Representing Embedded System Sequence Diagrams As A Formal Language

Elizabeth Latronico and Philip Koopman
Carnegie Mellon University
Electrical and Computer Engineering Department
D-202 Hamerschlag Hall, 5000 Forbes Avenue
Pittsburgh, PA 15213
beth@cmu.edu, koopman@cmu.edu

Abstract. Sequence Diagrams (SDs) have proven useful for describing transaction-oriented systems, and can form a basis for creating statecharts. However, distributed embedded systems require special support for branching, state information, and composing SDs. Actors must traverse many SDs when using a complex embedded system. Current techniques are insufficiently rich to represent the behavior of real systems, such as elevators, without augmentation, and cannot identify the correct SD to execute next from any given state of the system. We propose the application of formal language theory to ensure that SDs (which can be thought of as specifying a grammar) have sufficient information to create statecharts (which implement the automata that recognize that grammar). A promising approach for SD to statechart synthesis then involves `compiling` SDs represented in an LL(1) grammar into statecharts, and permits us to bring the wealth of formal language and compiler theory to bear on this problem area.

1 Introduction

One of a designer's toughest challenges is attaining the appropriate level of abstraction. Include too little information, and a design is under-specified, often resulting in an incorrect or incomplete implementation. Include too much information, and the design is overly constrained or exceeds time-to-market allowances. Distributed, embedded systems present particularly onerous design challenges, combining complex behavior with a need for quick development cycles.

The Unified Modeling Language (UML) supplies an approach to express requirements and design decisions at various stages in the product development life cycle. The ideal level of abstraction provides the minimum sufficient amount of information required to create a set of correct, cohesive diagrams. UML offers two main diagrams for modeling system behavior: sequence diagrams and statecharts. While related, these diagrams often originate separately and serve diverse purposes. Sequence diagrams (SDs) are easier for people to generate and discuss, while statecharts provide a more powerful and thorough description of the behavior of a system. We present an algorithm to ensure that sequence diagrams contain sufficient information to be translated into statecharts. Specifically, this algorithm determines whether or not a set of sequence diagrams produces deterministic statecharts. This allows a designer to start with a skeletal structure and add information only when necessary. If automated, this technique could relieve a large burden of manual consistency-checking.

UML can be used to model a wide range of systems. To ensure that our results are applicable to actual embedded systems, the problem space needs to be carefully defined. Embedded systems may differ from the traditional transaction processing paradigm. Three major differences include:

- *Multiple initial conditions*
  Distributed, embedded systems typically run continuously and handle many user requests concurrently. Therefore, the system may not necessarily be in the same initial state for each user. Additionally, users may have disjoint objectives and responsibilities, so a second user may finish what a first user started.
- *Same user action evokes different system response*
  In transaction processing systems, there tends to be a one-to-one mapping from a user request to a system response. Embedded systems often have a limited user interface, so interface component functionality may depend on context.
- *Timing sensitivity*
  Embedded system functionality may depend on temporal properties such as duration, latency, and absolute time.

Sequence diagrams may contain information besides objects and messages. Three categories of additional information are presented, to show the scalability of the method and to provide examples of situations where supplemental information is required.

- *State*
  System history may affect present behavior. One example is a toggle power switch. The switch turns the system on or off, depending on current system state. Systems typically have a finite number of states.
- *Data*
  System behavior may depend on the current value of a variable with effectively infinite range. For example, in many elevators, selecting a floor will cause the doors to close, unless the user selects the current floor - then the doors will open.
- *Timing*
  Properties such as latency, duration, and absolute time may affect system response. One example is a car radio, where buttons are held to set the station.

In this paper, we explore how to use UML sequence diagrams to support the needs of embedded systems designers. Section 2 reviews methods for composing sequence diagrams that support flexible embedded systems modeling. Section 3 shows how determining required information content can be represented as a grammar parsing problem to guarantee correct, cohesive diagrams. A generic approach is described, with supporting embedded systems examples incorporating state, data, and timing information. Finally, the more commonly discussed transaction processing model is revisited to illustrate system differences. Section 4 summarizes conclusions.

2 Terminology and Related Work

2.1 Scenarios

2.1.1 Sequence Diagrams and Message Sequence Charts

A scenario describes a way to use a system to accomplish some function [5]. UML supports two main ways of expressing scenarios: collaboration diagrams and sequence diagrams. Sequence diagrams emphasize temporal ordering of events, whereas collaboration diagrams focus on the structure of interactions between objects [7]. Khriss et al. show how each may be readily translated into the other [7].

