

Refinement-Based Requirements Modeling Using Triggered Message Sequence Charts

Bikram Sengupta Rance Cleaveland

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
Fax: 1-631-632-8334
{sbikram,rance}@cs.sunysb.edu

Abstract

Triggered Message Sequence Charts (TMSCs) are a visual, mathematically precise notation for capturing system requirements as conditional and partial scenarios. This paper shows how TMSCs may be used to formalize two different requirements modeling methodologies. The first approach combines prescriptive (“do this”) and constraint-based (“don’t do that”) requirements within a single specification; it is useful for composing localized subsystem requirements with global system ones. The second approach supports layered specifications in which partial descriptions of requirements may be elaborated on in a succession of steps; it is suitable for the incremental development of complex behavior in which “error” scenarios are “layered on top of” normative ones. Both methodologies derive their formal robustness from the notion of semantic refinement for TMSCs, which is based on DeNicola’s and Hennessy’s must preorder. Case studies are used to illustrate the utility of the work.

1. Introduction

Triggered Message Sequence Charts (TMSCs) have been proposed in [20] as a scenario-based visual formalism for distributed systems. TMSCs allow users to integrate prescriptive (“do this”) and constraint-based (“don’t do that”) system requirements within a common framework, and supports a mathematically precise notion of when one requirement specification *refines* another. Based on the well-known Message Sequence Charts (MSCs) notation [3], TMSCs enrich the expressiveness of scenario-based notations through syntactically simple but semantically powerful extensions to MSCs.

MSCs are widely used in capturing system require-

ments in the initial stages of system development, when many design-related issues typically remain unresolved and not all eventual implementation scenarios may be known. However, semantically, MSC specifications as typified in [3, 14, 18], are interpreted as *deterministic* and *complete*, and require implementations to exhibit exactly the execution sequences the MSCs exhibit. As a consequence, MSC-based early-stage specifications can only capture *exact behavior* (“do this, and only this”), and substantial burdens are imposed on the requirements elicitation process, since complete behavior must also be reflected in the set of requirements.

TMSCs represent an attempt to bridge this gap between the semantic definition and the practical use of MSCs. At the heart of TMSCs lie *conditional scenarios*, which represent requirements that constrain system behavior only when certain “triggering behaviors”, are observed; and *partial scenarios*, which permit users to leave aspects of system behavior unspecified. The theory is also equipped with a refinement ordering that determines when one specification is a “correct elaboration of” another, by correctly adhering to prescriptive and conditional-scenario constraints and properly “filling in” unspecified behavior in partial scenarios.

The goal of this paper is to illustrate how the TMSC theory of [19], and the notions of conditional and partial scenario coupled with scenario refinement, provide practical support for two different styles of requirements modeling. In the first, conditional scenarios are used to eliminate undesirable behaviors in a simple base system specification. In the second, partial scenarios are used in support of an elaboration-based specification style in which successive refinements define system behavior in response to different execution conditions while leaving others undefined. We introduce each methodology via a case study and show how TMSCs support a separation of design concerns.

The rest of the paper is organized as follows: in Section 2

we present a brief overview of the syntax and semantics of TMSCs. The following section presents our first requirements elicitation method based on the use of conditional scenarios to constrain system behavior; this is accompanied by an appropriate case-study. The next section describes the use of partial scenarios in facilitating a step-wise development of specifications; a second case-study serves to illustrate this method. Section 5 contrasts the two methods and discusses TMSC tool support, while Section 6 presents conclusions and directions of future research.

2. Overview of TMSCs

We begin by presenting a brief overview of the syntax and semantics of TMSCs. For details, the interested reader is referred to [20], from which this section is adapted.

Visual syntax. Graphically, we represent TMSCs as in Fig. 1. There are two new features in the visual syntax of

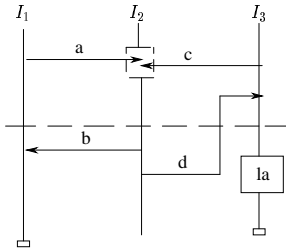


Figure 1. An Example TMSC

TMSCs when compared to traditional MSCs. The first is the horizontal line running through the instances which partitions the sequence of events on an instance’s axis into two: the first subsequence, located above the line, constitutes the instances *trigger*, and the second subsequence, below the line, constitutes its *action*. This partition, in effect, forms the basis of a *conditional scenario*: for each instance, the execution of the action is conditional on the occurrence of the trigger. In other words, the behavior of the instance is constrained to its action *only* when it has executed its trigger; otherwise, there are no restrictions. The second new feature in a TMSC is the presence/absence of a small bar at the foot of each instance. The presence of such a bar (as in instance I_1 in Fig 1) indicates that the instance cannot proceed beyond this point in the TMSC, while the absence (as in instance I_2) means that the behavior of this instance beyond the TMSC is left unspecified i.e. there are no constraints on its subsequent behavior. Such a scenario is thus *partial*, and may be extended in future.

The TMSC in Fig.1 may be read as follows:

If I_1 sends a to I_2 , then it should receive b from I_2 and terminate; if I_2 receives a from I_1 and c from I_3 in any order (the dashed box on I_3 ’s axis is a

co-region, which allows incident events to be ordered arbitrarily), then it should send b to I_1 and d to I_3 , and its subsequent behavior is left unspecified; if I_3 sends c to I_2 and receives d from I_2 , then it should perform the local-action la and terminate.

