TRIM: A Tool for Triggered Message Sequence Charts*

Bikram Sengupta Rance Cleaveland

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

Abstract. TRIM is a tool for analyzing system requirements expressed using *Triggered Message Sequence Charts* (TMSCs). TMSCs enhance MSCs with capabilities for expressing conditional and partial behavior and with a refinement ordering. This paper shows how the Concurrency Workbench of the New Century may be adapted to check refinements between TMSC specifications.

1 Introduction

Triggered Message Sequence Charts (TMSCs) [16] are a scenario-based visual formalism for capturing requirements of distributed systems. TMSCs enhance traditional MSCs [3] with capabilities for expressing *conditional* and *partial* behavior and with a mathematically precise notion of *refinement*, which may be used to check whether one set of requirements correctly elaborates on another. This paper presents TRIM, a tool for checking refinement between TMSCs. The main features of TRIM are: (i) a textual language for TMSCs that includes the algebraic combinators of [16]; (ii) a routine for checking refinements among TMSC specifications; and (iii) a capability for generating diagnostic information in the form of *tests* when one system fails to refine another.

TMSCs. Graphically, TMSCs, as exemplified in Fig. 1, extend MSCs in two respects. The first is the dashed horizontal line cutting across the *instances* (vertical axes) and partitioning the sequences of actions for each instance into a *trigger* — the subsequence above the line — and an *action* — the subsequence below. A TMSC requires that if an instance performs its trigger, then it must execute its action; otherwise, it is unconstrained. The second new feature in TMSCs is the possibility of a hollow bar at the foot of each instance, as in instances I_1 and I_3 in Fig. 1, and whose presence signals *termination*: no further behavior is allowed. A bar's absence (cf. instance I_2) means that there are no constraints on subsequent behavior, which may be extended in the future. TMSCs have an abstract textual syntax that is described formally in [16]; [7] compares and contrasts TMSCs with other MSC-based formalisms.

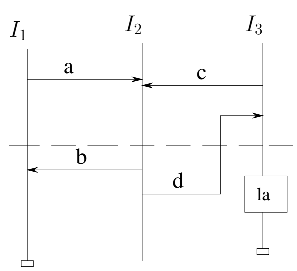

Fig. 1. An Example TMSC

* Research supported by NSF grants CCR-9988489 and CCR-0098037 and Army Research Office grants DAAD190110003 and DAAD190110019.

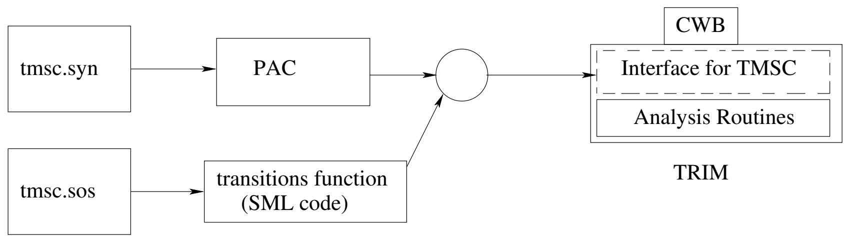

Fig. 2. Implementing TRIM

TMSC Expressions. Single TMSCs capture individual system requirements. The work in [16] also presents a set of operators for constructing structured collections of TMSCs. The resulting *TMSC expressions* have the following syntax.

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

The language includes programming-like constructs such as sequential and parallel composition, delayed choice, and recursion (to express iterative behavior), as well as more declarative operators such as internal choice and conjunction. The latter enable TMSCs to capture logical as well as operational requirements. The semantics of TMSC expressions is based on *acceptance tress* and the *must preorder* [12, 16].

2 TRIM

TRIM supports the textual notation for TMSCs given in [16] and provides: **a simulator** for executing TMSC expressions; **a compilation tool** for converting TMSC expressions into manually inspectable acceptance trees for manual inspection (impractical for large examples but often effective for smaller, early-stage artifacts); **routines for checking the refinement ordering** between TMSC expressions, and for returning diagnostic information when refinement fails to hold. The tool also includes routines for checking temporal properties of, and minimizing, finite-state TMSC expressions. The remainder of this section briefly describes how TRIM is implemented and used.

Implementing TRIM. TRIM is implemented on top of the Concurrency Workbench of the New Century (CWB-NC) [8, 9], an easy-to-retarget verification tool for finite-state systems. Instances of the CWB-NC consist of a front end that handles syntax and semantic issues of design notations, and a back-end that implements the analysis routines, including a simulator, a model checker, and several refinement-checking procedures. As the CWB-NC computes the must preorder, a natural approach for TRIM is to develop a TMSC front end for the CWB-NC. This is what we did; Fig. 2 gives an overview. The remainder of this section describes some of the issues involved in this program.

