From Requirements to Code Revisited

Tewfik Ziadi, Xavier Blanc, Amine Raji

From Requirements to Code Revisited

Tewfik ZIADI, Xavier BLANC
LIP6-University of Paris 6
104, avenue du Président Kennedey
Paris, France
{tewfik.ziadi, xavier.blanc}@lip6.fr

Amine RAJI
ENSIETA
Brest, France
amine.Raji@ensieta.fr

Abstract

In his article entitled "From Play-In Scenarios to Code: An Achievable Dream", David Harel presented a development schema that makes it possible to go from high-level user-friendly requirements to a full system model, and from there to the final implementation.Even if Harel's schema represents a real contribution to filing the gap between user requirements and final implementations, there is few work on its feasibility and none within UML2. This paper addresses this lack. First we use UML2 sequence diagrams as a formalism for requirement specification. Then an approach that synthesizes state machines from UML2 sequence diagrams is presented. From the obtained state machines, we implement a transformation to code. The AIBO platform (one of several types of robotic pets designed and manufactured by Sony) is used as a case study to illustrate our implementation.

1. Introduction

In his article entitled "From Play-In Scenarios to Code: An Achievable Dream" [6], David Harel presents a development schema, illustrated in Figure 1, that makes it possible to go from the high-level user-friendly requirements to a whole model of the system, and then to build the final implementation. The schema distinguishes three levels: Requirements, System model and Code. Requirements are specified as a set of scenarios while the system model is modeled by two views: Structure (like UML class diagrams) and Behavior (like UML state machines). This schema represents a real contribution to bridging the gap between user requirements and code.Harel [6] puts special emphasis on the formalisms and methodologies needed to achieve this schema. In such a context, he argues that the following three issues should be addressed:

- *A formalism to specify requirements*. Requirements specify what the user expects from the system. To be useful in the schema activities, requirements should be specified using rigorous formalisms. Scenario-based formalisms (such as UML sequence diagrams) are the most popular way to specify requirements. They take their benefits from their nature that makes them intuitive to the users.
- *A methodology for system model synthesis*. From the user requirements, developers need a methodology to synthesize in a rigorous way the system model. This synthesis allows to keep traceability between requirements and the system model. Harel [6] argues that the main difficulty concerns the synthesis of the behavioral aspect of the system model (he assumed that the structure has been already determined).
- *A methodology for code generation*. Once the model system has been synthesized, the last point towards the final implementation concerns the code generation from the system model.

Even if this schema represents a real contribution to bridge gap between user requirements and the final implementation, there are only two approaches that investigate its feasibility in the context of UML1.x [14], [4]. However to our knowledge, there is any study regarding UML2. UML2 improves over early versions in many aspects [5]. In particulary, sequence diagrams have been significantly changed in UML2. We now have a richer scenario-based formalism including new concepts, such as interaction operators that allow composing sequence diagrams to specify complex requirements. Therefore, we believe that the Harel's schema should be revisited within UML2.

This paper contributes to this. Firstly, we use UML2 sequence diagrams (SDs) and their new concepts as a formalism for requirement specification. For the second issue, this paper significantly extends the method from [15] to automatically synthesize state machines from UML2 sequence diagrams. The extensions concern the parallel composition of behaviors which is not supported in the original method. The last contribution concerns the code generation from UML state machines. We propose a generation approach, based on a well defined semantics that allows the execution of the state machine. The ideas in the paper have been implemented in a tool, PLiBS-NG, which is implemented as a plug-in to the IBM RSA (Rational Software Architect). The AIBO platform is used throughout the paper as a case study to illustrate our approach.

The rest of the paper is organized as follows: Section 2 presents the case study and the specification of requirements using UML interaction. Section 3 illustrates our two stages towards code generation. Section 4 discuses implementation and the validation issues. Section 5 presents related work and Section 6 concludes the paper.

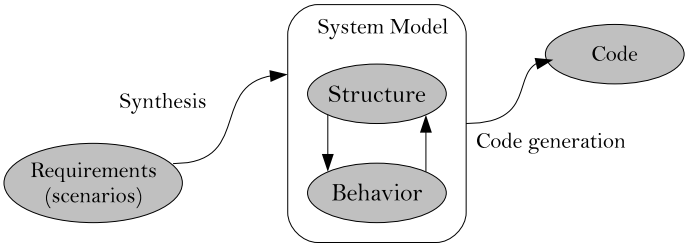

Figure 1. From Scenarios to Code

2. Requirement specification

2.1. Aibo: the Case Study

Throughout this paper we will use the AIBO platform of Sony [1] as a concrete reactive system to illustrate our approach. AIBO is one of several types of robotic pets designed and manufactured by Sony. AIBO is a reactive system that can move, see, hear and speak.