The trigger/action requirements are thus localized to each instance. Also, the trigger (action) of one instance may depend on the action (trigger) of another instance in a TMSC, which results in messages crossing the horizontal line as shown in Fig 1. Finally, it may be noted that an MSC M corresponds to a TMSC M' depicting the same scenario, where each instance in M' has an empty trigger, and each instance terminates in M' .

Abstract Syntax. Mathematically, a TMSC M is a tuple $\langle \mathcal{I}, \mathcal{M}, \mathcal{L}, trig, act, term \rangle$, where:

1. \mathcal{I}, \mathcal{M} and \mathcal{L} are finite sets of *instances, messages* and *local action names*, respectively.
2. *Trigger function* $trig$, and *Action function* act map an instance into the sequence of co-regions that make up the trigger and action respectively, of that instance in M
3. $term \subseteq \mathcal{I}$ is the *terminating set* and contains those instances which terminate upon performing their actions.

TMSC Expressions. Single TMSCs, as depicted above, serve as the basic building blocks for structured system specifications. An algebra of operators is used to generate larger specifications out of subspecifications. The resulting terms, which are referred to as *TMSC expressions*, have the following syntax:

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

The TMSC language offers a selection of “behavioral” and “logical” operators (as opposed to purely behavioral constructs typically used in MSC specifications) to facilitate a structured approach to *requirements management* whereby composite requirements are generated by interweaving prescriptive and constraint-based requirements. $\parallel, \mp, ;$ and $recX$ falls into the behavioral category, \wedge is a logical construct, while \oplus falls into both categories. The \parallel operator runs two TMSC expressions in parallel. $S_1 \mp S_2$ represents the “deterministic choice” between S_1 and S_2 while $S_1 \oplus S_2$

represents the nondeterministic choice: a successful refinement can choose either. In this respect \oplus has overtones of logical disjunction. $S_1; S_2$ denotes the *asynchronous* sequential composition [18] of S_1 and S_2 . The recursive operator, *rec* allows us to model infinite behavior of processes, where a new execution cycle starts whenever there is a reference within S , to the variable used in the recursive definition (say X). Finally, $S_1 \wedge S_2$ represents the logical conjunction of S_1 and S_2 , i.e. it specifies a system that needs to satisfy the requirements expressed by both S_1 and S_2 .

Semantics of TMSCs. The formal semantics of TMSCs is based on *acceptance trees* and the *must preorder* of [11]. The semantics is described in detail in [19], and it is beyond the scope of this paper to revisit it. Intuitively, however, an acceptance tree of a system is a function which maps sequences of events (that may arise during system execution) to *acceptance sets*. An acceptance set of a system P , after an execution sequence w , is a measure of *non-determinism*: for each *state* of the system that is reachable by w from the start state of P , $Acc(P, w)$, the acceptance set of P after w , contains the events that are enabled in that state, i.e. those events which may be performed from that state.

The *must preorder*, which arises in the theory of process testing given in [11], relates two systems on the basis of their relative non-determinism, and may be characterized in terms of acceptance sets. Given two systems P_1 and P_2 with acceptance trees $T[P_1]$ and $T[P_2]$ respectively, we say that $P_1 \sqsubseteq_{\text{must}} P_2$, if and only if for all execution sequences w , $T[P_1](w) \supseteq T[P_2](w)$.

The semantics of the TMSC language is based on interpreting individual TMSCs as acceptance trees: a TMSC is essentially treated as a non-deterministic choice of all behaviors violating the trigger, together with those in which the trigger is satisfied, and “progress” is made on performing the action. This treatment then serves as a basis for interpreting the TMSC expression operators and for translating TMSC expressions into acceptance trees. As a byproduct of this approach, we immediately obtain a refinement ordering based on the *must preorder* as outlined above: given two TMSC expressions S_1 and S_2 , we say S_2 is more refined than S_1 , written $S_1 \sqsubseteq_{\text{must}} S_2$, if S_2 in effect includes a subset of the nondeterministic behaviors of S_1 . One of the useful properties of our TMSC semantics is that it is compositional in the sense that we may refine a TMSC expression by refining its subexpressions.

Related Work: TMSCs combine the intuitive appeal of *visual languages* with the mathematical rigor of *formal methods*, approaches which have gained wide acceptance in requirements engineering. Scenario-based visual languages are often used for requirements elicitation, and may later lead to a specification given as a statechart-like model [22]. RSML [15] combines a state-based notation similar to statecharts with a tabular notation for transition definitions. An-

other formal tabular notation widely used is SCR [10]. Of direct relevance to our work, [13], [9], [21] have all proposed trigger-like precondition operators for scenarios in different contexts. The interested reader is referred to [6] for a comparison of these approaches with TMSCs.

3. Conditional Scenarios: Constraints Based Requirements Elicitation

Many distributed systems are functionally composed of *architectural components* i.e. groups of units whose behaviors are sufficiently independent to be modeled in isolation. The functionality of the distributed system may then be described by appropriate composition of these *sub-systems*. Most systems, however, also need to preserve some sort of global *safety* properties, which may imply that some of the execution traces produced by the unconstrained composition of the behaviors of the sub-systems are *undesirable*. Thus we need to impose constraints on the way the sub-systems interact to preserve the intended behavior of the overall system.