We concentrate on sequence diagrams because they elucidate temporal and object relation properties. As defined in the UML standard 1.3, a sequence diagram models temporal and object relationships using two dimensions, vertical for time and horizontal for objects [14]. Activity diagrams may also specify temporal properties, but typically do not include objects. As statecharts are usually defined per object, sequence diagrams are a more natural candidate for synthesis. The notation of sequence diagrams is based on, and is highly similar to, the Message Sequence Chart standard [6].

2.1.2 Composition of Scenarios

A crucial challenge in describing distributed embedded systems is the composition of scenarios. In order to be adequately expressive, sequence diagrams must reflect the structures of the programs they represent. In this paper, we survey approaches to modeling execution structures and transfer of control, and select a method that lends itself to embedded systems.

Our first objective is to refine a model that utilizes sequential, conditional, iterative, and concurrent execution. As many ideas exist, our task is to determine which are appropriate for embedded systems. Hsia et al. [5] discuss a process for scenario analysis that includes conditional branching. Somé et al. [12] present three ways of composing scenarios: sequential, alternative, and parallel. Glinz [2] includes iteration as well. Koskimies et al. [8] and Systä [13] present a tool that handles "algorithmic scenario diagrams" - sequence diagrams with sequential, iterative, conditional and concurrent behavior. We use elements of each, for a combined model that allows sequential, conditional, iterative, and concurrent behavior.

Our second objective is to model transfer of control through sequence diagram composition. The main concern is where to annotate control information. One approach is to include composition information in individual diagrams. Hitz and Kappel [4] examine sequence diagram generation from use cases, and discuss the probe concept - the insertion of a small scenario into a larger one at a specified juncture. Koskimies et al. [8] present a similar method using sub-scenarios. A second approach is to use a separate hierarchical diagram, instead of embedding control information in the constituent diagrams. The Message Sequence Chart (MSC) standard specifies a separate diagram to organize sub-diagrams [6]. Leue et al. [9] explore the usage of base MSCs and high-level MSCs. The high-level MSC graph describes how to compose base MSC graphs to obtain sequential, conditional, iterative, and concurrent execution. Li and Lilius [10] present an additional example of a high-level MSC graph, and apply this method to UML sequence diagrams to assess timing inconsistency. We use the hierarchical diagram approach.

2.1.3 Finite State Machines and Statecharts

Finite state automata describe the possible *states* of a system and *transitions* between these states. Unfortunately, properties of complex systems such as concurrent execution of components lead to extremely large state machines that challenge human comprehension. Statecharts were proposed by Harel [3] to control state explosion problems with finite state machines by introducing the concepts of hierarchy and orthogonal execution, and are the basis for UML statecharts [14].

2.1.4 Statechart Synthesis

The second challenge in describing distributed embedded systems is ensuring there is sufficient information for correct, cohesive diagrams. Sequence diagrams are often constructed first in the design life cycle; therefore, we addresses synthesis of statecharts from sequence diagrams.

Existing work has two shortcomings. First, sufficiency of information for generating statecharts is not checked. Additional information is either absent or applied globally. Our goal is to provide an approach by which a designer can include a minimum amount of information, thereby reducing design time and guaranteeing a correct set of statecharts. We present a methodology to verify sufficiency, by applying well-established parsing theory.

Second, systems with all three embedded system qualities of multiple initial conditions, mapping identical user actions to different system responses, and timing dependencies have not been scrutinized. Systems that lack one of these three qualities generally do not require additional information to produce correct statecharts; therefore, the sufficiency question seems to not have arisen.

We present three embedded systems and show, by applying grammar parsing techniques, that these embedded systems do require additional information to produce correct statecharts. Additionally, we examine a transaction processing system, to illustrate that additional information is not required.

Prior work contains a number of suggestions as to what information sequence diagrams should include to enable statechart synthesis. Information is used for various purposes, but deterministic translation to statecharts has not been emphasized, and information is globally annotated. Hsia et al. [5] give a regular grammar for scenarios in order to construct a deterministic finite state machine. This grammar is similar to ours, but information sufficiency is assumed, not proven. Other work has proposed additional information, comprising three categories: state, data, and timing information. Douglass [1] advocates incorporating state symbols to represent object state. Somé et al. [12] use data pre-conditions and post-conditions to define possible scenario execution ordering. Whittle and Schumann [15] discuss implications of repeated user actions as a motivation for incorporating data pre-conditions and post-conditions. Koskimies et al. [8] annotate sequence diagrams with assertions on data variables. Timing intervals between messages are included by Li and Lilius [10]. Our examples examine these three categories, exposing situations where additional information is needed for statechart synthesis, and situations where it is not.

