

# Programming with MSCs

Jia Liu and Don Batory  
Department of Computer Sciences  
The University of Texas  
Austin, Texas 78712  
{jliu,batory}@cs.utexas.edu

**Keywords:** message sequence charts, domain specific languages, state machines.

**Classification:** 3rd year Ph.D.

## 1 Introduction

*Message Sequence Charts (MSCs)* [1][2] are graphical representations that capture interactions between objects and show system execution traces. MSCs are widely used in software development processes to document system requirements. However, MSCs have well-known limitations. First, MSCs cannot be easily integrated into software life-cycles because they are difficult to incorporate with other requirement models and MSC specifications themselves do not help users to make a smooth transition from the requirement stage into the design stage [3]. Furthermore, because of the lack of semantics [4] and the expressability of detailed object internal actions, MSCs are inadequate to be used as guidelines for programming, or as programs themselves. Consequently, MSCs are often only used as complementary documents to accompany other specification models such as data flow charts.

The goal of our work is to enhance the potential of MSCs by extending MSC models for programming and unifying MSCs with *state machines (SMs)*. With our extended MSC model, programming is elevated to a higher level, which allows the developers to program directly with MSCs. This is done by extending MSC models so that they can be expressed as *domain specific languages (DSLs)* and building a compiler for the language. Our compiler translates MSC programs into state machine programs, and these programs into pure Java programs [5]. By fully automating the code generation process, we integrate MSCs seamlessly into the software development process. This integration smooths the transition from requirement analysis to design, and connects MSC models with design models. Since programming in an extended MSC model eliminates unnecessary design and implementation feedbacks, the burden of system maintenance is reduced. Additionally, our work facilitates the interactions between system developers and domain experts, which enhances the efficiency and accuracy of requirement collection and specification.

## 2 Programming with MSCs as a DSL

Though traditional MSC models are widely accepted, they are only used as a documentation tool due to its limitations. To more fully exploit the usage of MSCs and

enhance their flexibility and expressiveness, we extend traditional MSC models with new features to make it a DSL. Among the features that we added are:

**Local Actions.** Traditional MSC models focus on communications between objects in a system, which best describes external behaviors of the objects but leaves internal actions unspecified. However, in developing software systems the need for capturing local actions of objects is mandatory. In our model, actual Java code is integrated into MSCs through local action specifications. This added detail allows developers to write an MSC as a domain specific program.

**Guards.** In traditional MSC models there are no guards for message sending and receiving behaviors of each object, so MSC executions are non-deterministic when there are multiple choices. However in practice, deterministic executions are often desired. We added local guards for branch conditions and loop conditions to specify deterministic MSC executions.

With GenVoca technology [6] and JTS's tool support we have built our MSC language compiler in a layered manner. Fig. 1 illustrates the architecture of the MSC compiler, which consists of three layers: the Java language, state machine extensions to Java, and our MSC extension to Java.

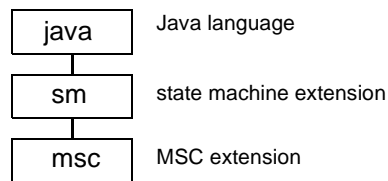


Fig. 1: Architecture of the MSC Compiler

Using this compiler, an MSC program can first be compiled into state machines, then these machines are translated into pure Java code.

### 3 FSATS Mortar Mission: An Example of Message Sequence Specification

FSATS is a fire simulation support software package which implements a military command-and-control scenario simulator [7]. FSATS consists of a number of objects (actors) that exchange messages according to a set of protocols under various kinds of missions. Due to the complexity of FSATS, we select from it one simple example, mortar mission, to illustrate our approach to MSC programming.

Fig. 2 describes the scenario of a mortar mission in conventional MSC representation. Four objects in this scenario are *Forward Observer (FO)*, *Fire Support Team (FIST)*, *Battalion (BN)* and *Mortar (MTR)*. When FO receives a *target event* message, it sends an *init* message to FIST. FIST can either deny the mission by sending a *deny* message to FO or forward the *init* message to BN. BN too can deny the mission or pass it along to MTR. Upon receiving the *init* message, MTR sends *accept*, *shot* and *roundComplete* messages all the way back to FO. Finally FO ends the mission by sending an *eom* (end of mission) message to MTR through FIST and BN.