The specification requirements described above find a natural expression in the TMSC language. We first give a simple definition of the behavior of each architectural component in isolation, and then “glue” these specifications together, to obtain a *coarse* specification of the overall system-behavior. Note that although the scenarios describing the basic behavior of the components may be deterministic and may 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. Once the description of the basic system behavior is in place, we can “walk through” this specification to determine if there are undesirable execution traces that may be generated. If so, we use *conditional scenarios* to “weed-out” the incorrect execution sequences, and *refine* the initial coarse specification by adding these constraints using the \wedge operator.

There are a number of advantages to this approach. Firstly, by identifying the architectural components, we can focus on specifying the behavior of smaller systems to start with. Secondly, each such sub-specification, in effect, becomes a description of the *interface* of an unit with respect to the others in that sub-system. Thirdly, the combination of the behaviors of the sub-systems leads to a clearer understanding of how the behavior of the overall system is derived. Finally, the conditional scenarios, added in later, explicitly represent the behavior required of the system to the different *triggering* conditions that may arise.

The above strategy thus offers greater flexibility in requirements elicitation compared to deterministic MSC-based approaches, although it would still scale comparably

with the latter. The following case-study will illustrate the points made above.

Automated Resuscitation and Stabilization System

We present a generic description of the behavior of an Automated Resuscitation and Stabilization System (ARSS), which is integrated with many medical devices nowadays (e.g. [1]) to automatically track a patient's blood pressure and add fluids as necessary to stabilize the patient's condition.

Physical Units. The system consists of a blood pressure measuring device (B), an infusion pump (P), a display and alarm unit (D/A) and a software component (R) that controls the resuscitation process. It is assumed that the care-giver has provided R with a *target* blood pressure for the patient; R is to periodically poll the current pressure of the patient, and maintain a suitable flow-rate through the infusion pump until the patient's blood pressure reaches the target value, and his/her condition stabilizes. Each of B , P and D/A interacts only with R , and not with each other. The interaction of R with any one of these units may thus be modeled in isolation of the rest of the system.

R operates in cycles and communicates with each unit during a cycle. We distinguish between two kinds of cycles: *terminating* and *non-terminating*. A terminating cycle is one where R determines that the patient's blood-pressure has reached the target value, so the fluid-infusion may stop. All other behaviors constitute a non-terminating cycle, and R has to set the fluid-rate to the appropriate value during such cycles.

We first model the basic interaction between R and the other physical units during a cycle of operation (Fig. 2).

R and B. The possible interactions between R and B are depicted in TMSCs M_1 , M_2 and M_3 . R sends a *query* message to B and B responds by either sending the current blood-pressure $p-val$ to R (as in M_1 and M_3), or sending an error message $p-err$ 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 the correct pressure value is obtained, then R compares this value with the target value and performs the local-action ls (M_1) if the current pressure is less than the target, or the local-action eq (M_3) if the two pressure values are the same. M_3 thus depicts the interaction between R and B during a terminating cycle; the possible interactions during a non-terminating cycle is given by the TMS 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 $p-val$ or $p-err$.

R and P. R may send three types of messages to P :(i) *flow-rate* (M_4) is the flow-rate at which the pump should supply fluids to the patient; R computes *flow-rate* by the local-action *comp* (ii) *default* (M_5) is a pre-determined *safe* flow-rate at

which the pump may operate in case R is unable to compute the correct flow-rate and (iii) *stop* (M_6), which instructs P to stop the infusion. In each case, P responds to these messages by performing appropriate local-actions to adjust the flow. M_6 arises only during the terminating cycle. The specification for a non-terminating cycle is given by $RP = M_4 \oplus M_5$; 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 .

R and D/A. R may interact with D/A in the following ways:(i) under normal operating conditions, R sends the current pressure value $p-val$ and flow-rate $f-rate$ for display to D/A . This scenario is depicted in M_7 (ii) if the immediate attention of the care-giver is required, then R instructs D/A to sound an alarm of type 1 (M_8) and (iii) if the blood-pressure of the patient has reached the desired value, then R asks D/A to display the current pressure, and inform the care-giver by an alarm of type 2, so that the care-giver may decide on the next course of action. M_9 represents this terminating behavior. The interaction between R and D/A for other cycles is expressed by $RDA = M_7 \oplus M_8$. As before, \oplus is used as the choice will be made internally by R , depending on its interaction with the rest of the system.

Base Specification. We will now glue together the behaviors described above to obtain an initial base specification of the overall system. R begins each cycle by querying B about the current blood-pressure of the patient. It subsequently sends messages to P and D/A (the exact order of the messages may remain unspecified in this early-stage system description) to control the flow-rate of the pump and to display appropriate messages or warn the care-giver. For a non-terminating cycle, an initial specification is thus given by

$$\begin{aligned} NT &= RB; (RP \parallel RDA) \\ &= (M_1 \mp M_2); ((M_4 \oplus M_5) \parallel (M_7 \oplus M_8)) \end{aligned}$$