A number of different systems have been explored and documented; however, these systems lack the combination of multiple initial conditions, same user action evoking different system responses, and timing criteria. Systems without one of these characteristics often fail to manifest sufficiency issues.

A library checkout system is explored by Glinz [2] and by Khriss et al. [7]. In [2], scenarios have differing initial conditions, and system response depends on data attributes. However, statecharts are constructed directly from an informal textual description, not sequence diagrams. [7] synthesizes statecharts from UML collaboration diagrams, but these diagrams have identical initial conditions, one-to-one response mapping, and no timing criteria.

The Automated Teller Machine (ATM) system is a common example, discussed by Somé et al. [12], Whittle and Schumann [15], and Koskimies et al. [8]. [12] permits timeouts and global timed transitions, but all scenarios share a single initial condition, and user actions are mapped one-to-one with system responses (aside from time-influenced transitions). Scenarios in [15] can have a one-to-many action-response mapping, but have identical initial conditions and no timing restrictions. [8] approaches the problem iteratively, generating partial statecharts from sequence diagrams and vice versa. Different subscenarios may handle the same user request; however, there is a single initial condition and timing information is not discussed. The methodology in [8] is extended by Systä [13] for a File Dialog application with the same properties.

We examine three embedded systems to provoke sufficiency questions, then apply our methodology to a traditional transaction processing system to show that these systems do not require additional information for sufficiency.

2.2 Sequence Diagram Composition

The hierarchical graph approach used by the Message Sequence Chart community [6, 9, 10] explicitly represents composition information not shown in standard UML sequence diagrams. Figure 1 shows a set of sequence diagrams for a television power switch. TV_1 and TV_2 are regular sequence diagrams. The system has two objects - the user and the TV. The user can send one message, `power`. The TV can send two messages, `turn_on` and `turn_off`. TV_{main} expresses the relationships between TV_1 and TV_2. The triangle indicates a possible initial condition - the system may start out in

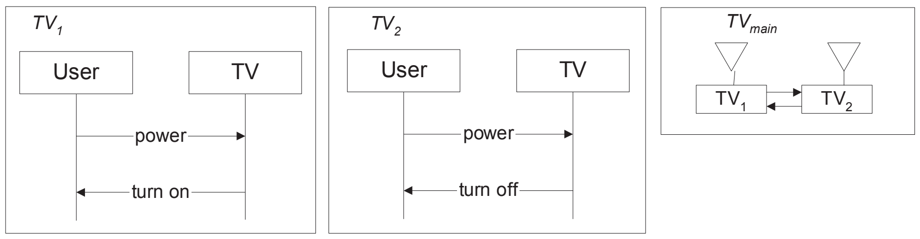

Figure 1 : Sequence diagrams for a television power switch

TV_1 or TV_2 . Arrows indicate legal compositions. TV_1 and TV_2 must alternate - the sequence $TV_1\,TV_1$ is not allowed. Without TV_{main}, composition information is absent.

Embedded system statechart synthesis typically requires more information than solely the messages an object receives. Three cases will be examined where sequence diagrams can be extended using state, data, and timing information to generate a deterministic grammar. Customary representations include state symbols, pre-conditions and post-conditions, and timing marks. Finally, the widely used ATM example will be reviewed to show that the ATM sequence diagrams generate a deterministic grammar without additional information regarding state, data, or timing.

3 Diagram Content

3.1 Deterministic Grammar

The main challenge in statechart synthesis is generating correct statecharts from a set of sequence diagrams with minimum sufficient information. The statecharts do not necessarily need to be complete, but they should give an unambiguous representation of the system. Rather than attempt an exhaustive annotation, a more achievable goal is to include the minimum sufficient amount of information.

Correct statechart synthesis from sequence diagrams with minimal annotation can be posed as a context-free grammar parsing problem. A similar approach was used by Hsia et al. [5] for text-based scenarios. To identify information gaps, we locate sequence diagram messages that translate into non-deterministic transitions in statecharts, as non-deterministic transitions often indicate information deficiencies. Standard methods for removing non-determinism, such as left factoring [11], and for implementing non-determinism, such as backtracking [11], cannot always be applied to embedded system sequence charts because messages may have global side effects on the external environment. Therefore, the only guaranteed correct approach is to ensure that sequence diagrams form an LL(1) grammar without left factoring or backtracking.