Using specific programming languages, AIBO software can be developed in order to make AIBO react (move and speak) when it receives events (hear and see). Many languages have been recently proposed to develop AIBO Software. In this paper we consider URBI (Universal Real-time Behavior Interface) [1] that consists of a scripting language and an interpreter. The interpreter runs on the robot and interprets the script transmitted by the programmer.

URBI provides a set of primitives to ask AIBO to perform basic actions (sit down, stand up, lie down, go forward, turn left). For example, the URBI `robot.stand()` primitive asks AIBO to stand. URBI also provides a set of primitives to access the AIBO sensors. For example, it is possible to recover the image observed by AIBO.

2.2. Requirements as UML2 Sequence Diagrams

Following Harel's schema (cf. Figure 1), requirements are initially specified to reflect what users expect from the system. Then synthesis and code generation activities are used to obtain an implementation that fulfils user requirements. In this paper we specify requirements using UML2 sequence diagrams. Sequence Diagrams (SD) have been significantly changed in UML2. Notable improvements include the ability to define what is called *combined SD*. A Combined SD is a sequence diagram that refers to a set of SD and composes them using a set of interaction operator. The main operators are: `seq`, `alt`, `loop`, and `par`. The `seq` operator specifies a weak sequence between the behaviors of two SD. The `alt` operator defines a choice between a set of SD. The `loop` operator specifies an iteration of a SD while the `par` operator allows specifying concurrency between SD.

To illustrate our requirement specification, let us consider an example of a behavior that the user expects from the AIBO platform. We first describe this behavior in English:

"..Before all, the user asks to initialize the robot. Then he (or she) sends sitting and standing requests (in this order). Once the robot is standing, the user will ask AIBO to walk, to search and track the ball. Walking and searching are concurrent behaviors. This behavior will be repeated infinitely.."

This requirement can be specified using UML sequence diagrams as illustrated in Figures 2.2 and 2.2. Figure 2.2 shows three basic Sequence Diagrams (SD) [2]: `Initialization`, `Walking` and `SearchBall`. The basic SD `Initialization` describes the interactions between two instances `User` (an actor) and `anAIBO` (instance of the AIBO class). The vertical lines represent life-lines for the given instances. Interactions between instances are shown as horizontal arrows called messages (like `robot.initial`). Each message is defined by two events: message emission and message reception, which induces an ordering between emission and reception. Events situated on the same lifeline are ordered from top to down.

Figure 2.2 shows a combined SD that refers to three basic SD and composes them using interaction operators. The combined SD illustrates the use of three operators: `seq` (to specify the sequence between `Initialization` and the rest), `par` (to specify that `Walking` and `SearchBall` are concurrent,i.e., they can be executed in any order), and `loop` (to specify the iteration of the AIBO behavior). The behavior specified in the combined SD is equivalent to the expression E_{CSD} defined as:

E_{CSD} = `Initialization` **`seq`** (**`loop`**((`Walking` **`par`** `SearchBall`))).

Next sections will show how URBI code can be generated from these requirement specifications.

3. System Model Synthesis

As illustrated in Figure 1, the first activity towards code generation is to synthesize the system model from requirements. While requirements specify what users expect from the system, the model represents the design of the future system that fulfils theses requirements. In Harel's

