STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 3, 2011

EXTENDING UML STATE DIAGRAMS WITH BEHAVIORAL PATTERNS

DAN MIRCEA SUCIU

Abstract. State diagrams generated by reverse engineering based on the execution of software applications or those that model Interactive Voice Response (IVR) applications are usually very dense and the benefit of using *super-states* or *orthogonal regions* to increase their readability is poor. We propose to extend state diagrams with *behavioral patterns*, which are powerful constructions that could significantly reduce the number of displayed transitions.

1. Introduction

1.1. **UML State Diagrams.** UML State Diagrams [3] have their origins in finite state machines. Software systems with complex behavior were modeled in dense state machines which were difficult to read and understand. In order to increase their readability the *super-states* and *orthogonal regions* were proposed [1]. Both constructions help designers to build less complex state diagrams, with a smaller number of states and transitions, without affecting their expressibility power. Anyway, there are research fields where state diagrams still need to be refined, the current graphical syntax being insufficient to deal with tens or hundreds of states and transitions. In [5] is presented a method of generating state diagrams by observing the behavior of a software system based on its inputs and outputs. For very simple software systems, which deal with two or three variables, this kind of reverse engineering is straight forward even in the absence of *super-states* or *orthogonal regions*. As far as the number of variables is increasing, the complexity of the resulted state diagrams is exponential. In this context, one idea is to find a way to group together states having similar behavior, and enclose them in a *super-state* [2]. In practice we can find many such groups to which a state belongs, but a *super-state* could

Received by the editors: April 10, 2011.

2010 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.3 [**Software**] : Software Engineering – *Coding Tools and Techniques* D.2.7 [**Software**] : Software Engineering – *Distribution, Maintenance and Enhancements*.

Key words and phrases. UML, state diagrams, finite state machines, behavior modeling.

cover only one group at a time. This is a restriction that is solved by our proposed concept of *behavioral pattern* presented in current paper.

1.2. **IVR applications and state diagrams.** Another popular usage of state diagrams is for designing *Interactive Voice Response* (IVR) applications. IVR is a technology that allows a computer to decode humans requests via a telephone keypad or by speech recognition and which can respond with pre-recorded or dynamically generated audio. While a traditional IVR depended upon proprietary programming or scripting languages, the modern IVR applications are generated in a similar way to Web pages, using standards such as *VoiceXML*,[8] and State Chart XML (SCXML) [7]. SCXML provides a generic state-machine based execution environment based on Harel Statechart elements [1]. Commons SCXML is an implementation aimed at creating and maintaining a Java SCXML engine capable of executing a state machine defined using a SCXML document, while abstracting out the environment interfaces. For professional IVR applications with medium or high complexity, the corresponding state diagrams contain tens of states and hundreds or even thousands of transitions. The graphical view of these state diagrams is in many cases useless, due to their complexity. Our solution does not just increase the readability of such diagrams, but allow designers to define patterns which could be easily reused in subsequent IVR applications.

1.3. **Paper structure.** The next section describes the concept of *behavioral pattern* and makes a comparison with the *super-state* concept introduced by Harel. The third section presents a case study using the state configuration of a medium-size IVR application and shows how *behavioral patterns dramatically* decrease the number of transitions, while the *super-states* are not useful in this context. The last section describes the potential of behavioral patterns and some ways of extending them.

2. Behavioral patterns

The left diagram from figure 1 shows a simple state-diagram containing 3 states and 7 transitions. We can observe that every time an event *reset* occurs, the system enters in state **A**. At the same time, if the system is in state **B** or **C**, the event Event occurs and the logical expression cond is evaluated to TRUE, the system enters again in state **A**. We can say that in the first case the system behaves in the same way when an event reset occurs, and in the second case it behaves in the same way if it is in some particular states (**B** or **C**) and the event Event occurs. These are, in fact, two examples of *behavioral patterns*. If we want to use super-states in order to make the initial diagram more readable, we can group the states **B** and **C** under the

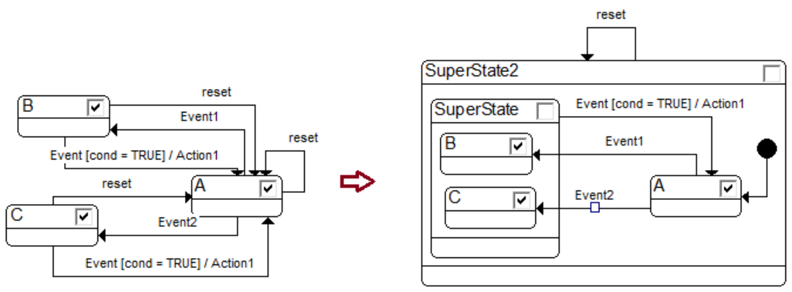
B
reset
Event1
Event [cond = TRUE] / Action1
reset
reset
A
C
Event2
Event [cond = TRUE] / Action1
reset
SuperState2
SuperState
Event [cond = TRUE] / Action1
B
Event1
C
Event2
A