The context-free grammar for a sequence diagram may be defined as a set of message-response pairs. Given a message or set of messages, an object must produce a unique response or set of responses. An SD can be defined as a series of message-response events:

$$\text{SD} \rightarrow \text{message } \textbf{response} \text{ SD} \mid \varepsilon \tag{1}$$

where ε indicates the absence of a message or response. The goal is to construct an SD with a context-free grammar of the form

$$\text{message } \textbf{response} \rightarrow \alpha \textbf{ ResponseA} \mid \beta \textbf{ ResponseB} \mid \chi \textbf{ ResponseC} \ldots \tag{2}$$

where α, β and χ are distinct sequences of messages. A grammar of the form

$$\text{message } \textbf{response} \rightarrow \alpha \textbf{ ResponseA} \mid \alpha \textbf{ ResponseB} \tag{3}$$

does not produce a deterministic state machine. Upon receipt of α, the object does not know whether to execute `ResponseA` or `ResponseB`. The sequence diagram set for this grammar is shown in Figure 2. The system may start in either Seq_1 or Seq_2, and execute any combination of Seq_1 and Seq_2.

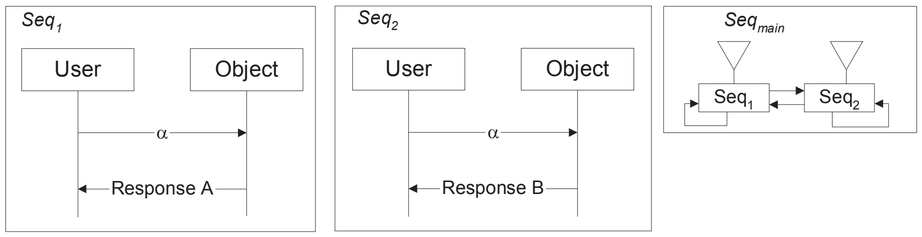

Figure 2 : Sequence diagrams for a generic non-deterministic grammar

Left factoring transforms the grammar in (3) to

$$\begin{aligned} &\text{message } \mathbf{response} \rightarrow \alpha\ A' \\ &A' \rightarrow \mathbf{ResponseA} \mid \mathbf{ResponseB} \end{aligned} \tag{4}$$

This is equivalent to the sequence diagram set given in Figure 3. The sequence diagram $\mathrm{Seq}_{\mathrm{factor}}$ is executed, followed by either Seq_1 or Seq_2. However, this only changes the composition of the diagrams. The problem of whether to execute `ResponseA` or `ResponseB` after the receipt of α remains.

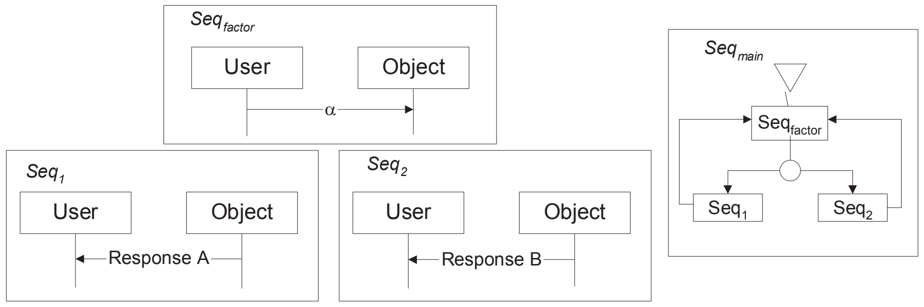

Figure 3 : Left-factored sequence diagrams for a generic non-deterministic grammar

The backtracking method picks a random response to be executed, and backtracks if the incorrect response was selected. Say the grammar is

$$\text{message } \mathbf{response} \rightarrow \alpha\ \mathbf{ResponseA}\ \delta \mid \alpha\ \mathbf{ResponseB}\ \phi \tag{5}$$

Upon receipt of α, it is unclear whether `ResponseA` or `ResponseB` is the correct behavior. Suppose `ResponseA` is randomly selected. The next message is ϕ. `ResponseA` was clearly incorrect, so the system backtracks and chooses `ResponseB` instead. However, in many real-time embedded systems, responses cannot be undone. For example, a microwave oven cannot undo burning popcorn, and an airbag cannot undo triggering its pyrotechnic charge. A greater difficulty emerges in scenarios where it is impossible to select a correct response based on messages alone. For instance, if α is the only message the user can generate, no amplifying information can be acquired; thus, the correct choice will never be known without querying existing system state.

3.2 State Information

A system may require state information to generate a deterministic set of statecharts. State symbols, advocated by Douglass [1], provide a succinct annotation. (Pre-conditions and post-conditions may alternatively be used, and are discussed in the next section). Figure 1 shows the sequence diagram set for a television with a power button. The television either turns on or off in response to the power message. Two initial conditions are possible – the television may be on or off when the user enters the room.