If the blood-pressure reaches the target value (M_3), then R asks P to stop the flow (M_6) and also sets off an alarm in the D/A unit to inform the care-giver (M_9). Thus the terminating behavior may be described by

$$T = M_3; M_6; M_9$$

The choice between terminating and non-terminating behavior is delayed until R determines whether the current pressure and the desired pressure values are equal or not. Hence the overall base specification is:

$$\begin{aligned} BS &= NT \mp T \\ &= ((M_1 \mp M_2); ((M_4 \oplus M_5) \parallel (M_7 \oplus M_8))) \\ &\quad \mp (M_3; M_6; M_9) \end{aligned}$$

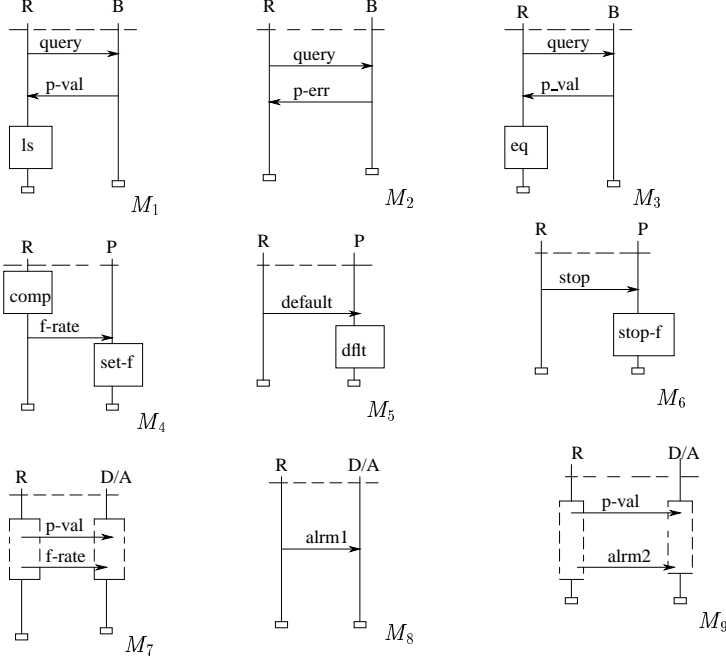


Figure 2. Basic interaction between R, B, P and D/A

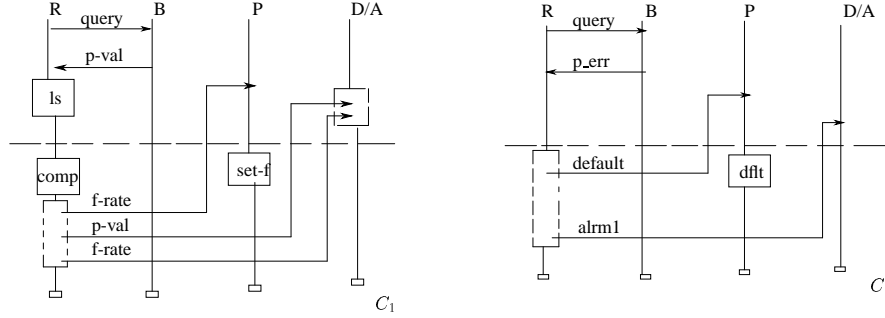


Figure 3. TMSCs representing constraints on ARSS

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 a global execution sequence), the specification becomes too permissive; for example, it allows R to sound the type 1 alarm even when B has correctly read the pressure. This is because the base specification is obtained by gluing together several local views, and hence is relatively *coarse* in its ability to describe global system requirements.

This is where TMSC based conditional scenarios are useful: we *weed out* the undesirable execution sequences by appropriately constraining system behavior under different *triggering* conditions. C_1 and C_2 (Fig.3) are two requirements we place on the system under consideration. C_1 constrains the system behavior when the blood pressure has

been read correctly, and ensures that the correct flow-rate is computed and displayed when this happens. Similarly, C_2 ensures that if B has lost the pressure source, then the D/A unit sounds the alarm to warn the care-giver, while P maintains a safe fluid-infusion rate till help arrives. We then *refine* our base specification by adding in these constraints to get a refined specification RS :

$$RS = (((M_1 \mp M_2); ((M_4 \oplus M_5) \parallel (M_7 \oplus M_8))) \wedge C_1 \wedge C_2) \mp (M_3; M_6; M_9)$$

It may be shown that $BS \sqsubseteq_{\text{must}} RS$.

Overall Specification. The overall *ARSS* specification may now be obtained by considering its behavior over multiple cycles; the system continues to supply fluid to the patient at an appropriate rate till his blood-pressure reaches the desired value, when the execution terminates. Thus the

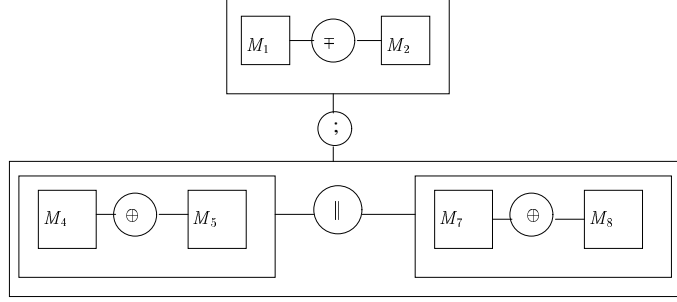


Figure 4. A graphical notation for TMS Expression NT

final specification is given by:

$$FS = recX.((((M_1 \mp M_2); ((M_4 \oplus M_5) \parallel (M_7 \oplus M_8))) \wedge C_1 \wedge C_2); X) \mp (M_3; M_6; M_9)$$

Finally, TMS expressions, represented by process algebraic terms, may become visually complex. In order to make the syntax more appealing, practitioners may use a simple visual syntax, where TMS subexpressions may be represented by boxes, and circular nodes labeled by TMS operators may be used to connect them to construct TMS expressions. Recursion would then correspond to loops in such a graph. A visual notation for the TMS expression $NT = (M_1 \mp M_2); ((M_4 \oplus M_5) \parallel (M_7 \oplus M_8))$, is shown in Fig. 4.