FIGURE 1. Using super-states to decrease the number of transitions

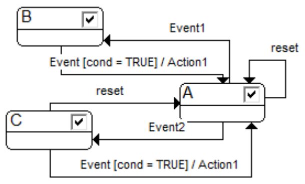

FIGURE 2. Sample state diagram

same *super-state* (let's call it **SuperState**) and nest the result together with the state **A** into another *super-state* (called **SuperState2**). The result of this operation in presented in the right diagram of figure 1. Consequently, the number of transitions decreased to 4 and, even if the final number of states is bigger now, the diagram is clearer than the initial one.

In Figure 2 we have the same initial state-diagram with only one modification: the transition triggered by event *reset* from state **B** to state **A** was deleted. This change makes impossible the usage of both *super-states* identified in the previous example. For the initial behavioral pattern we need to group together the states **A** and **C** and for the second one we need to group the states **B** and **C**. Because we cannot define super-states which have only a part of their sub-states in common, we are forced to use only one *super-state*. As a consequence, the final number of transitions we obtain is 5, more than in the previous example, even if we initially cut one transition.

Such anomalies are excluded in case of using specific graphical constructions to model behavioral patterns. Our proposal is shown in figure 3: we use doubled rounded rectangles to specify behavioral patterns and they are positioned in a separate area (called pattern area) together with modeled transitions and their state targets. In our example, state **A** was copied two times inside the pattern area to help the definition of two patterns: **Pattern1**, which describes the behavior of the system when *reset* event occurs, and **Pattern2**,

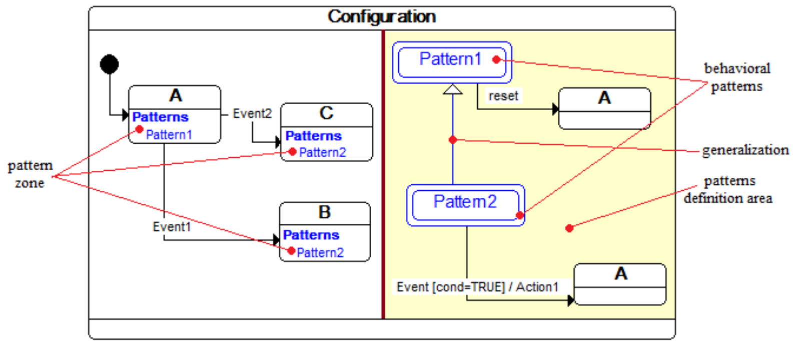

FIGURE 3. Using behavioral patterns to decrease the number of transitions

which describes the behavior of the system when event *Event* occurs and the system is in states **B** or **C**. In order to specify that a state follows a specific pattern, a new section is introduced in the graphical representation of a state called Patterns.

It makes sense to define also a generalization relation between patterns. In our example, there is a generalization relation between **Pattern2** and **Pattern1**, which means that the behavioral pattern **Pattern2** inherits the entire behavior described by **Pattern1**. Because the states **C** and **B** react at *reset* event in the same manner as A but, at the same time, react in a similar way in the presence of event *Event* and when the logical expression *cond* is evaluated to FALSE, they implement both **Pattern1** and **Pattern2**. But **Pattern2** inherits the entire behavior of **Pattern1**, so only **Pattern2** is enough to be specified as implemented pattern by **C** and **B**. If we are in conditions described by the state diagram from figure 2, there is no generalization relationship between **Pattern2** and **Pattern1**. In this case, **Pattern1** should be explicitly specified for state **C**, and the total number of displayed transitions remains the same. Therefore, the anomaly which appears in case of using super-states is solved.