1. http://support.sony-europe.com/aibo/

2. A basic SD is a SD without interaction operators. It only shows simple interactions between instances.

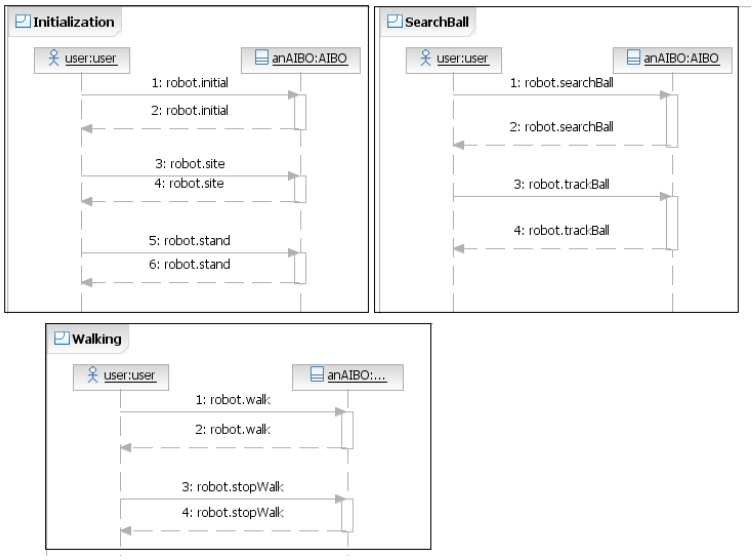

Figure 2. Basic Sequence Diagrams

Figure 3. A Combined Sequence Diagram

schema, the system model is specified in what is called XUML [6]. This includes a UML class diagram that specifies the structure and a set of UML state machines that specify the behavioral aspect. Each class is associated with a state machine which describes its full behavior. In this paper, we use this framework and we also follow Harel's assumptions to consider that the structural aspect of the system model is provided by developers.

Model System Synthesis activity is reduced to state machine synthesis from requirements. Knowing that requirements are specified as UML sequence diagrams(cf. section 2), the first activity towards code generation is, thus, defined as state machine synthesis from sequence diagrams.

One of the authors of this paper proposed a two-step synthesis method which is presented in [15]. The inputs of this method are a set of basic sequence diagrams (BSD) and one combined sequence diagram (CSD). From these inputs a state machine for each instance that participates in the CSD is synthesized.

The original method only supports three UML2 interaction operators: **`seq`**, **`alt`**, and **`loop`**. Limiting the framework to only these three operators without considering the parallel composition weakens the method for reactive systems (such as AIBO) for which the parallel execution of behaviors is a primary characteristic. The combined SD of Figure 2.2, illustrates this need; we used the `par` interaction operator to specify that the walking and searching are concurrent.

In this section we propose to extend the original framework to support the `par` interaction operator. First, we extend the definition of a state machine in [15] by introducing the notion of regions and orthogonal composite states that are necessary to formalize the parallel composition. Figure 3.1 shows an example of UML state machine that we will consider in our work. The `Example State Machine` of Figure 3.1 is defined with five simple states (`S1`, `S2`, `S3`, `S4` and `S5`). It also defined with one orthogonal state called **`CS`** with two regions separated with a dashed line. As defined in the UML standard, the behaviors specified in regions within an orthogonal state are concurrent. A transition in a state machine is labeled with e/a. The e part designates the *event* that triggers the transition where the a part is the *action* that represents an effect of the transition. The states with the «`junctionState`» stereotype represent junction states.i.e. states that are close to the usual notion of final states in classical automatas and they are introduced to formalize state machine composition [15].

Formally a UML state machine is defined as follows:

Definition 1: A state machine is a tuple $\langle SS, COS, s_0, E, A, J, \delta, \Re, \phi, subregions, \omega \rangle$ where:

- SS is the set of simple states,
- COS is a set of composite orthogonal states,
- s_0: a set of initial states [3],
- $\Re$:a set of regions,
- E: a set of events,
- A: a set of actions,
- J: a set of junction states,
- $\delta \subseteq SS \times E \times A \times SS$: a transition relation
- $\phi \subseteq SS \cup \delta \rightarrow \Re$: a relation that links each element to its container
- $subregions \subseteq COS \times \Re$: a relation that links each orthogonal state to the contained regions,
- $\omega \subseteq SS \cup COS \cup \delta \rightarrow \Re$: a relation that links each element to its container.

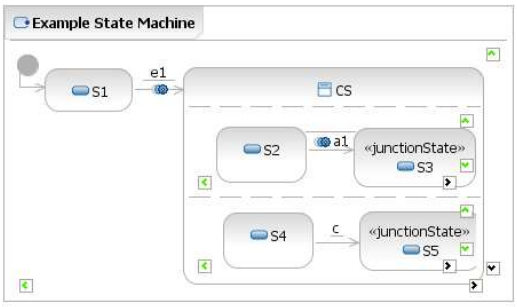

Figure 4. Example of UML state machines

Based on this definition, the next subsections present the two steps of our synthesis method. The first one generates state machines from basic SD while the second composes them with regard to the combined SD.