4. Partial Scenarios: Elaboration-Based Requirements Specification

MSC based specifications, as mentioned in Section 1, are expected to be *deterministic* and *complete*. This implies that all aspects of the system behavior have to be clearly, and more importantly, completely, specified to start with. However, it is often virtually impossible during the early stages of system development to be able to describe not only the normative behavior of the system, but also the different error conditions that may arise, and how to handle each. As a result, this leads to an unsystematic way of designing systems: we start with an initial specification (which, semantically, is supposed to be complete), discover new scenarios that may arise, and make ad hoc changes to the initial specification to get a new specification which may not have any coherent relationship with the former. Indeed, the new behaviors may conflict in subtle and undesirable ways with existing specified behavior; such *feature interactions* are notoriously difficult to uncover and correct in traditional system specification notations. Moreover, it may very well turn out that even this new specification does not depict all possible scenarios, and so is again “tampered” with. Such unplanned development is an unfortunate outcome of attempting to specify everything at once; it not only carries the risk

of introducing errors into the specification, but gives rise to specifications that are difficult to maintain.

TMSs address the above problem by allowing specifications to be *partial*. This is achieved through the use of partial scenarios, which describe an *incomplete* interaction sequence between instances. Semantically, the behavior of the system beyond that depicted in the scenario, is left unspecified, i.e. there are no constraints on its subsequent behavior. Such an approach has many advantages. For example, we may specify systems *incrementally*, by describing only a part of its behavior to start with, and then adding more and more detail in subsequent steps. Each new specification is then an elaboration of the previous one, and represents a *refinement* of the system behavior. Thus, in the first step, we may describe the normative operation of the system, with simple “stubs” for the abnormal conditions that can arise; then, in subsequent refinements, we may “fill in” the stubs, by defining what the system should do to handle the corresponding error condition. Moreover, in case we decide to change the recovery action in response to an error, we are, in effect, only considering a different refinement to the original specification, rather than proposing an entirely new specification, with no relationship to the former. In other words, we are only *backtracking* a few steps in the refinement sequence and then progressing in a different direction. Such a stepwise development greatly reduces the possibilities of errors seeping into the specification. The use of stubs makes it clear where a certain change has to be made if the need arises, and leads to specifications that are easy to maintain.

[5] describes a framework based on the SCR tabular notation that also separates erroneous and normal behavior during requirements specification. The main motivation behind such approaches is to facilitate a separation of design concerns; as such, our method also has connections with *aspect-oriented* software development [12], which provides useful abstractions for representing “cross-cutting” design idioms in a scalable fashion. When system behavior can be thought of in terms of “cross-cutting” layers of concerns, the method outlined above would allow such systems to

scale: adding a new behavior can be achieved with minimal disturbance to an existing specification.

We will now illustrate these notions via a case study.

Steam-Boiler Control Specification

We present the specification of a steam boiler control [4] using TMSCs. The description that follows is along the lines of the informal text available at [2]; however, [2] is significantly biased to a particular implementation, while our description of the boiler operation is more general. For brevity, we have also made certain simplifying assumptions.

Physical Units. The system consists of the steam boiler, a pump (with controller P) to provide the boiler with water, a device (with controller W) to measure the quantity of water in the steam-boiler, another device (with controller S) to measure the quantity of steam which comes out of the steam-boiler, and a valve for the evacuation of water in the initial phase. The controller(C) for the boiler, communicates with P , W and S .

The boiler has a total capacity of Q units. There is a *minimal quantity*, say, Q_1 units, of water in the boiler, below which the boiler would be in danger if steam continued to come out at its maximum quantity without supply of water from the pump. Similarly, there is a *maximal quantity*, say Q_2 units, of water in the pump, above which the boiler would be in danger if the pump continued to supply water without the possibility to evacuate the steam. There are also a *minimal normal quantity* N_1 , and a *maximal normal quantity* N_2 , of water to be maintained in the boiler during regular operation. Obviously, $Q_1 < N_1$ and $N_2 < Q_2$.

In the initial phase, the quantity of water in the boiler is adjusted so that it is between N_1 and N_2 . This is done by using the pump to supply water to the boiler, or by using the valve to drain out water from the boiler, as appropriate. For brevity, we will not include this phase in our specification.

Once the initial phase is complete, the system starts operating in a cycle and a priori, does not terminate. During each cycle, the boiler control program performs the following actions in sequence a) receive messages from P , W and S b) analyze information that have been received c) transmit messages to P , W and S .

Normal Mode. In the normal mode, the water-level in the boiler is between N_1 and N_2 , and all the physical units are operating correctly. As soon as the water level falls below N_1 , or rises above N_2 , the level is adjusted by the program by switching the pump *on* or *off*. The appropriate decision is taken on the basis of the information received from W .

