Deriving prototypes from UML 2.0 sequence diagrams

Zbigniew Huzar[1] and Grzegorz Loniewski[1]

Institute of Applied Informatics, Department of Computer Science and Management, Wroclaw University of Technology, Poland
zbigniew.huzar@pwr.wroc.pl, grzegorz.loniewski@student.pwr.wroc.pl

Abstract. Executable prototypes generated on early stages of software development bring many benefits, first of all they help to develop and validate system's specification. The paper presents an approach to automatic system prototype generation based on a collection of UML 2.0 sequence diagrams. In the approach a set of sequence diagrams representing behavior of a specified system is transformed into a state machine and next a Java code is generated for the state machine. The transformations are described informally by presentation of simple examples. Architecture of the system implementing the transformation is briefly described.

Keywords: Sequence diagram, state machine, prototype, code generation

1 Introduction

Sequence diagrams are very often used as behaviour specification of developed systems. They become very popular with advent of the UML specification language. UML sequence diagrams were adopted from message sequence charts that were known already in other visual languages developed long ago by the International Telecommunication Union. UML 2.0 also uses other diagrams to specify behaviour, e.g. communication and collaboration diagrams. All those diagrams express similar although not identical information and show it in a way different from the sequence diagrams. Sequence diagrams are used to specify scenarios as sequences of messages passing between objects. A sequence diagram represents an interaction - a set of communications among objects arranged visually in time order. They can exist in a descriptor form (describing all possible scenarios) and in an instance form (describing one actual scenario). In the paper we deal with an instance form only.
Sequence diagrams have some practical limitations. Nevertheless, they are used at very early stage of software development for rapid but usually partial specification of system's behaviour. For developers the most important factor should be quick validation of system's behaviour defined by a given set of sequence diagrams. Sequence diagrams still have an informal semantics, therefore the most

effective way of such a validation is generating system prototype and next its intensive testing.
The aim of the paper is presentation of an approach [3] to generating the system prototype on the basis of a set of sequence diagrams. The paper is organized as follows. Section 2 explains our definition of software system specification. In Section 3 the algorithm transforming such a specification into a state machine is described. Section 4 outlines the structure of a programming system generating prototypes, and describes generation of Java code. The last Section 5 evaluates the obtained results and points out some future works.

2 Systems specification

Specification of a system consists of two parts [7][8].
The first one representing static aspect of the system, consists of two elements: a class diagram and an object diagram. The class diagram reflects the set of potential configurations of the system - a set of linked objects. The object diagram refers to an initial configuration of the system.
The second one represents dynamic aspect of the system and includes a set SD of sequence diagrams. The set is partially ordered by relation <, which is defined as follows: for $sd_1, sd_2 \in SD$, $sd_1 < sd_2$ means that interaction represented by sd_1 should occur before interaction represented by sd_2.
The dynamic part should be consistent with static part of the specification. It entails that each sequence diagram should be consistent with class diagram, i.e. the objects that appear in the sequence diagram are to be elements of an object diagram being an instance of the class diagram. At least one sequence diagram should act in accordance with the initial object diagram.
We use UML 2.0 sequence diagrams [6] with the following restrictions:

- only asynchronous messages are allowed,
- combined fragments only with three main operator types: `alt`, `loop` and `strict`.

Additionally, to maintain consistency of the specification the following conditions should be hold:

- objects belonging to the initial object diagram are not created on none of sequence diagrams,
- other objects may be created only once via `create` operation and on one of the sequence diagrams, otherwise they are considered to be different objects,
- objects and their messages can not cause inconsistencies, e.g. they have to keep the time ordering resulting from objects creations and destructions times,
- at least one guard condition in a combined fragment should always be fulfilled,
- conditions in nested combined fragments should not be contradictory.

Specification example is presented in Fig. 1. The oval areas on the figure that represent some objects' activities will be used further to explain the idea of sequence diagrams transformations.