The grammar for the television is

$$\begin{aligned}&\text{SD} \rightarrow \text{message } \textbf{response} \text{ SD} \mid \varepsilon \\ &\text{message } \textbf{response} \rightarrow \text{power } \textbf{turn\_on} \mid \text{power } \textbf{turn\_off}\end{aligned} \tag{6}$$

This is of the form

$$\text{message } \textbf{response} \rightarrow \alpha\ \textbf{turn\_on} \mid \alpha\ \textbf{turn\_off} \tag{7}$$

and therefore non-deterministic, per the discussion in section 3.1

Adding state information can solve this non-determinism. The problem is that the state of the television is not represented in either sequence diagram, so the response to the power message is ambiguous. The television can be in two states, *on* or *off*. Appending this information to the sequence diagrams yields Figure 4.

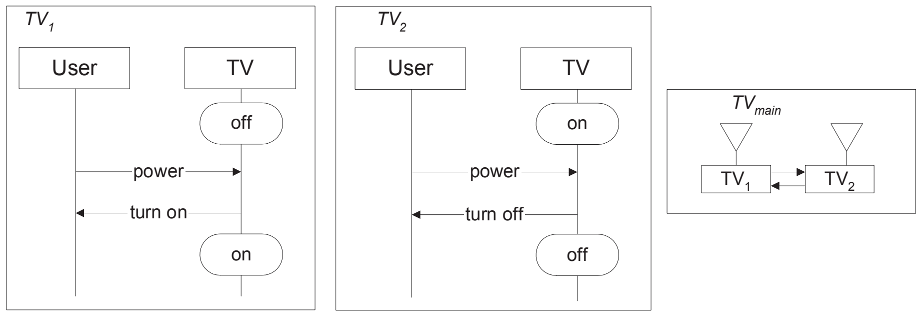

Figure 4 : Sequence diagrams for a television power switch, including state information

The new state information can be incorporated into the grammar. The template for constructing the grammar is now

$$\begin{aligned} &\text{SD} \rightarrow \textit{state}\ \text{message}\ \textbf{response}\ \ \text{SD} \mid \varepsilon \\ &\textit{state}\ \text{message}\ \textbf{response} \rightarrow \textit{off}\ \text{power}\ \textbf{turn\_on} \mid \textit{on}\ \text{power}\ \textbf{turn\_off} \end{aligned} \tag{8}$$

This is of the form

$$\text{message}\ \textbf{response} \rightarrow\ \alpha\ \textbf{turn\_on} \mid \beta\ \textbf{turn\_off} \tag{9}$$

and is therefore deterministic.

3.3 Data

Execution may depend on the value of a stored piece of data that is not directly modeled as a state or transition. Pre-conditions/post-conditions and assertions have been used to represent this additional information (e.g, [8, 13, 15]). Statements are annotated, usually in a formal language, that specify interesting properties of variables.

As an example, consider an elevator. The elevator contains a set of numbered car buttons, one per floor, that passengers use to select a destination floor. While inside the car, if a passenger pushes the button for the floor the elevator is already on, the doors will open. This is required to allow passengers inside an idle elevator to disembark at the current floor. If the passenger pushes the button for a floor other than the current floor, the doors will close. This is a common, although not universal, set of elevator behaviors. The sequence diagram set for car button behavior is shown in Figure 5.

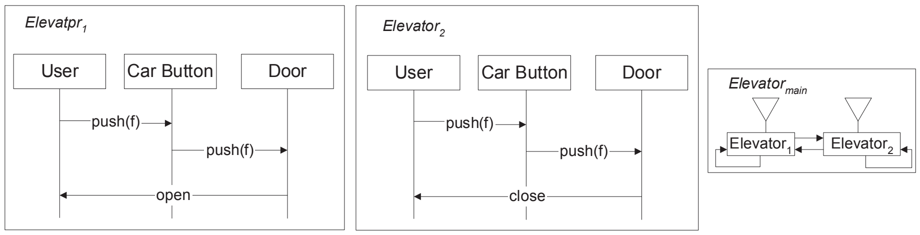

Figure 5 : Sequence diagrams for an elevator

The grammar for this example is

$$\begin{aligned} &\text{SD} \rightarrow \text{message}\ \textbf{response}\ \ \text{SD} \mid \varepsilon \\ &\text{message}\ \textbf{response} \rightarrow \text{push(f)}\ \textbf{close} \mid \text{push(f)}\ \textbf{open} \end{aligned} \tag{10}$$

This is of the form

$$\text{message}\ \textbf{response} \rightarrow \alpha\ \textbf{close} \mid \alpha\ \textbf{open} \tag{11}$$

and therefore non-deterministic.

