Tamagotchi in ObjectGEODE

23rd March 1999

Jürgen Jeitner
jeitner@informatik.tu-muenchen.de

Bernd Müller
muellerb@informatik.tu-muenchen.de

Alexander Wisspeintner
wisspein@informatik.tu-muenchen.de

Department of Computer Science
Technische Universität München

Contents

1 ObjectGEODE and SDL

1.1 Introduction

ObjectGEODE is a commercial CASE (Computer Aided Software Engineering) tool for developing reactive real time systems. It combines several different methods for describing static and dynamic aspects of a system. The developer can use the description techniques OMT (Object Modeling Technique), SDL (Specification and Description Language) and MSC (Message Sequence Chart) in combination for specification [Ver98].

SDL

The main features of ObjectGEODE, like code generation and simulation, can only be used on SDL specifications. SDL has been defined in the early seventies by the ITU (International Telecommunication Union) former CCITT (Consultative Committee for International Telegraph and Telephony) as a language for describing telecommunication systems [BH93, page 94]. A first standard of SDL was set in 1976, allowing to specify behavior of systems using state machines. Up to now many extensions to the original standard has been made. Today SDL supports object oriented design and allows the description of hierarchical structures. The latest SDL standard has been recommended 1996 [Soc98]. ObjectGEODE uses the older standard SDL92.

Notation

SDL offers a structural and a behavioral view of a system. An SDL system and its environment are conceived of as a structure of blocks connected by channels. Blocks and channels may be decomposed into blocks and channels recursively over several levels until the basic components and processes are reached. Figure 1 shows a SDL structure diagram for a simple light switch. A "User" can interact with the switch via the "Interaction" channel by sending a "Switch_On" or "Switch_Off" signal. Depending on the current state of the "Switch" component, the "Light" is turned on or off via the "Wire" channel.

SDL uses extended finite state machines to describe the behavior of single components. A single component with an assigned FSM is called process. Figure 2 describes the behavior of the above explained system "Light Switch". At startup of the system, the FSM turns into the state "Off". When a signal of type "Switch_On" occurs, the subsequent state is "On". While this transition is fired a signal "Light_On" is sent to "Light". The input signal "Switch_Off" fires the transition between state "On" and subsequent state "Off" by sending out the signal "Light_Off" to the component "Light".

There are no shared data to be found outside the processes, so signals are the only means for processes to communicate. There is no way for one process to directly manipulate data of another process. Signals have no priority during processing. Signals arriving at a process are merged into

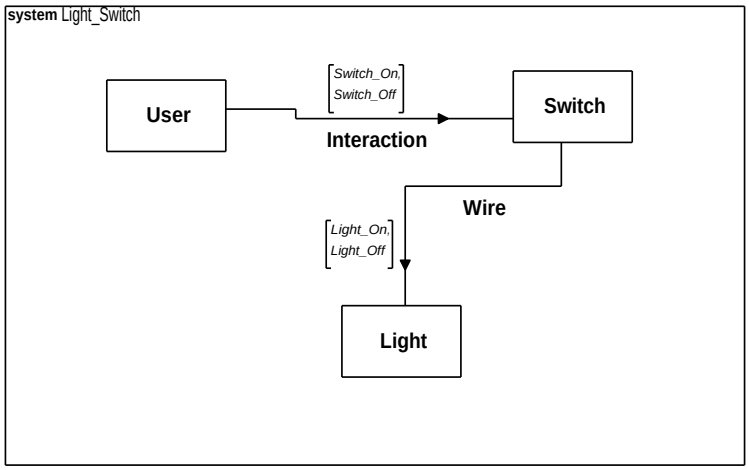

Figure 1: SDL Structure diagram of the system “Light Switch”