3. CASE STUDY

In order to validate the efficiency of behavioral patterns, we extended *Active CASE* tool ([5]) with the following features:

- the root state of each class behavioral model has by default a pattern definition area (in order to not be confounded with an orthogonal region, it has a distinct background color);
- two new graphical elements are available for statechart editing: *behavioral patterns* and *pattern generalization* - each regular state was

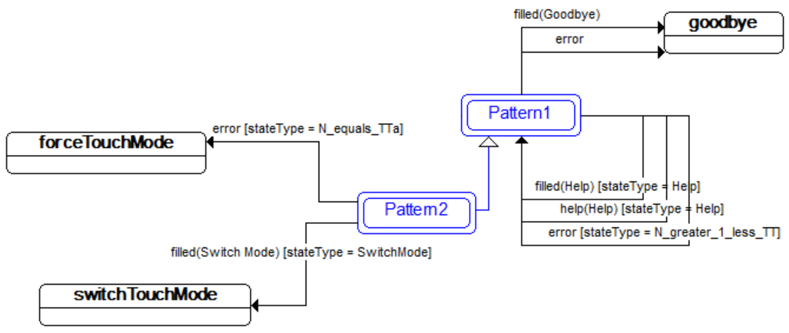
filled(Goodbye)
goodbye
error
Pattern1
error [stateType = N_equals_TTa]
forceTouchMode
filled(Help) [stateType = Help]
Pattern2
help(Help) [stateType = Help]
error [stateType = N_greater_1_less_TT]
filled(Switch Mode) [stateType = SwitchMode]
switchTouchMode

Figure 4. Automatically detected *behavioral patterns*

extended with an optional compartment for displaying the list of implemented patterns;

- a pattern generator, which automatically detects and creates *behavioral patterns* for a given statechart.

As a case study we selected the state diagram used for implementing a medium-size IVR application, containing 47 states and 708 transitions. The graphical representation of this state diagram shows a tangled net of transitions without any value for IVR designers. The pattern generator detected 30 distinct behavioral patterns, and the resulting statechart had just 195 transitions. So, more than 500 transitions where removed from the graphical representation of the state diagram using *behavioral patterns*.

Figure 4 shows two of the 30 generated patterns. **Pattern1** describes the behavior implemented by 35 states, so using this pattern 175 transitions were extracted from original state model. The second pattern, **Pattern2**, is implemented by 34 states, which means a number of 68 transitions replaced. Both patterns, together, cover almost a half of the total amount of replaced transitions. Not all generated patterns must be preserved, because many of them could not have a relevant impact in for the initial state diagram.

4. Conclusions and future work

We proved that *behavioral patterns* are sensitive superior to *super-states*, especially in those software domains which deal with state diagrams having a high level of complexity. Anyway, *behavioral patterns* and *super-states* can leave together inside the same state diagram. It is up to the system designer when to use super-states and when to use behavioral patterns in order to make the model clearer and with a high level of readability. Usually, normal state diagrams having up to 10 concrete states do not need behavioral pattern definition. There are some other interesting applications of *behavioral patterns*, which worth to be subject of future research, like:

- re-definition of state diagrams inheritance using *behavioral patterns*;
- support for automatic detection of rare events using statecharts ([4], [6]) by detecting any anomaly in *behavioral patterns* detected in different moments in time;
- definition of *behavioral pattern* at orthogonal region level, not at state level.

References

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

[2] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, Pamela Zave, "Matching and Merging Statechart Specifications", International Conference on Software Engineering, Proceedings of the 29th international conference on Software Engineering, pp 54-64, 2007

[3] Object Management Group, "OMG Unified Modeling Language Specification, Superstructure version 2.3", May 2010 (available at http://www.omg.org/spec/UML/2.3/)

[4] Vasile-Marian Scuturici, Dan-Mircea Suciu, Romain Vuillemot, Aris Ouksel, Lionel Brunie, "Detecting Anomalies in Data Streams using Statecharts", Extraction et Gestion des Connaissances (EGC'10), Revue des Nouvelles Technologies de l'Information, RNTI-E-19, Hammamet, Tunis, January 2010, pp 635-636

[5] Dan Mircea Suciu, "Reverse Engineering and Simulation of Active Objects Behavior", Knowledge Engineering, Principles and Techniques - "KEPT-2009" Selected Papers, "Babes-Bolyai"University of Cluj-Napoca, pp. 283-290 , July 2-4 2009

[6] Dan Mircea Suciu, Romain Vuillemot, Marian Scuturici, "Visual Detection of Rare Events Using Statechart", IEEE VisWeek Compendium, VAST Contest, Piscataway, NJ, IEEE. October 10, 2009

[7] World Wide Consortium, "State Chart XML (SCXML): State Machine Notation for Control Abstraction", 16 December 2010 (available at http://www.w3.org/TR/scxml/)

[8] World Wide Consortium, "Voice Extensible Markup Language (VoiceXML) 3.0", 16 December 2010, (available at http://www.w3.org/TR/voicexml30/)

Babeş-Bolyai University, Department of Computer Science, 1 M. Kogălniceanu St., 400084 Cluj-Napoca, Romania

E-mail address: `tzutzu@cs.ubbcluj.ro`