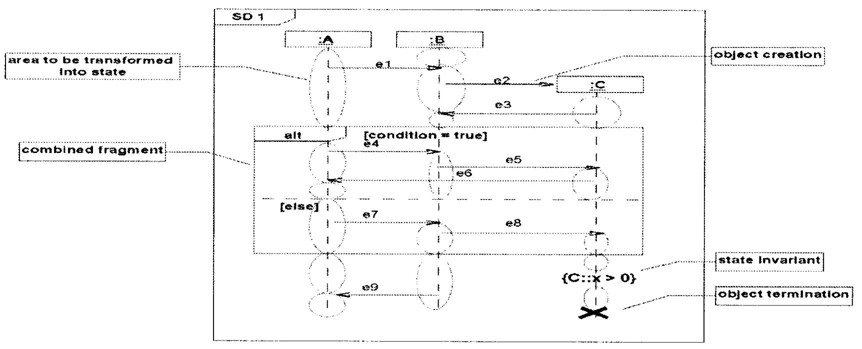

Fig. 1. Example of a sequence diagram

3 Transformation of sequence diagrams into state machines

Deriving of system prototype consists of two phases: first a state machine is created and next Java code is generated. The transformation of a set of sequence diagrams into a single state machine should preserve determinism as well as consistency. The transformation works in the following steps:

- For each sequence diagram from the given specification and for each object on this diagram a state machine is generated. The machine represents a fragment of behaviour of the class to which the object belongs. The machines are represented in form of UML statecharts.
- The state machines generated for the same class instances appearing on different sequence diagrams are merged into a single, global state machine. The merging has to maintain transitions between states located on each of composite machine. States of the whole system are represented by states of the global state machine.
- The set of states of the resulting state machine is minimized.

The idea of the first step of the transformation is explained for the specification example from the previous section. Messages on the sequence diagram which can change the object state are recognized. It is assumed that only receiving events on the object lifeline may cause changing of its state. The areas between marked messages (oval areas in Fig. 1.) are distinguished and states from these areas are derived.
Incoming messages become events triggering transitions between states, whereas the outcoming messages become actions executed as entry actions of a particular state. There are also other sequence diagram elements that produce new states like state invariant (ϵ-transition, i.e. internal transition, with state invariant condition[1]), and combined fragment (for object that begins the interaction within

[1] State invariant condition - an interaction may be continued by the object provided if the state satisfies the invariant condition

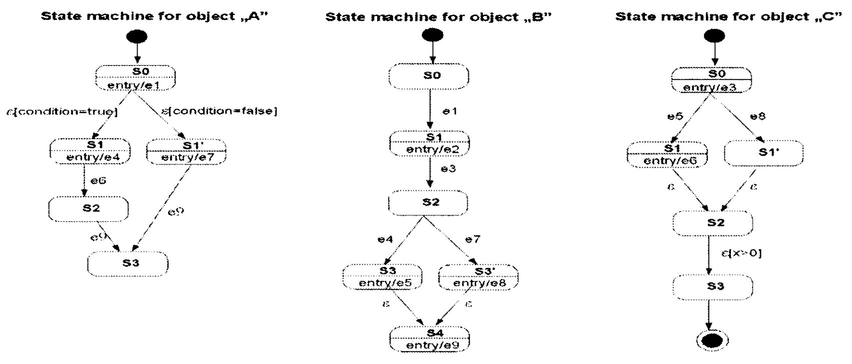

Fig. 2. Example statecharts for derived from sequence diagram

the fragment ε-transition with operand condition for other objects taking part in the combined fragment interaction new states triggered by a proper events). Statecharts created by the algorithm possess only simple states (simple - in sense of UML statecharts), and therefore they are called flat statecharts. Example of state machines generated out of a given scenario from Fig. 1. is shown in Fig. 2.

The second step of the transformation merges state machines representing behaviour of the same class. The transformation has to maintain partial ordering between state machines which results from sequence diagrams.
The state machine for a given class is constructed as follows. First, the state machine derived from a scenario which possesses objects in their default states (i.e. objects belonging to the initial object diagram) is taken as an initial state machine. Second, other state machines are attached to the initial one. The synthesis proceeds as follows. At the beginning equivalence between initial states of the initial state machine and the joining state machine is examined.
If state machines can not be joined by their initial states system looks for other states in the initial state machine that are similar to the initial state of the merged machine. If there are two equivalent states[2] and joining them will not cause indeterminist situation a synthesis decision can be taken.
However, a situation when no merging state has been found might occur. In such a case a new transition between the initial state of the initial state machine and the initial state of the merging state machine is added and marked as λ-transition[3]. This sort of transition can be executed if no other transitions are triggered and user will decide to continue the prototype execution with the λ-transition path. Two state machines merging procedures of aforementioned description are shown on simple examples in Fig. 3.