CWB-NC front ends must contain: a parser, an unparser, and a routine for computing the single-step transitions of system descriptions. The Process Algebra Compiler (PAC) [13] is designed to simplify the task of implementing CWB-NC front ends. The PAC generates front ends from design-language specifications describing the syntax of the language in the form a YACC-like grammar and the semantics of the language given as sets of Plotkin-style SOS rules [11]. The two specifications for a given language `L` are stored in two different files: `L.syn` for the syntax and `L.sos` for the semantics. Our initial goal was to use the PAC to generate the TMSC front end, and to this end we needed to devise two files: `tmsc.syn` and `tmsc.sos`.

Implementing `tmsc.syn` was straightforward. Devising SOS rules proved trickier. The semantics in [16] is essentially denotational; it defines the meaning of TMSC expression operators as constructions mapping acceptance trees to acceptance trees. It also allows for an arbitrary number of messages to be "pending", i.e. sent but not yet received. Both of these features pose problems for the CWB-NC, which requires an operational rather than a denotation semantics and also needs semantic entities to be *finite-state*. To cope with the former problem we gave SOS rules that are provably equivalent to the original declarative semantics; the main subtlety involved handling the interplay between nondeterminism and the conjunction operator. To help with the latter we equipped the operational semantics with a parameter that can be used to bound the number of messages in transit (i.e. the buffer size). This semantics thus approximates the "true" semantics, although they coincide for TMSC expressions whose number of pending messages never exceeds the parameter in the operational semantics.

We could not use the PAC to generate the semantic functions directly from our SOS rules, owing to PAC restrictions. For example, PAC does not allow the definition of mutually recursive auxiliary semantic relations; and yet our treatment of conjunction required this. Hence, we wrote the *transitions* function by hand from the `tmsc.sos` file, producing around 2,500 lines of SML code, and integrated it with the PAC-generated parser to build the TRIM interface for the TMSC language; see Fig. 2.

Using TRIM. TRIM is a research prototype; we did not pay close attention to performance issues in implementing the front end. Nevertheless, we have used TRIM to process several different TMSC-based specifications, including a simple protocol for atomic reading and writing [16]; the specification of an automated infusion pump used in treating trauma patients [2, 14]; the well-known steam-boiler example in [4, 14]; and a component of an air-traffic control system [1, 15]. The resulting transition systems ranged in size from 1,744 to 26,183 states, with the corresponding acceptance trees containing from 196 to 2,495 nodes. In all cases, refinements were proposed in terms of more deterministic TMSC expressions, and were verified using TRIM. The counter-example generation feature of CWB was a major advantage, especially because TRIM currently supports only a text-based interface and complex TMSC expressions typed in by a user are subject to typographical errors such as unintentional mistakes in message names. The analysis times ranged from several minutes to several hours, although performance improvements of two orders of magnitude are achievable, in our view.

Related Work. Several tools have been developed to support the use of scenarios in design requirements. MESA [6] allows certain properties, such as process divergence to be efficiently checked on MSCs. UBET ([5]) detects potential race conditions and timing violations in an MSC, and also provides automatic test case generation over HMSCs. The *play-in/play-out* approach of [10] is based on LSCs and has been implemented via a tool called the *play engine*. The tool LTSA-MSC [17] supports the synthesis of behavior models from MSC-based specifications and implied-scenario detection.

3 Conclusions and Future Work

We have presented TRIM, a tool that provides automated support for analyzing systems requirements given in the TMSC notation [16]. The tool provides a number of useful routines, including a simulator and a refinement checker, that are inherited from the CWB-NC verification tool on which it is based. Retargeting the CWB-NC to TMSCs required us to adapt the semantic account of the notation given in [16] to (i) make it operational and (ii) to bound pending messages. For future work, we plan to improve the performance of the TRIM front end and to develop a graphical user interface.

References

1. Center-TRACON automation system (CTAS). `URL:http://ctas.arc.nasa.gov/`.
2. Integrated Medical Systems Inc. `URL:http://www.lstat.com/lstat.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. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
6. H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. *Proc. TACAS'98*, LNCS volume 1384:118–135.
7. B.Sengupta and R.Cleaveland. Towards formal but flexible scenarios. *2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.
8. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, 1993.
9. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, January 2002.
10. D.Harel and R.Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and System Modeling (SoSym)*, 2003.
11. G.Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
12. M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
13. R.Cleaveland, E.Madelaine, and S.Sims. A front-end generator for verification tools. *Proc. TACAS'95*, LNCS volume 1019:153–173.
14. B. Sengupta and R. Cleaveland. Refinement-based requirements elicitation using triggered message sequence charts. To appear in 2003 Intl. Conf. on Requirements Engineering.
15. B. Sengupta and R. Cleaveland. Toward formal but flexible scenarios. To appear in the ICSE 2003 Workshop on Scenarios and State Machines.
16. B. Sengupta and R. Cleaveland. Triggered message sequence charts. *Proceedings of ACM SIGSOFT 2002, FSE-10*.
17. J.Kramer S.Uchitel and J.Magee. LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios. *Proc. TACAS'03*.