Pre-conditions for the messages can be added to make this example deterministic, as shown in Figure 6. The crucial piece of missing information is that the response of the elevator depends on the value of (f) in push(f) compared to the current state. The value of (f) in push(f) can be either the same as the current floor or other than the current floor.

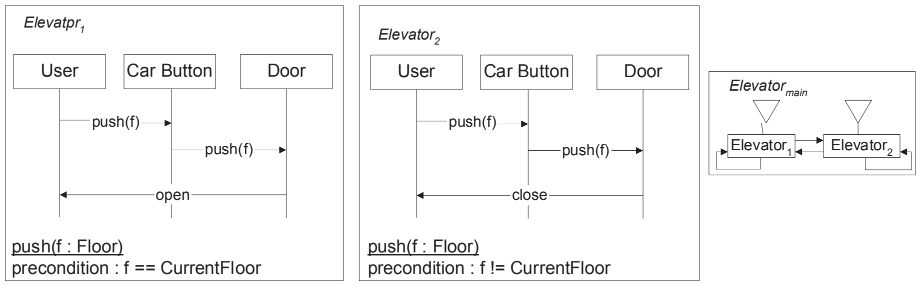

Figure 6 : Sequence diagrams for an elevator, including pre-conditions

The template for constructing the grammar with pre-conditions is

$$\begin{aligned} &\text{SD} \rightarrow \textit{pre-condition}\ \text{message}\ \textbf{response}\ \ \text{SD} \mid \varepsilon \\ &\textit{pre-condition message}\ \textbf{response} \rightarrow \\ &\quad \textit{(f == currentFloor)}\ \ \text{push(f)}\ \textbf{open} \mid \textit{(f != currentFloor)}\ \text{push(f)}\ \textbf{close} \end{aligned} \tag{12}$$

This is of the form

$$\text{message}\ \textbf{response} \rightarrow \alpha\ \textbf{close} \mid \beta\ \textbf{open} \tag{13}$$

and is deterministic.

3.4 Timing Information

The response of an embedded system may depend on timing information, such as the duration of the stimulus. Consider a car radio with a set of buttons to allow users to save and switch to preferred stations. If the button is held for a short time, the radio will change stations to the button's preset station when the button is released. If the button is

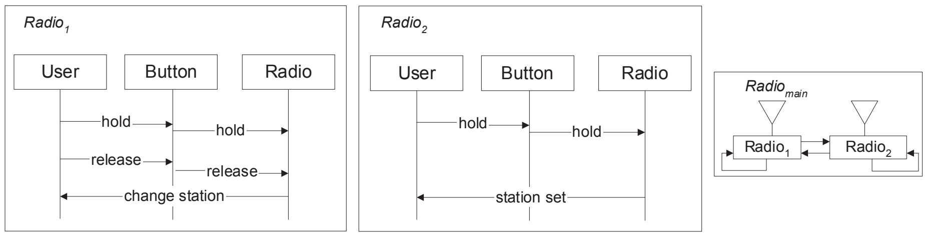

Figure 7 : Sequence diagrams for a radio

held longer, the radio will save the current station as the value of the button. The basic sequence charts for this system are given in Figure 7.

The grammar for the car radio is

$$\begin{aligned}&\text{SD} \rightarrow \text{ message } \textbf{response} \text{ SD} \mid \varepsilon\\ &\text{message } \textbf{response} \rightarrow \text{hold release } \textbf{change\_station} \mid \text{hold } \textbf{station\_set}\end{aligned} \tag{14}$$

This is of the form

$$\text{message } \textbf{response} \rightarrow \alpha \text{ release } \textbf{change\_station} \mid \alpha \textbf{ station\_set} \tag{15}$$

and therefore non-deterministic. At first glance, it may seem deterministic because of the `release` message. However, assume the system receives the `hold` message. Does it do nothing (waiting for `release`), or set the station?

Timing information is needed to express which transition should be taken. Figure 8 illustrates the car radio sequence charts with timing information included.