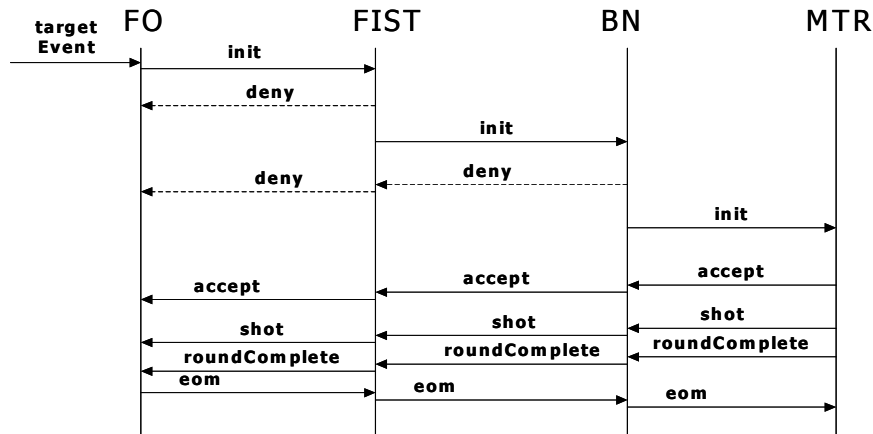


Fig. 2: Mortar Mission Scenario in MSC

The program shown in Fig. 3 is an implementation of the mortar mission in our MSC-based DSL. The program contains *exclusive* constructs in which only one branch will be executed. Each branch is guarded by a conditional written in the form *[object: pred-*

```

message targetEvent: Env -> FO; // a single message sent from
                                // the environment to FO
FO: { initiate(); } // <-- local action of FO
message init: FO -> FIST;
FIST: { initiate(); } // <-- local action of FIST

exclusive { // <-- branch point
  [FIST: !missionOK()]
    message deny: FIST -> FO; // if missionOK() test fails,
                              // FIST denies the mission

  [FIST: missionOK()] { // otherwise FIST proceeds
                        // the mission

    message init: FIST -> BN;
    BN: { initiate(); } // <-- local action of BN

    exclusive { // <-- branch point
      [BN: !missionOK()] { // if missionOK() test fails,
                          // BN denies the mission

        message deny: BN -> FIST;
        message deny: FIST -> FO;
      }
      [BN: missionOK()] { // otherwise BN proceeds
                          // the mission

        message init: BN -> MTR;
        MTR: { initiate(); } // <-- local action of MTR
        message accept: MTR -> BN -> FIST -> FO;
        MTR: { shootTarget(); } // <-- local action of MTR
        message shot: MTR -> BN -> FIST -> FO;
        message roundComplete: MTR -> BN -> FIST -> FO;
        message eom: FO -> FIST -> BN -> MTR;
      }
    }
  }
} // end of exclusive
} // end of exclusive

```

Fig. 3: Mortar Mission Code Written in DSL

icate] which allows the program to test some predicate on the designated object. Notice that local actions and branch predicates (both of which are coded as Java statements and expressions) are integrated into the program, which grants the developers finer control over the application without losing the benefits that MSCs provide. Fig. 4 depicts the state machine that was generated for each object in the MSC<sup>1</sup>.

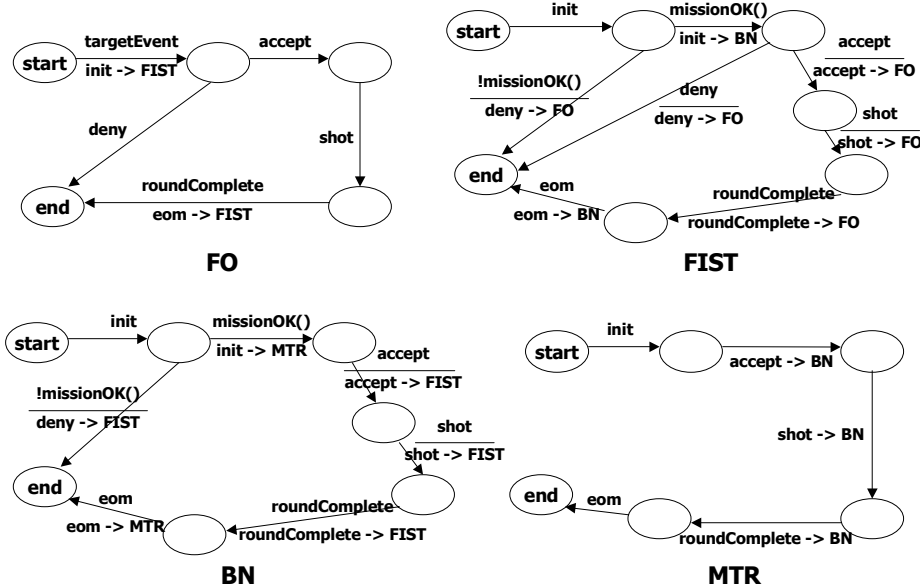


Fig. 4: Generated State Machines for Each Object

## 4 Conclusion and Future Work

MSCs are playing progressively more important roles in software development. They are extensively used in requirement analysis to display execution sequences of an application. MSCs are also used as documentation which give a graphical representation of object interactions and enhances user understandings of the system.

We believe that MSCs can play an even a larger role if MSCs are elevated to programming languages. Our extended MSC model improves the usability of traditional MSC specifications by giving explicit conditions for choices and details of the internal computations of a object. Extended MSC models are DSLs and MSC specifications can be treated as programs. We have built a compiler to automatically generate Java code

---

1. Here in Fig. 4 we use the same state machine notation as in [7]. Each transition (edge) has a guard and an action, both of which are optional. A guard can be some local predicate or a receiving message event, and only when the predicate is satisfied or the expected message is received from another object is the transition triggered. If an action is present, e.g. sending a message to another object, it will be executed during the transition.

from an MSC specification. This enables MSCs to be better integrated into the software development process and programming is elevated to a higher abstraction.

We foresee potential in this approach as well as a lot of work ahead. To better support modular design and enhance system extensibility, MSC inheritance will be investigated and MSCs will be fitted into layered structures of JTS applications based on MSC inheritance. Also more case studies are needed for our work. We will use other FSATS missions, network protocols and other scenario-based application for case studies. Finally, A MSC programming environment that supports writing, testing and visualizing MSCs as well as state machines will likely be needed. We expect such tools will facilitate software development with message sequence specifications.

## 5 References

- [1] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.
- [2] ITU-TS. Recommendation Z.120: Annex B. Geneva, 1998.
- [3] I. Kruger, “*Distributed System Design with Message Sequence Charts*”, Ph.D thesis, Technical University of Munich, 2000
- [4] S. Leue, L. Mehrmann, and M. Rezaei, “Synthesizing ROOM Models from Message Sequence Chart Specifications”, Technical Report 98-06, University of Waterloo, 1998.
- [5] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.
- [6] D. Batory and S. O'Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398, October 1992.
- [7] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, to appear *ACM Transactions on Software Engineering and Methodology*.