[2] states described by equal attributes values of the corresponding class and equal set of executing actions
[3] λ-transition - transition triggered only by the user decision about execution path if no other transitions are possible

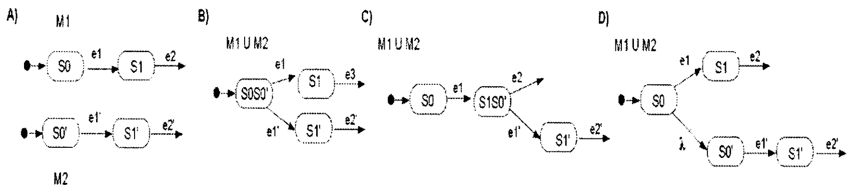

Fig. 3. State machine synthesis process:
A) initial state machines M1 and M2;
B) merging through equivalent initial states;
C) merging through equivalent states (state machines ordered);
D) merging by adding new λ-transition

The third step of transformation is the minimization process. State machines generated for objects on the base of different sequence diagrams may possess similar states. Such states are derived from identical interactions which were located on different sequence diagrams. For that reason applying the concept of interaction use is recommended. When creating system specification similar interactions can be designed on a separate sequence diagram. Synthesis process does not deal with removing similar states. To improve the state machine efficiency redundant states should be removed but still maintaing the state machine deterministic.

4 Code generation and system design

Code generation out of statechart diagrams is based on the following schema:

- state is transformed to a class where its behaviour consists of methods derived from actions that state activity consists of,
- events and respective actions are transformed into method calls,
- for each class a state variable indicating current state of its instances is implemented; assignments of that state variable correspond to transitions from a state machine,
- transitions conditions are converted into if-else statements in generated event methods, etc.

Gathering of a behaviour associated with a single state and its encapsulation in one class is an idea taken from state design pattern [5] of which created system makes use. It gives flexibility while performing some changes in specification (states activities) which results in code modifications. State design pattern also provides the architecture of Java classes.
UML state machines and Java programming language are based on different concepts. For that reason code generation is not an easy, straightforward mapping of state machine elements into the code. However, object-oriented programming and design patterns enable generation the code skeleton of executable prototype.

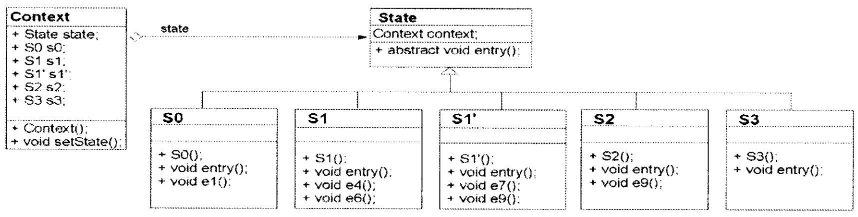

Fig. 4. Example of classes structure based on state pattern

An example of such a skeleton for the state machine for object "A" from Fig. 2., is shown in Fig. 4.

Methods deriving from state transitions are implemented as follows:

```
public void transitionMethod() throws Exception {
  if (condition == TRUE) {
    executeStateTransition();
  } else {
    throw new Exception();
  }
};
```

Testing condition in `if-else` statement is derived from a transition condition. If satisfied the transition is executed, otherwise an exception arises.

On the basis of above mentioned code generation and transformations rules, system for prototypes automatic generation was created[3]. It gets as an input system specification and produces executable Java code as an output. Functioning of the system described by UML activity diagram is presented in Fig. 5. Each action from the activity diagram corresponds to a system module. These modules work in a cascade each of which taking as its input the output of the previous module demonstrates the data flow on demonstrated activity diagram. The system includes also a graphical user interface for sequence diagrams creating and editing.

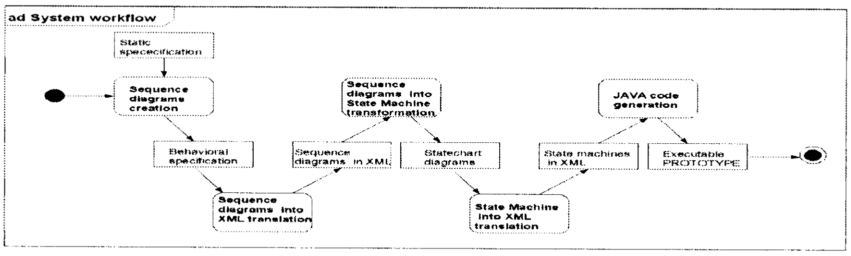

Fig. 5. System workflow activity diagram

All the necessary data present during its transformation process is stored in XMI format. This enables exchanging the specification information with other UML modelling tools.