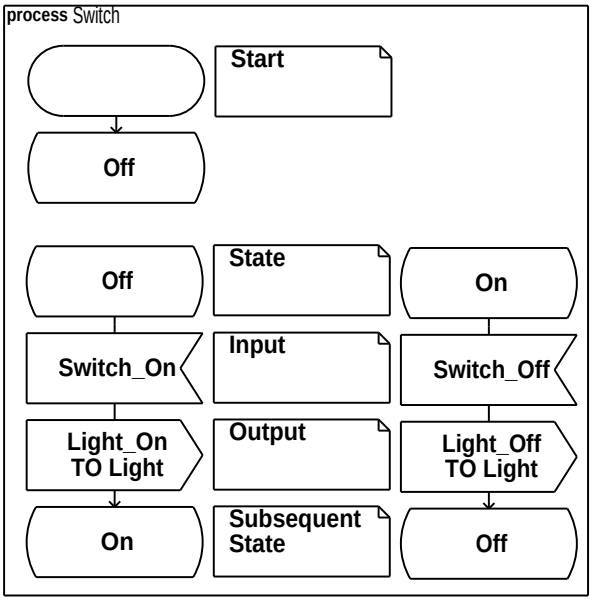

Figure 2: SDL finite state machine of the component “Switch”

one single queue in the order in which they come in. Each process owns a single signal input queue.

Time representation

SDL was developed for use in cases where explicit specification of time-related matters is of great importance. With the help of the SDL primitive “Time” a timer can easily be described. Timers are declared similarly to variables and SDL provides several methods to work with them. Until a timer is not set it is inactive. A Timer is activated when setting a time value. After a specified duration the timer becomes inactive and sends a message to the related process. The process can reset a timer to inactivate it so that no signal will be issued. The operator “Active” returns the state of a timer. The reserved word “Now” contains the current real time. A data type for representing time intervals is the type “Duration”. The already mentioned data types and its methods fulfill the time specific needs of the users.

SDL offers primitive data types known from common programming languages like “boolean”, “integer”,”real” and different array types. Further more the user can specify custom data types like structures based on primitive data types. The definition of enumeration data types is also possible. Operators and methods, working on user defined data types, can be declared by the programmer.

Tool ObjectGEODE

The CASE tool ObjectGEODE combines several different graphical editors for system specification. A specified system can be checked on consistency criteria. Such traces can be used in combination with the simulation tool to verify correct behavior. There is also the possibility of random tests. A random test is generating random events from the environment and checks the reaction of the system.

C code can be built directly out of a SDL specification. This C code can be compiled for different platforms using a cross compiler. The execution of the resulting program on the target platform can be traced from a host computer. This tracing is done with the “Design Tracer” tool.

Both, “Design Tracer” and the simulation, allows the recording of a session for later replay. MSCs can also be generated automatically after a run.

Textural requirements can be added to a system specification in form of an ASCII file. Furthermore externally generated encapsulated Postscript files can be organized in the ObjectGEODE Project.

Methodology

SDL has hierarchical views of the static structure of a system. We describe a system structure at a high level of abstraction. In the following process we refines the system components.

First we describe the interface between the environment and the embedded system. This interface is part of the requirements of our Tamagotchi. It contains the definition of used signals and the channel structure. We use a Top-Down methodology to refine our structure using different layers of blocks (SDL components). A block can consist of sub-blocks. This refinement is done until the desired level of abstraction is reached.

We describe the behavior of all single components at the lowest hierarchical level. The behavior is specified in SDL using finite state machines.

After the behavior of all components is specified, the specification in a special simulation environment can be tested. These test runs can be used to validate the requirements of the specified system. The simulator can generate MSCs automatically during a test run. We can use these message traces for troubleshooting.

1.2 Approach and Result

We specify the software of a "Tamagotchi" to analyze the development process of embedded systems using ObjectGEODE. This embedded system is relatively complex and has many time critical requirements. Therefore it is a good example system for a case study analyzing the suitability of ObjectGEODE on this field.

The first modeling step of the "Tamagotchi" system is the definition of the system-environment interface. We use the methodology introduced in Section 1.1 during the whole development process. The structure of the system specification is illustrated in Section 1.3.1. The specification of the full behavior of our components is very extensive. Here we present only a small part of it, namely the functions for playing with the Tamagotchi (Section 1.3.2).

ObjectGEODE has a revision control mechanism. In this way several users can work simultaneously at the same project. During our work with ObjectGEODE only one user license was available. Consequently we were not able to use the multi user facilities of the tool.

Our specification of the "Tamagotchi" system implements all defined textual requirements. The complete menu control and all aspects of the "Tamagotchi" life cycle are specified. This includes all time critical requirements.

The simulation of the specified system during the development process was not possible. ObjectGEODE only allows the simulation of complete consistent models. Therefore a simulation can only be done when all components of a system are completely specified.