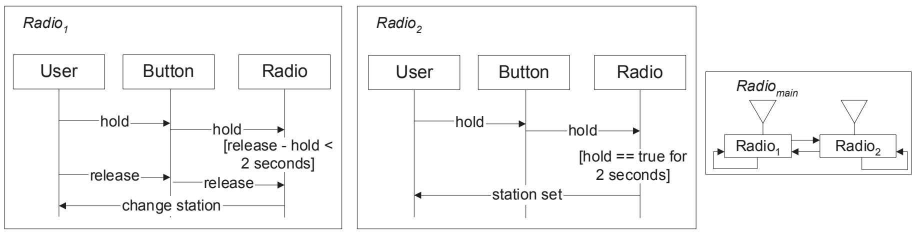

Figure 8 : Sequence diagrams for a radio, including timing information

The template for constructing the grammar with timing information is

$$\begin{aligned}&\text{SD} \rightarrow \text{message } \textit{duration } \textbf{response} \text{ SD} \mid \varepsilon\\ &\text{message } \textit{duration } \textbf{response} \rightarrow\\ &\quad \text{hold } \textit{(holdDuration < 2 seconds)} \text{ release } \textbf{change\_station} \mid\\ &\quad \text{hold } \textit{(holdDuration reaches 2 seconds)} \textbf{ station\_set}\end{aligned} \tag{16}$$

This is of the form

$$\text{message } \textbf{response} \rightarrow \alpha \text{ release } \textbf{change\_station} \mid \beta \textbf{ station\_set} \tag{17}$$

and is deterministic.

3.5 ATM example

To demonstrate the distinction between embedded systems and transaction processing systems, the classic Automated Teller Machine (ATM) example will be analyzed. Whittle and Schumann [15] synthesize statecharts from a set of four scenarios for the

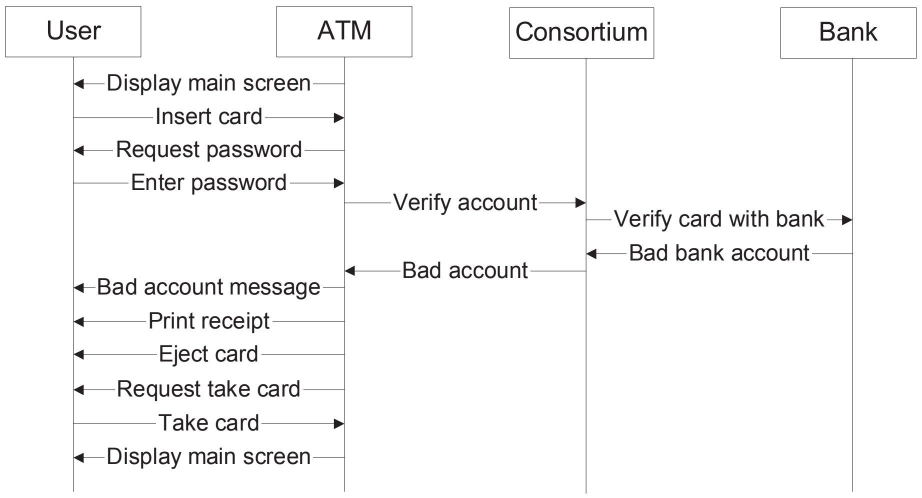

Figure 9 : Interaction with an ATM (from Whittle and Schumann [15])

ATM system. Figure 9 is the sequence diagram for the first scenario. (The complete set of SDs can be found in [15]). Four objects exchange messages : the user, the ATM, the consortium, and the bank. In this example, statecharts are generated for the ATM object only. The scenarios share the same initial condition. We constructed grammar descriptions for the set of diagrams, given in formulas 18-21. We will apply grammar parsing to locate any non-determinism present.

SD → message **response** SD | ε
message **response** → **Display_main_screen** |
Insert_card **Request_password** | Enter_password **Verify_account** |
Bad_account **Bad_account_message Eject_card Request_take_card** |
Take_card **Display_main_screen** (18)

SD → message **response** SD | ε
message **response** → **Display_main_screen** |
Insert_card **Request_password** | Enter_password **Verify_account** |
Bad_password **Request_password** |
Cancel **Canceled_message Eject_card** | Take_card **Display_main_screen** (19)

SD → message **response** SD | ε
message **response** → **Display_main_screen** |
Insert_card **Request_password** |
Cancel **Canceled_message Eject_card Request_take_card** |
Take_card **Display_main_screen** (20)

SD $\rightarrow$ message **response** SD | ε
message **response** $\rightarrow$ **Display_main_screen** |
Insert_card **Request_password** | Enter_password **Verify_account** | (21)
Cancel **Canceled_message Eject_card Request_take_card** |
Take_card **Display_main_screen**

Table 1 lists all the message-response pairs observed in the grammar for the sequence diagram set.