The normal mode of operation is represented by the TMSC expression

$$S_{nml} = T_1; (T_2 \mp T_3 \mp T_4)$$

where the TMSCs T_1 , T_2 , T_3 and T_4 are shown in Fig. 5. In T_1 , C performs the local action *nml* when the messages p , w and s indicate that all the corresponding units are operating normally, and the water-level is within the safe limits. C then determines the actual water level from w . If this water level is between the normal limits of N_1 and N_2 , indicated by the local-action *bt12*, then C sends out the message *ok* to all the physical units. If the water level is below N_1 (corresponding to the local action *ls1*), then C sends a message *pump-on* to P to switch on the pump, which P does by performing the local-action *on*; else, if the level is above N_2 (corresponding to the local-action *gr2*), then C asks P to switch off the pump, and P does so by the local-action *off*.

Error Modes. We will consider the following conditions to be erroneous and requiring appropriate recovery actions:

- The message p from the pump-controller P indicates that the pump is malfunctioning
- The message w from the controller W indicates that either the water-level is approaching the unsafe limits Q_1 or Q_2 , or that the device is malfunctioning.

For simplicity, we will assume that the steam measuring unit is always working normally. The specification for the modes will be developed in three steps, each step adding more detail to the previous one.

First Stage. Our first attempt at a specification of the erroneous behavior is given by the TMSC expression:

$$S1_{err} = T_5; (T_6 \mp T_7 \mp T_8),$$

where T_5 , T_6 , T_7 and T_8 are as shown in Fig. 6.

In the error mode, C firsts receives the messages p , w and s , and performs the local-action *error* to indicate that an undesirable condition has occurred (TMSC T_5). This can either be a malfunctioning of the pump supplying the boiler with water (and corresponds to the local-action P -*flt* on C 's axis in T_6), or it may be a trouble reported by W , indicating that the the water-level is reaching the maximal-limits Q_1 or Q_2 , or that there is a problem with the water-level measuring device. *Lev-flt* (T_7) and *W-flt* (T_8) are the local-actions performed by C , in these two cases.

In case the pump is malfunctioning, C enters an *emergency stop* mode (by performing the local-action *stop*): it sends a *stop* message to the physical units, and itself terminates. This scenario is depicted in T_6 . At this stage, this is the only erroneous behavior that we model; we indicate the possibility of the other faults (in T_7 and T_8), but the subsequent behavior in those cases is left unspecified. T_7 and T_8 thus depict *partial scenarios*, that will be elaborated in subsequent stages.

Second Stage. We will now specify the behavior of the boiler control, if the maximal limits are approached. We suggest two possibilities: in the first, when an analysis of

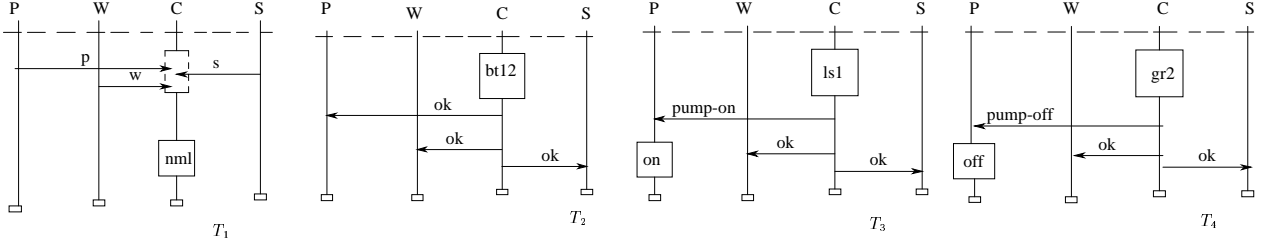


Figure 5. TMSCs representing the normal operation of the boiler

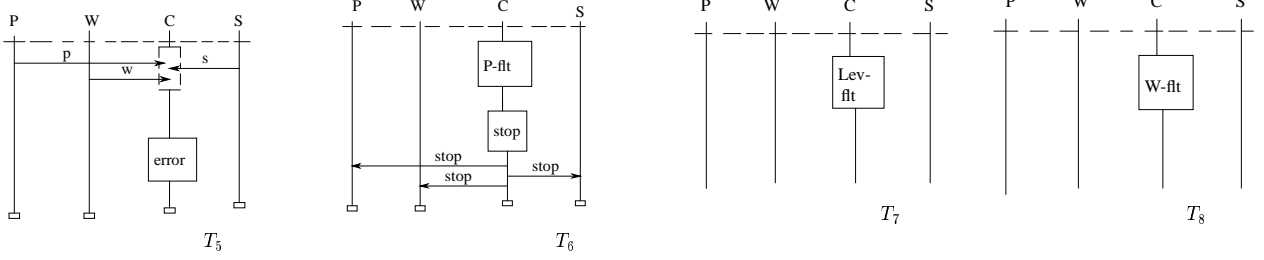


Figure 6. Error condition: malfunctioning of the pipe

w by C indicates that either Q_1 or Q_2 is approaching (indicated by the local-actions $appr1$ and $appr2$, in T_9 and T_{10} respectively), the system goes into an emergency-stop mode; in the second, we still go into the emergency mode in case the lower limit Q_1 is reached, but if the upper limit Q_2 is being approached, we may make use of the valve (used primarily in the initial stages) to quickly drain out an appropriate amount of water (through the local-action $openlv$), and thus ensure continuity of operation. Which alternative to take will depend on a more detailed analysis of the system: how long the boiler can stay in this erroneous mode before the condition becomes dangerous, how fast the valve can drain out the water, what is the maximum time the boiler-controller C may take to finish one cycle of its operation etc. Design decisions will be based on these considerations, and an appropriate choice may be made at a later stage.