To verify and validate a system specification, ObjectGEODE offers consistency checks and a SDL simulator. The consistency checks can only find very primitive errors. But you can use the simulator to validate the system behavior. We have tested all textual requirements until the

teenage stage of the “Tamagotchi”. The special requirements for the adult stage are implemented in the specification, but we did not make extensive tests in this evolution state.

The simulator is a good tool for finding specification faults. We have found a lot of those inconsistencies using the simulator. Finally we have solved all known problems in the specification.

The final SDL specification contains a total number of 29 diagrams. 5 SDL structure diagrams describe the static aspects of the system and 24 FSMs are used to specify the behavior. The printout of the whole specification requires 136 pages. This huge space requirement may be reduced a little bit by doing some layout efforts, but it is a fact that SDL specifications require a lot of space.

1.3 Specification

1.3.1 Structure

The first step in specifying a system is requirements specification. In this phase the interface between the environment of a system and the embedded system itself is defined.

Structure “Tamagotchi Microcontroller”

Figure 3 shows the top level of the hierarchical structure of the “Tamagotchi” system. The “Tamagotchi” has three buttons, a reset stripe, a display and a buzzer. The three buttons are represented by the channels “Select Button” (left button), “Acknowledge Button” (middle button) and “Cancel Button” (right button). You can send signals about the button states via these channels. The special channel “SelectCancel” is used to inform the controller when both, the select and cancel button are pressed simultaneously. The reset stripe generates a reset event to initialize the whole system.

The display shows the evolution state and menu information of the “Tamagotchi” system. For example the signal “ES_Display” includes the information about the current stage of our creature. Furthermore the signal list “Menus” is used to send menu information to the display. The “Tamagotchi” has a buzzer to signalize critical situations of the creature. The micro-controller uses the channel “Buzzer” to communicate with this buzzer.

The “Tamagotchi” can display the current time. In our specification, the clock of the Tamagotchi is not part of the micro-controller. An external clock device is used to deliver the required information. This external clock sends the current time (hour, minute and second) via the “Time” channel to the controller. Time signals are sent every second.

This external clock is not only used to display the current time at the display. It is also used to watch all time critical requirements of the system.

After the environment-system interface is defined, we can refine the specification. The top level is split into two parts. The “Controller” watches all input signals from the environment and

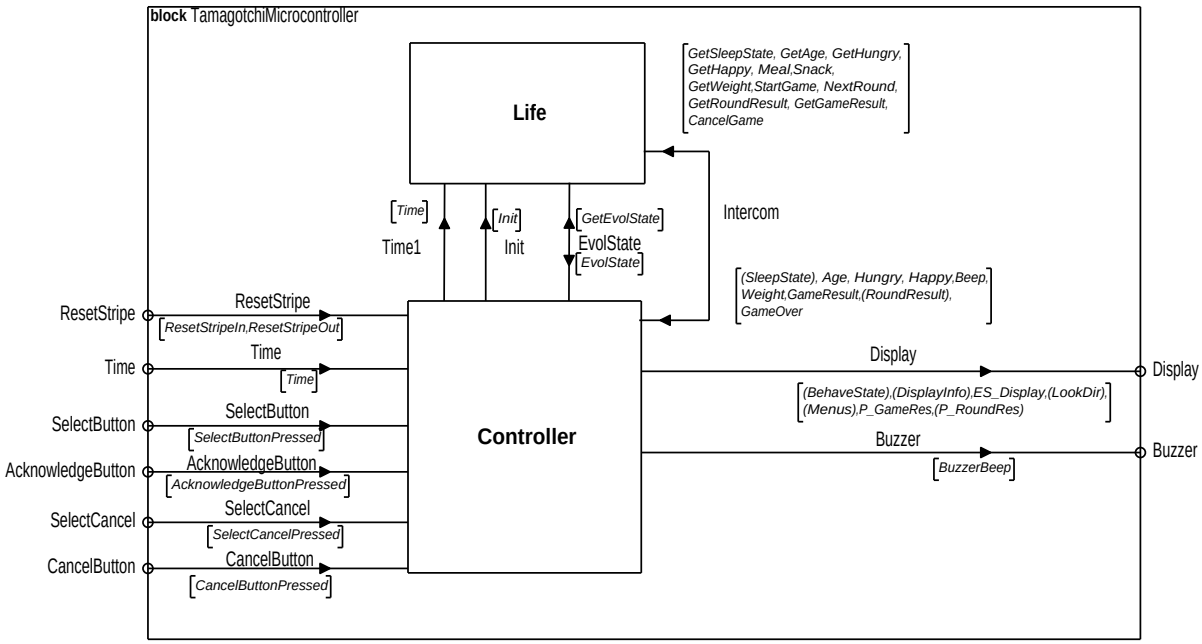

Figure 3: Structure of the component “Tamagotchi”

implements the user interface. All output signals are also generated in this component. The second part of our “Tamagotchi Microcontroller” is the “Life” component. This part handles all aspects of the creature’s life.