3.1. Step 1: From basic SD to State machines

The first step is based on an algorithm generating a state machine depicting the behavior of each object in each basic SD. We do not detail here the algorithm, which can be found in [15]. To summarize, it uses projections of basic SD on instance lifelines to generate the state machine. Receptions of messages in the basic SD become events in the state machine and emissions become actions. For a transition associated with a reception of message, the action part will be void, and for transitions associated with a reception, the event part will be empty. The generated state machine contains a single junction state which corresponds to the state reached when all events located on an instance lifeline have been executed. Figure 3.1 illustrates the state machines associated with the `anAIBO` instance which is synthesized from the three basic SD of Figure 2.2. `anIBO_Initialization` for example is the state machine obtained from the `Initialization` basic SD.

3. Initial states represent the states that are initially activated where the state machine is executed.

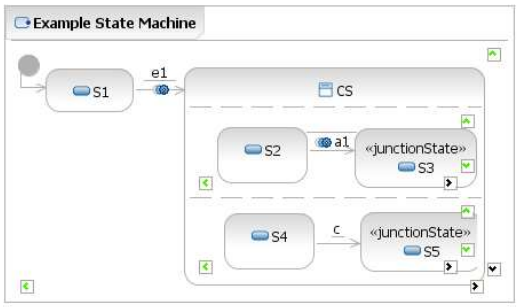

Figure 5. Example of UML state machines

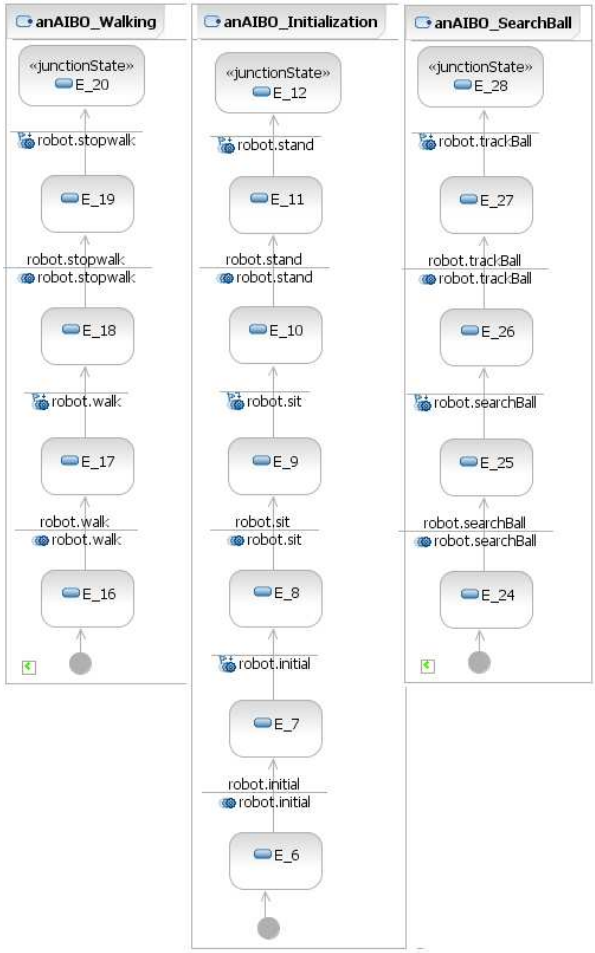

Figure 6. State machines for the AIBO object

3.2. Step 2: State machine composition

The second step consists in composing the state machines generated in the first step to obtain a complete state machine that corresponds to the combined SD. For this, the original method [15], proposes an algebraic framework for state machines. This framework formalizes three state machine operators: $\texttt{seq}_s$, $\texttt{alt}_s$ and $\texttt{loop}_s$ respectively for the sequencing, alternation, and the iteration of state machine. Then the method uses a correspondence between these operators and interaction operators to obtain the complete state machine. Extending the method to support the `par` interaction operator in combined SD, means a need to formalize a new state machine operator (noted $\texttt{par}_s$ hereafter) allowing a parallel composition of state machines. In what follows, we formalize this new operator using the notion of composite orthogonal states (cf. definition 1).

Parallel composition ($\texttt{par}_s$). The parallel composition of two state machines SM1 and SM2 produces a new state machine with a new composite orthogonal state with two regions. We put in each region all the elements (states and transitions) associated with each state machine operand.