The possible controller behavior in case the maximal limits are reached is then given by:

$$S_{LF} = T_9 \mp (T_{10} \oplus T_{11})$$

where T_9 , T_{10} and T_{11} are given in Fig. 7.

We have thus elaborated on the partial scenario depicted in T_7 , and it may be shown that $T_7 \sqsubseteq_{\text{must}} S_{LF}$

The refined specification obtained in the second stage is given by:

$$S_{2err} = T_5; (T_6 \mp S_{LF} \mp T_8)$$

As our semantics is compositional, it follows that $S1_{err} \sqsubseteq_{\text{must}} S_{2err}$.

Third Stage. We now turn our attention to the error condition that arises from the malfunctioning of the water-level indicator. Once again, we will propose two possible responses to this problem. A simple solution would be to let the boiler control go into the emergency-stop mode when this occurs. T_{12} in Fig. 8 depicts the corresponding scenario. The program terminates, and has to be restarted when the water measuring unit has been repaired. A more sophisticated and fault-tolerant solution would be to continue the operation by estimating the water-level by a computation which is done taking into account the maximum dynamics of the quantity of steam coming out of the boiler. This calculation may have to assume that n units of water, supplied by the pumps, account for exactly the same amount of boiler contents i.e. there is no thermal expansion. The calculation, to be performed by the boiler control, is indicated by the local-action $lev\text{-}calc$ in TMS T_{13} . The result may either indicate that the water-level is within safe limits, as in T_2 , T_3 or T_4 , or that it is approaching one of the maximal limit quantities, as in T_7 , and appropriate action needs to be taken as proposed in S_{LF} .

Thus, if the water-level indicator malfunctions, the proposed behavior is described by the TMS expression:

$$S_{WF} = T_{12} \oplus (T_{13}; (T_2 \mp T_3 \mp T_4 \mp S_{LF}))$$

This expression is an elaboration of T_8 , i.e. $T_8 \sqsubseteq_{\text{must}} S_{WF}$. The final specification of behavior in the erroneous mode is then:

$$S_{err} = T_5; (T_6 \mp S_{LF} \mp S_{WF})$$

Once again, our compositional semantics ensures that

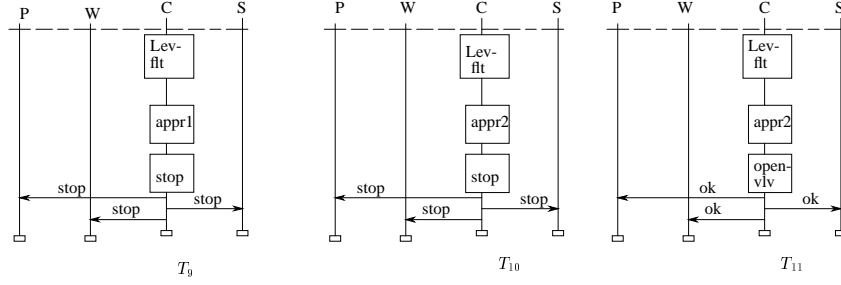


Figure 7. Erroneous condition: water level approaching maximal limits

$S2_{err} \sqsubseteq_{\text{must}} S_{err}$. Thus, $S1_{err} \sqsubseteq_{\text{must}} S2_{err} \sqsubseteq_{\text{must}} S_{err}$ forms a refinement sequence.

Overall Specification. We now put together the specifications of the normal mode and the erroneous mode to get the overall specification of the boiler control during one cycle of its operation. The choice between the two modes is delayed till the boiler control determines from the analysis of the messages, which mode it is going to operate in. Accordingly, the overall specification is given by:

$$S = S_{nml} \mp S_{err}$$

Finally, we may use the recursive operator *rec* to specify the behavior of the boiler control over an infinite number of cycles. This specification, S_{BC} is given by:

$$S_{BC} = \text{rec}X.(T_1; (T_2 \mp T_3 \mp T_4); X) \quad (1)$$

$$\mp T_5; (T_6 \mp \quad (2)$$

$$T_9 \mp (T_{10} \oplus (T_{11}; X)) \mp \quad (3)$$

$$(T_{12} \oplus (T_{13}; ((T_2 \mp T_3 \mp T_4); X) \quad (4)$$

$$\mp T_9 \mp (T_{10} \oplus (T_{11}; X)))))) \quad (5)$$

$$) \quad (6)$$

To describe the infinite behavior of the boiler control, during any cycle, we let the specification move on to the next cycle if the current one either involves normal operation or if the error condition is handled without stopping the operation; otherwise, the program terminates. Within the recursive operator, (1) describes the normal mode (S_{nml}), while (2) through (5) describe the erroneous mode of operation (S_{err}). Specifically, (2) handles the malfunctioning of the pipe controller, (3) handles the case when the water level reaches the dangerous limits, while (4) and (5) deal with the malfunctioning of the water-level indicator.