These two main parts communicate via channels as shown in Figure 3. Most of the inter-component communication is done via “Intercom”. Only the information about time, reset and evolution state is treated separately.

Structure “Controller”

The component “Controller” (Figure 4) consists of several elementary processes. The “Router” only distributes incoming signals to different processes. This is necessary because only point-to-point communication is allowed in SDL. “Time Serial” generates an unique time serial number. The accuracy of this serial number is one second and it is used to watch time intervals within the “Menu Timer” process.

The main component of “Controller” is the “Menu” process. It implements the whole user interface including menu functions and the display control. The “Menu” uses “Menu Timer” to set 5 second timeout intervals when displaying a menu item. After exceeding this time interval the “Menu Timer” sends a “Timer Event” signal to “Menu” and “Menu” closes the menu item and displays the normal “Tamagotchi”. The “Buzzer Controller” process only takes care of enabling and disabling the buzzer.

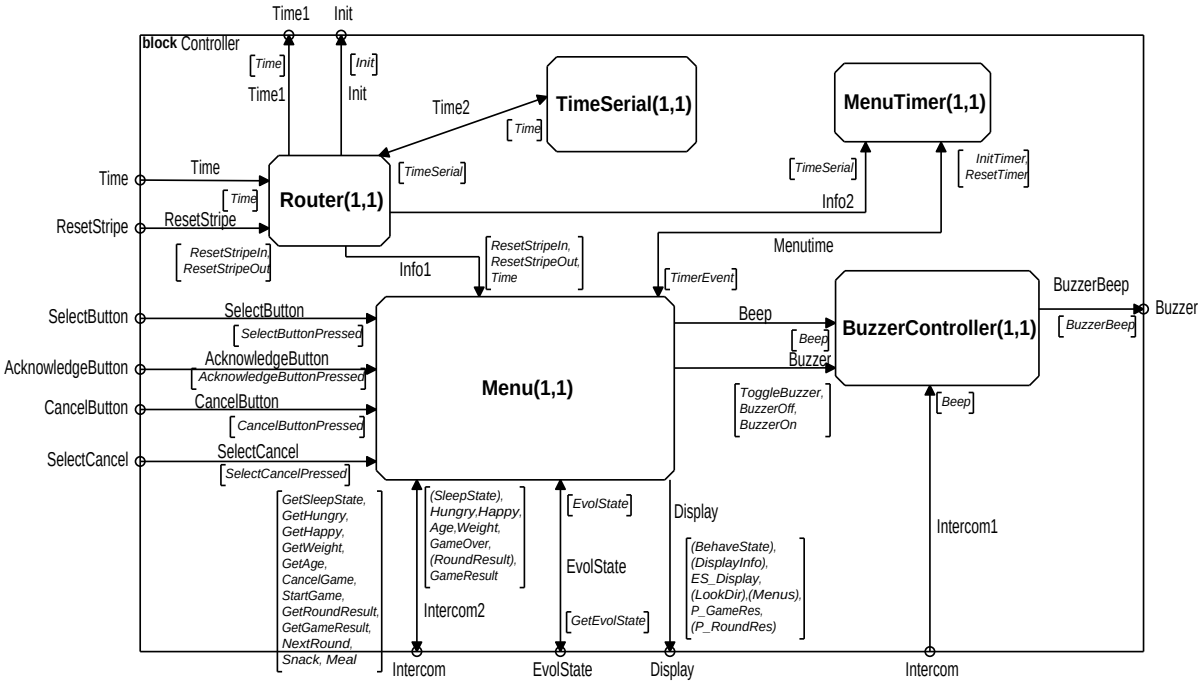

Figure 4: Structure of the component “Controller”

Structure “Life”

Figure 5 shows the structure of the component “Life”. The main process of “Life” is “Life Cycle”. It describes the evolution process of the creature and initializes all other “Life” processes depending on the evolution state.