Let $SM1 = \langle SS_1, COS_1, s_0^1, E_1, A_1, J_1, \delta_1, \Re_1, \phi_1, subregions_1, \omega_1\rangle$ and $SM2 = \langle SS_2, COS_2, s_0^2, E_2, A_2, J_2, \delta_2, \Re_2, \phi_2, subregions_2, \omega_2\rangle$ be two state machines. The $\texttt{par}_s$ is defined as follows:

Definition 2: The resulting state machine of a parallel composition of $SM1$ and $SM2$ is a state machine $SM1\ par_s SM2 = \langle SS, COS, s_0, E, A, J, \delta, \Re, \phi, subregions, \omega\rangle$, where:

- $s_0 = s_0^1 \cup s_0^2$: the initial state of $SM1\ par_s SM2$ is the union of those in the two operands.
- $SS = SS_1 \cup SS_2; A = A_1 \cup A_2; E = E_1 \cup E_2$: simple states, actions and events are the union of those in the two operands.
- $\delta \subseteq S \times E \times A \times S$: $\delta_1 \cup \delta_2$: transitions of $SM1\ par_s SM2$ is the union of those of the two operands.
- $J = J_1 \cup J_2$: junction states of $SM1\ par_s SM2$ is the union of those in the two operands.
- The result state machine contains a new composite orthogonal state os with two regions $r1$ and $r2$. $COS = COS_1 \cup COS_2 \cup \{os\}$; $\Re = \Re_1 \cup \Re_2 \cup \{r1, r2\}$ and $subregions = subregions_1 \cup subregions_2 \cup \{(os, r1), (os, r2)\}$.
- the new region $r1$ contains all elements of SM1, and $r2$ contains those of SM2. i.e. $\omega = \omega^1 \cup \omega^2 \cup \{(e1, r1) \in SS_1 \cup COS_1 \cup \delta_1 \times \{r1\} \cup \{(e2, r2) \in SS_2 \cup COS_2 \cup \delta_2 \times \{r2\}$,

Using state machines operators ($\texttt{seq}_s$, $\texttt{alt}_s$, $\texttt{loop}_s$, and $\texttt{par}_s$), state machine composition in this second step is based on a correspondence between interaction operators and state machine operators. This composition is realized by the construction of an expression for state machines from the expression associated with the combined SD.

To illustrate this construction, let us consider the expression E_{CSD} (cf. section 2) associated with the CSD of Figure 2.2. The expression for state machine is obtained by replacing the E_{CSD} `seq`, `alt`, and `par` respectively by state machine operators $\texttt{seq}_s$, $\texttt{alt}_s$, and $\texttt{par}_s$, and each reference to a SD by the state machine obtained in the Step 1. The obtained expression for state machine is E_{SM} defined as follows:

E_{SM} = `anAIBO_Initialization` **$\texttt{seq}_s$** (**$\texttt{loop}_s$** (`anAIBO_Walking` **$\texttt{par}_s$** `anAIBO_SearchBall`)).

The application of operators: $\texttt{seq}_s$, $\texttt{loop}_s$ [4], and $\texttt{par}_s$ allows obtaining a complete state machine that specifies the full behavior of the instance throughout the combined SD. The obtained one from the E_SM expression is illustrated in the Figure 3.2. It shows the behavior of the `anAIBO` instance throughout the different state machines of the Step 1. The behavior specified in this state machine between the initial state and the `E_11` state concerns the state machine `anAIBO_Initialization`. The obtained state machine also contains an orthogonal state called OS with two concurrent regions that corresponds to the parallel composition of `anAIBO_Walking` and `anAIBO_SearchBall`.

4. Code Generation

In the previous section, we illustrated the translation from scenarios to state machines. The last stage in Harel's schema is to generate executable code from the obtained state machine. In our context, this generation would be shown as a transformation from UML state machines to URBI code. Defining this transformation means a need to implement on URBI an appropriate semantics for UML state machines, including mechanisms for event management and concurrency processing. To avoid such need, we decided to delegate the state machine semantic management to an existing framework. Therefore our transformation can be formalized as a mapping between our state machines and the reused framework concepts. We have chosen the *PauWare* [2] framework which is a Java API providing means to define an UML2 state machine,.i.e, defining its states (including composite orthogonal states), events, actions, and transitions. *PauWare* then implements run-to-completion semantics as defined in [10] to execute UML2 state machines.