Table 1 : Message-response pairs for the ATM system

Message	ATM Response	Used in SD#
ε	Display main screen	All
Insert card	Request password	All
Bad account	Bad account message Print receipt Eject card Request take card	SD_1
Bad password	Canceled message Eject card	SD_2
Cancel	Canceled message Eject card	SD_2
Cancel	Canceled message Eject card Request take card	SD_3, SD_4
Take card	Display main screen	All

Note that each incoming message produces a unique set of system responses, with the exception of `Cancel`. In the second SD grammar (19), `Cancel` evokes `Canceled_message` and `Eject_card`. In the third and fourth SD grammars, (20) and (21), `Cancel` evokes `Canceled_message`, `Eject_card`, and `Request_take_card`. Upon reflection, this is likely an omission in the second sequence diagram, not a design decision.

The `Display_main_screen` message occurs before the receipt of any user messages, but does not cause non-determinism because the ATM has a single initial condition. If multiple initial conditions existed, this would pose a problem. Whittle and Schumann [15] discuss a permutation of SD_1, where `Insert_card` is repeated.

message **response** $\rightarrow$ Insert_card ε | Insert_card **Request_Password** (22)

This is non-deterministic and would mandate additional information for constructing statecharts (which the authors provided).

4 Conclusions

We have presented a methodology that guarantees *sufficient* sequence diagram information to generate correct statecharts. We convert sequence diagrams to a context-free grammar and apply parsing theory to locate non-deterministic behavior. When state, data, and timing information are included in a grammar, being LL(1) seems to be sufficient to guarantee determinism for the embedded systems we discussed. We showed how this approach identified additional information needed to attain deterministic behavior, and provided examples incorporating state, data, and timing information. Finally, we discussed a transaction processing example to show that transaction processing systems commonly used as examples tend to be more deterministic than embedded control systems.

We have also examined diagram composition and information content to assess adequacy for embedded systems. We advocate hierarchical diagrams [6, 9, 10] as the preferred format for sequence diagram composition for designing embedded systems. Hierarchical diagrams work well for expressing sequential, conditional, iterative, and concurrent execution of sequence diagrams common in embedded systems. Further, they support multiple initial conditions, one-to-many action-response mapping and timing dependencies.

Acknowledgements

This research is supported by the General Motors Satellite Research Lab at Carnegie Mellon University, Intel and the United States Department of Defense (NDSEG/ONR). Additionally, this paper would not have been possible without the scrutiny and guidance of John DeVale.

References

[1] Douglass, B. Doing Hard Time. Addison-Wesley, 1999.

[2] Glinz, M. An Integrated Formal Model of Scenarios Based on Statecharts. In Proceedings of the 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp. 254-271.

[3] Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, vol.8, no.3, 1987, pp. 231-274.

[4] Hitz, M., and G. Kappel. Developing with UML - Some Pitfalls and Workarounds. *UML '98 - The Unified Modeling Language*, Lecture Notes in Computer Science 1618, Springer-Verlag, 1999, pp. 9-20.

[5] Hsia, P. et al. Formal Approach to Scenario Analysis. *IEEE Software*, vol.11, no.2, 1994, pp. 33-41.

[6] ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996.

[7] Khriss, I., M. Elkoutbi, and R. Keller. Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams. *UML '98 - The Unified Modeling Language,* Lecture Notes in Computer Science 1618, Springer-Verlag, 1999, pp. 132-147.

[8] Koskimies, K., T. Systä, J. Tuomi, and T. Männistö. Automated Support for Modeling OO Software. *IEEE Software*, vol.15, no.1, 1998, pp. 87-94.

[9] Leue, S., L. Mehrmann, and M. Rezai. Synthesizing Software Architecture Descriptions from Message Sequence Chart Specifications. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, Honolulu, Hawaii, 1998, pp. 192-195.

[10] Li, X. and J. Lilius. Checking Compositions of UML Sequence Diagrams for Timing Inconsistency. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC 2000)*, Singapore, 2000, pp. 154-161.

[11] Louden, K. Compiler Construction : Principles and Practice. PWS Publishing Company, 1997.

[12] Somé, S., R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from User Requirements. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, Australia, 1995, pp. 48-57.

[13] Systä, T. Incremental Construction of Dynamic Models for Object-Oriented Software Systems. *Journal of Object-Oriented Programming*, vol.13, no.5, 2000, pp. 18-27.

[14] Unified Modeling Language Specification, Version 1.3, 1999. Available from the Object Management Group. `http://www.omg.com.`

[15] Whittle, J., and J. Schumann. Generating Statechart Designs from Scenarios. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000, pp. 314.