“Life Time” generates a time serial number from the external time. This number indicates the total life time in minutes of our creature. The component “Sleep” watches the time and generates signals when the creature falls asleep and awakes.

“Hungry” automatically decrements the satiety after a specified time interval and watches several hungry criteria. If the satiety is 0 a whole day long, the component “Hungry” instructs “Life Cycle” to kill the creature. Furthermore the controller component can ask for the current satiety value.

The “Age” process increments the age value, every time the creature awakes. The life expectancy is also watched in this component. “Weight” stores the current weight of our creature and watches the weight limits. Similarly the “Happy” process has a variable to store the happiness.

The “Play” component implements the part of the play feature that is in related to the creature’s life. The display control and user interface of the play facility is integrated in the “Menu” process of the “Controller” block.

Finally we have the “Signal Watch” process. It watches critical situations of the creature. A critical situation occurs, if the satiety or the happiness falls below a value of 3. In this case

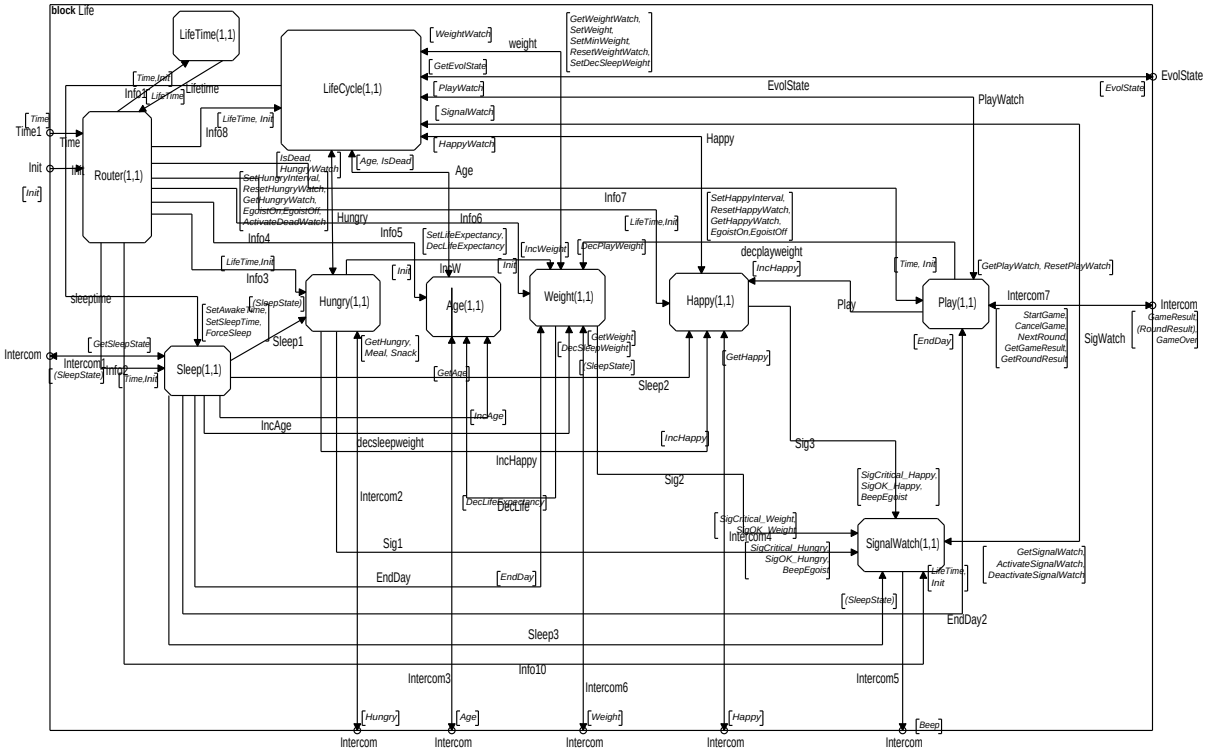

Figure 5: Structure of the component “Life”

“Signal Watch” validates if the user solves the problem within two hours.

1.3.2 Behavior

After we have designed the whole structure of the system it is now possible to create the individual finite state machines to describe the behavior. The specification of the behavior consists of 24 FSMs. Therefore we illustrate only an extract of the whole specification.