It may also be noted that the expression S_{BC} may be refined further, as each \oplus operator presents alternatives for eventual implementation behavior.

The final specification of the steam-boiler control is thus fairly complex, and it would have been difficult (and almost impossible for more complex systems) to express this complete behavior in a single attempt. However, partial sce-

narios, together with our refinement notion, supports a systematic development of such specifications, where we may focus on only certain aspects of system behavior at a time, and yet extend specifications as necessary. This flexibility is not afforded by traditional MSC specifications.

5. Discussion

Conditional vs Partial Scenarios. In the preceding two sections, we have looked at two complementary styles of requirements modeling using TMSCs. In the constraints-based approach, we model a distributed system bottom up from its largely independent sub-systems, and then *eliminate* undesirable execution sequences by imposing constraints in the form of conditional scenarios. The elaboration-style approach, on the other hand, is ideal for systems whose behavior is fairly complex, and hence is best specified by *extending* partial execution sequences in an incremental manner.

Large systems should ideally be built in a step-by-step rather than ad-hoc manner, and thus are ideal for an elaboration style of development; however, parts of the system may have a functionality that is best reflected by the architectural style, as in the ARSS, and conditional scenarios may be used to appropriately constrain the way the sub-systems interact.

Conditional scenarios and partial scenarios are thus not mutually exclusive ways of writing scenario-based specifications, but they tackle two different aspects in the development of such specifications: the former provides a way to constrain system executions, while the latter facilitates a step-wise development of complex system behavior.

Tool Support. In order to provide practical support to the methodologies discussed above, we have designed and implemented a tool named TRIM [7] that accepts TMSC expressions, constructs the corresponding behavioral models in the form of acceptance trees, and can also check if a given TMSC expression is a *refinement* of another. We have performed a number of interesting case-studies using TRIM, including the two presented in this paper. TRIM is based on the Concurrency Workbench (CWB) [8, 17] and

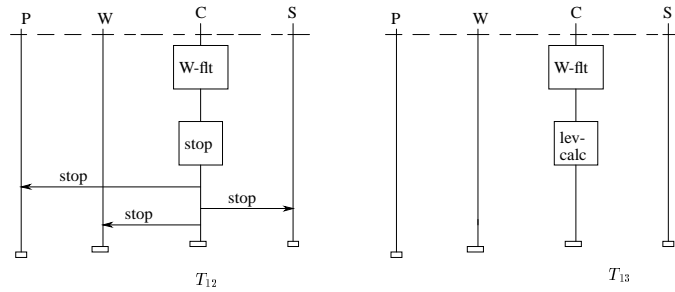


Figure 8. Erroneous Condition: malfunctioning of the water-level indicator

the Process Algebra Compiler (PAC) [16].

6. Conclusions

We have presented two refinement-based requirements modeling methods that utilize conditional and partial scenarios supported by the TMS language. These methods allow us to build specifications through successive refinements, either by imposing constraints on the way subsystems interact, or by extending partial behavior. Each method has been illustrated by a concrete case-study to highlight its practical usefulness.

For future work, we would like to investigate the generation of deterministic state-machine based descriptions of individual instances from scenario-based TMS specifications.

Acknowledgements: We would like to thank Dr. Constance Heitmeyer for her detailed comments on an initial version of the paper, and her subsequent help in improving it.

References

- [1] Integrated medical systems inc.-lstat. URL: <http://www.lstat.com/lstat.html>.
- [2] Steam-boiler control specification problem. URL: <http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html>.
- [3] Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
- [4] J. R. Abrial, E. Börger, and H. Langmaack. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. *LNCS volume 1165*, 1996.
- [5] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the scr requirements method. *Proceedings of 19th Digital Avionics Systems Conference*, 2000.
- [6] B. Sengupta and R. Cleaveland. Towards formal but flexible scenarios. *2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.
- [7] B. Sengupta and R. Cleaveland. TRIM: A tool for triggered message sequence charts. *Proceedings of 15th Computer Aided Verification Conference (CAV'03)*, 2003.
- [8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
- [9] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
- [10] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Mehtodology* 5, 3, pages 231–261, July, 1996.
- [11] M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
- [12] G. Kiczales and et. al. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.
- [13] I. Kruger. Distributed system design with message sequence charts. *PhD Thesis, Technical University of Munich*, 2000.
- [14] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.
- [15] N. Leveson and et. al. Requirements specifications for process control systems. *IEEE Transactions on Software Engineering*, pages 684–706, Sept. 1994.
- [16] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 1995*, LNCS volume 1019:153–173.
- [17] R. Cleaveland and S. Sims. The ncsu concurrency workbench. *CAV 1996*, LNCS volume 1102:394–397.
- [18] M. A. Reniers. Message sequence chart: Syntax and semantics. *PhD Thesis, Eindhoven University of Technology*, 1998.
- [19] B. Sengupta. Triggered message sequence charts. *Ph.D Thesis, SUNY Stony Brook*, 2003.
- [20] B. Sengupta and R. Cleaveland. Triggered message sequence charts. *ACM SIGSOFT 2002, 10th International Symposium on the Foundations of Software Engineering*.
- [21] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. *ACM SIGSOFT 2002, 10th International Symposium on the Foundations of Software Engineering*.
- [22] J. Whittle and J. Schumann. Generating statechart designs from scenarios. *International Conference on Software Engineering*, 2000.