5 Summary

The main motivation for automatic prototype generation is constructing a system that supports developing process in software. First of all, executable prototype enables validation of a specification on a preliminary stage of the project. Additionally, it provides the first vision of the developed system that can be further expanded and modified. It can also help project managers in project time estimations.
The system presented in the paper was tested by a set of simple specification examples. It appears that current version of the system may be effectively used only for non-complex specifications. This is not caused by the transformation algorithm that may cope with the system complexity but by clarity of generated code skeleton. The skeletons in present forms are not easy for extension or modification as their structure reflects final, minimally integrated state machine but there is not clear tracing to sequence diagrams or component state machines - machines representing individual classes. Solution of this discrepancy seems to be possible by another structuring of state machine intergration.
In addition it seems that within future works the system may be enhanced by new functionalities. Here are examples of issues worth to be considered:

- Providing system input not only with behavioural specification but also with prototype structure skeleton. Applied in created system state pattern for structuralizing the implementation is clear to understand but lack of readable class structure leads to illegibility of generated code. Providing the system with additional information about prototype architecture can facilitate its further expansion or modification.
- Such system should have a testing module which after generation of the final state machine will test its correctness regarding all the execution paths located on the scenarios provided as system input. On the basis of that module Java unit testing procedures can be created in order to ensure that generating based on state machines code executes properly as designed on sequence diagrams.
- The idea of expanding the system with generating graphical user interface can be taken into account. In [1] exists an approach of describing GUI by sequence diagrams. That work shows the power of UML sequence diagrams in describing systems complete specifications.
- Other idea is taking into consideration synchronous messages in the system specification. This approach allows only asynchronous messages which under certain assumptions facilitates the algorithm of constructing state machines. Objects synchronization is not necessary assuming that all objects are in proper states to receive messages from other objects (especially important when entering combined fragments).

There are also other methods that use sequence diagrams for specification of developed systems. Within them construction of state machine and further code generation often rely either on specification of additional assumptions or on a dialog with the system user providing ad hoc decisions. Such complete system (SCED) is presented in [2]. Moreover, to facilitate the process certain notation besides the UML standard is frequently used.

Whittle and Schumann in their work [9] make the most of OCL for messages pre- and post-conditions which makes the specification creation process much more intricate. An algebraic approach presented by Ziadi, Helouet and Jezequel in [10] expects that not only particular scenarios are described, but also one sequence diagram that collects and shows the connections between all other scenarios provided. In fact the final state machine is transformed from one sequence diagram. This is again the kind of user limitation to be applied when preparing system specification. Another statechart generation basing on sequence diagrams is presented by Makinen and Systa in MAS project [4]. This approach proposes interaction with user in accepting or rejecting generated state machines.

One of the aims of this work was making the system as self-reliant as possible which is to effect in limiting the system interaction with the user to the minimum, at the same time being in full compliance with the UML 2.0 notation standard.

References

1. P. Biecek and Z. Huzar. Graphical user interface and sequence diagrams in prototype generation. In Z. Huzar and Z. Mazur, editors, *Problemy i metody inynierii oprogramowania*. WNT, 2003.
2. K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, 1996. Available from: citeseer.ist.psu.edu/koskimies96sced.html.
3. G. Loniewski. State machine prototype generation on the basis of uml 2.0 sequence diagrams. Master's thesis, Wroclaw University of Technology, Poland, September 2006.
4. E. Makinen and T. Systa. Mas - an interactive synthesizer to support behavioral modeling in uml. In *23rd International Conference on Software Engineering (ICSE'01)*, pages 0–15, 2001.
5. I. A. Niaz and J. Tanaka. Mapping uml statecharts to java code, 2004. Available from: citeseer.ist.psu.edu/635242.html.
6. Object Management Group. *UML 2.0 OMG Specification*, 2005. Available from: www.omg.org/technology/documents/formal/uml.htm.
7. T. Pender. *UML Bible.* John Wiley & Sons, 2003.
8. P. Roques. *UML in Practice.* John Wiley & Sons, 2005.
9. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering*, pages 314–323, 2000. Available from: citeseer.ist.psu.edu/whittle00generating.html.
10. T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *26th International Conference on Software Engineering (ICSE 04), Edinburgh, UK*, page 242251, 2004.