The principles of our approach for code generation is illustrated in the Figure 8. First we generate the *PauWare* code corresponding to the complete state machine obtained

4. The complete formalization of $\texttt{seq}_s$ and $\texttt{loop}_s$ can be found in [15]

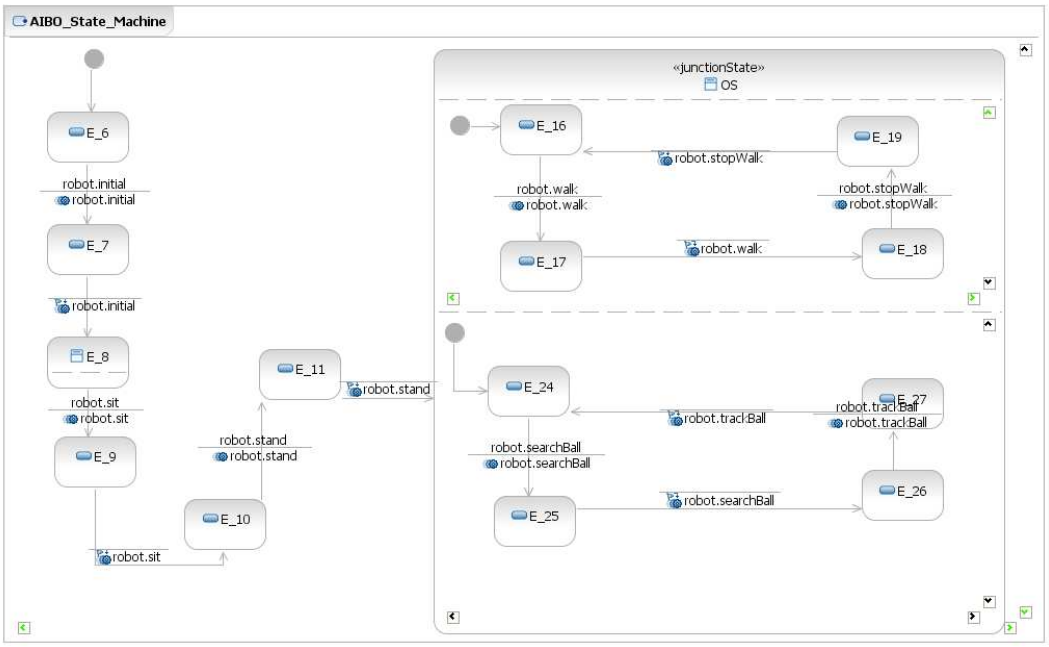

Figure 7. The complete state machine for the AIBO object

in the previous section. This allows executing our state machine following the *PauWare* semantics (cf. Part 1 in the Figure 8). Then URBI primitives specified in the state machines are sent to the robot to be executed (cf. Part 2 and 3 in the Figure 8). Finally the robot can send notification about executed actions and environment events to continue the execution of the *PauWare* state machine (cf. Part 4 in the Figure 8).

We implemented this generation as Model-to-text transformation. This includes a transformation from UML state machine to *PauWare* java code. Such code consists in declaring states, events, actions and setting up transition of the state machines according to the *PauWare* syntax.

Below a portion of the *PauWare* java code generated from the state machine of the Figure 3.2 contains the definition of all states in the state machine.

```
   public class AIBO_State_Machine extends
Timer_monitor {
  static public UClient c = new UClient;
     /** UML states */
  protected AbstractStatechart _E_6, _E_7,
          _E_8, _E_9;
  protected AbstractStatechart _E_16, _E_17,
          _E_18, _E_19;
  protected AbstractStatechart _E_24, _E_25,
          _E_26, _E_27;
```

Following the *PauWare* syntax, the instantiation of states and their mutual linking have to occur within a dedicated method named `init_behavior` (see below). To link states, the *PauWare* API proposes two principal methods: `xor` (an exclusiveness relationship) and `and` (orthogonality operator). The code below shows the a portion of the `init_behavior` method associated with the AIBO state machine. The operator ***xor*** is used to link states within a same region, while the ***and*** operator is used to instantiate the OS orthogonal state.

```
   /** INIT_BEHAVIOR method*/
protected void init_behavior()
   throws Statechart_exception {
..............
// mutual linking through exclusiveness
_A1 = (_E_16.xor(_E_17).xor(_E_18).
   xor(_E_19)).name("_A1");
_A2 = (_E_24.xor(_E_25).xor(_E_26).
    xor(_E_27)).name("_A2");

//mutual linking through orthogonality
_OS = _A1.and(_A2).name("_OS");
}
```

