999.

Bikram Sengupta obtained a B.E. in Computer Science and Engineering from Jadavpur University, India, in 1998, and M.S. and Ph.D. degrees in Computer Science from the State University of New York (SUNY), Stony Brook in 2000 and 2003, respectively. He is currently working as a Research Staff Member at IBM Research, India.

Rance Cleaveland obtained B.S. degrees in Mathematics and Computer Science in 1982 from Duke University, and M.S. and Ph.D. degrees in Computer Science from Cornell University in 1985 and 1987, respectively. In 1991 he won a National Young Investigator award from the National Science Foundation and a Young Investigator Award from the Office of Naval Research. He is currently a Full Professor of Computer Science at the State University of New York.