Then for each entry action in the state machine, a corresponding method is created in the *PauWare* java class. The method `robot_initial` below concerns the action `robot.initial` in the state machine. It contains URBI primitives that will be sent to the robot using an existing communication protocol called UrbiLib (cf. the UClient class). In the same, communication between the robot and

the *PauWare* state machine is also performed using UrbiLib.

```
/** UML actions */
  public void robot_initial() {
    c.send("robot_initial:robot.initial;");
    ............
    c.setCallback(robot_initial,
       "robot_initial");
  }
```

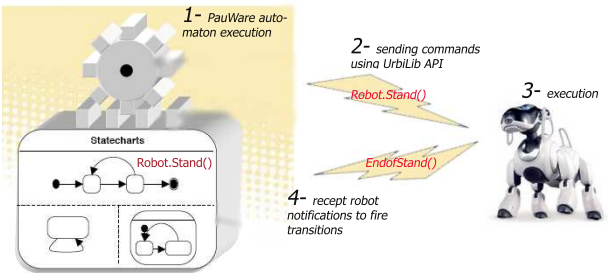

Figure 8. Code generation principals

5. Discussion

5.1. Implementation

The complete chain for code generation, presented in Section 3, and 4 has been implemented in a tool, PLiBS-NG, which is implemented as a plug-in to the IBM RSA (Rational Software Architect). More information about PLIBS-NG can be found in [5]. Our implementation is materialized by a succession of two main model transformations: State machines synthesis from UML2 sequence diagrams (as a Model-to-Model transformation) and a PauWare code generation from the obtained state machines (as a Model-to-Text transformation) We used the PLIBS-NG on the requirement presented in this paper and on ten others requirements. The execution of the generated *PauWare* code allows the execution of the synthesized state machine. Therefore the URBI primitives are sent to the robot, thanks to UrbiLib communication protocol, to be executed. In addition to AIBO, we validated our approach on the well-known ATM example. This includes eight sequence diagrams.

5.2. Discussion of the Approach

Limitations. The approach presented in this paper is completely integrated with UML2. However it has some limits concerning the used method for system model synthesis (cf. Section 3). The first one concerns values of parameters in sequence diagrams. Thus, it is necessary in some cases to specify the values of some parameters in messages within a sequence diagrams. For example, the AIBO primitive `robot.walk` allows specifying the distance as a parameter. The used synthesis method does not support the use of parameters and by consequence our code generation approach. The second limit concerns the use of guards associated with messages which is not yet supported in our approach.

As presented in the section 4, our code generation is based on the state machine semantics of *PauWare*. This semantics is close to the semantics proposed in UML2 [10]. However it does not supports the semantic variation points defined in the standard. Our next investigation will concern this aspect by integrating existing framework on state machine semantics, such as [3].

The dream: is it achieved? . Within UML2 and applied on the AIBO platform, the dream is achieved. We started from requirements specified as sequence diagrams and we generated a code which execution on AIBO fulfils requirements. Thus, the robot executes exactly behaviors specified in the sequence diagrams. However, we believe that this generation is achieved because AIBO is a reactive system. Indeed AIBO, or URBI precisely, provides a set of primitives (e.g.; `robot.stand`) that allows controlling the robot. Therefore, the generated code is only a control code which relates the AIBO actions to the events received from its environment.

How this development schema can be used?. As mentioned by Harel [6], we always need to develop system incrementally (with various cycles). Thus, the proposed development schema can be used for rapid prototyping and not for a direct bridge between requirements and code. Developers generate a first implementation from requirements which can be continually refined and extended after. In this paper, we only focused a subpart of the whole development cycle, that is code generation. We do not investigate issues concerning the code to requirement relationship and the verification between code and requirements after several iterations [6].

6. Related work

Even if the Harel's schema(cf. Figure 1) represents a real effort to bridge the gap between user requirements and final implementation, to our knowledge, there is no study on its feasibility within UML2. There are only few work regarding UML1.x. Within NASA Ames,*Whittle et al.* [14] applied the synthesis method defined in [12] to generate state machines from UML1.x sequence diagrams. Then, the authors used Rational Software RealTime to generate a C++ code from the obtained state machines. The conclusion of authors showed that the results are very interesting in the context of the Air Traffic Control case study. This

5. http://pagesperso-systeme.lip6.fr/Tewfik.Ziadi/plibs/PLIBS.html

approach was proposed before the standardization of the UML2. Therefore, it does not supports interaction operators and it only concerns basic sequence diagrams. Note that the Whittle's state machine synthesis has been extended to UML2 in a recent paper [13]. However to our knowledge, there is no validation of the new synthesis method from the perspective of code generation.

ElKoutbi et al. [4] presents a similar approach for prototyping User Interface. This approach is based on a method which synthesizes stat machines from UML1.x collaboration diagrams. Then the approach generates user interfaces from the obtained state machines. As *Whittle et al.* 's, this approach does not supports UML2 interaction operators.Two existing works [8], [11] deal with code generation from scenarios but without using state machines. Code is directly generated from scenarios by means of a set of transformation rules. We believe that code generation throughout state machine synthesis is more useful because it gives to developers a system model which can be refined and studied and used for other activities such as verification or testing. In this paper we chose the *PauWare* framework to execute our state machines. However there are other works on code generation from state machines that can used. The last one is the *Chauval et al*.'s work [3] which presents a complete approach with variation point management. Another aspect related to this paper is the "state machine synthesis from scenarios" domain. This domain have received a lot attention in recent years. *Liang et al.* [9] presents a nice comparative survey on all existing works. Finally, note that the *Harel et al*. [7] proposed in last years the Play-In/Play-out approach as an extension of the development schema studied in this paper. In this new new approach, *Harel et al*. uses LSC (Live Sequence Charts) as a formalism for requirement specification. Also, *Harel et al*. tended to focus less on synthesizing state machines and more on executing the scenarios directly . As *Harel et al.* mentioned [7] (pages. 24), the Play-In/Play-out can be used conjointly with the schema considered in this paper.

7. Conclusion

The paper studied the feasibility to go from high-level user-friendly requirements to final implementation in the context of UML2. We used UML2 sequence diagrams to specify requirements. Then a previous work on state machine synthesis have bee extended and integrated. Finally a code generation approach is proposed to complete the chain. The application of this work on the AIBO platform given interesting results and concludes that the Harel's dream can be achieved using UML2. Thanks to the richer scenario language (UML2 Sequence Diagrams) including in UML2. However, we believe that the code generation is achieved because AIBO is a reactive system . Also, we believe that the generated code is only a control code which relates the AIBO actions to the events received from its environment. The ideas in the paper have been implemented in a tool, PLiBS-NG, which is implemented as a plug-in to the IBM RSA (Rational Software Architect).

Future work will investigate the validation of the approach on other case studies, such as [14].We also plan to consider other advanced concepts in UML2 SDs, such as values of parameters in messages.

References

[1] Jean-Christophe Baillie. Universal programming interfaces for robotic devices. In *sOc-EUSAI '05*, pages 75–80, New York, NY, USA, 2005. ACM.

[2] Franck Barbier. Pauware users'guide. Technical Report ver. 1.1, PuaWare Research Group, UPPP, Pau, 04 2008.

[3] Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In *Proc. of MODELS/UML'2005*, volume 3713 of *LNCS*, pages –, Montego Bay, Jamaica, October 2005. Springer.

[4] M. Elkoutbi, I. Khriss, and R.K. Keller. Automated prototyping of user interfaces based on UML scenarios. *Automated Software Engineering*, 13(1):5–40, January 2006.

[5] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-driven development using uml 2.0: Promises and pitfalls. *Computer*, 39(2):59–66, 2006.

[6] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[7] D. Harel and R. Marelly. *Come, Let's Play*. Springer, 2003.

[8] Sang-Uk Jeon, Jang-Eui Hong, and Doo-Hwan Bae. Interaction-based behavior modeling of embedded software using uml 2.0. In *Proc. IEEE ISORC'06*. IEEE Computer Society, 2006.

[9] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. of SCESM '06*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[10] Object Management Group OMG. Unified Modeling Language Specification version 2.1: Superstructure. Technical Report formal/2007-02-03, OMG, February 2007.

[11] Mathupayas Thongmak and Pornsiri Muenchaisri. Design of rules for transforming uml sequence diagrams into java code. In *Proc. IEEE ASPEC'02*, page 485, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[12] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of ICSE '00*, pages 314–323, New York, NY, USA, 2000. ACM Press.

[13] Jon Whittle and Praveen K. Jayaraman. Generating hierarchical state machines from use case charts. In *(RE'06)*, pages 19–28. IEEE CS, 2006.

[14] Jon Whittle and Richard Kwan nd Jyoti Saboo. From scenarios to code: An air traffic control case study. *Software and Systems Modeling*, 4(1):73–93, Fev 2005.

[15] T. Ziadi, L. Hélouët, and J-M. Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *Proc. of ICSE '04*, pages 242–251, Washington, DC, USA, 2004. IEEE Computer Society.