Life Cycle

As mentioned in Section 1.3.1, the “Life Cycle” process covers the evolution process of the creature. The single states “Egg”, “Babytchi”, “Marutchi”, “Tamatchi”, “Kuchitamatchi”, “Mametchi”, “Kuchipatchi”, “Masktchi” and “Flying Tamagotchi” of the “Life Cycle” FSM stand for the different main evolution stages of the same name.

Figure 6 shows all transitions starting at the state “Babytchi”, the baby state of the creature. The most interesting transition is placed on the left side of the figure. An incoming “Age” signal triggers this transition. If the age of our creature is greater or equal 1, the baby creature becomes a child.

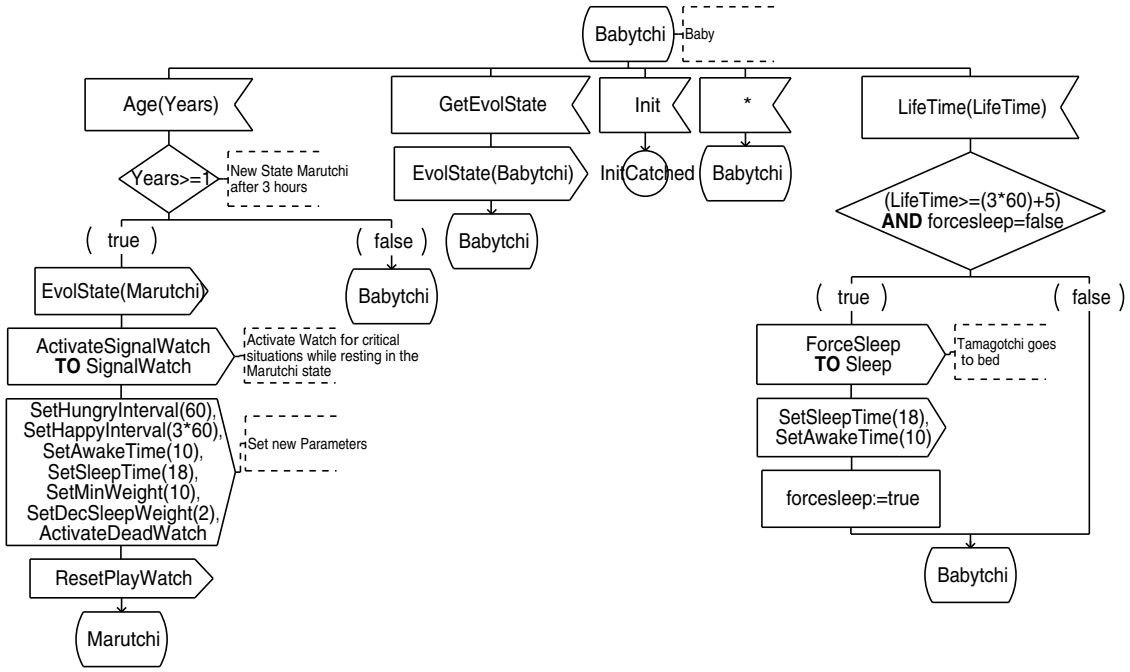

Figure 6: FSM of the component “Life Cycle” - State “Babytchi”

First the component “Controller” is informed about the new evolution state by the signal “EvolState(Marutchi)”. The different parameters of the new evolution state are set by sending out signals to the distributed “Life” components. In this way i.e. the signal “Set Happy Interval(3*60)” is sent to the “Happy” process. Consequently the decrement interval of happiness is changed to the value of 180 minutes.

After the new parameters are set, the new active state is “Marutchi”. This indicates the child phase of the creature. In this stage (Figure 7) an incoming “Age” signal with value greater 2 triggers a transition to the next evolution state. There are two possibile subsequent states. A “Marutchi” can become an angry or happy teenager.

To make a decision about the subsequent evolution state, the sub-procedure “Get Teenage Maintenance” is called. This sub-procedure requests several informations about the maintenance of the creature from the different “Life” components. The parameters for the following evolution stage are set.

Menu

The process “Menu” in the block “Controller” implements the user interface and display control of the system. Figure 8 shows the main state “Active Awake”. This state indicates that no menu item is selected and the creature is displayed. It handles all possible user inputs from the three buttons.

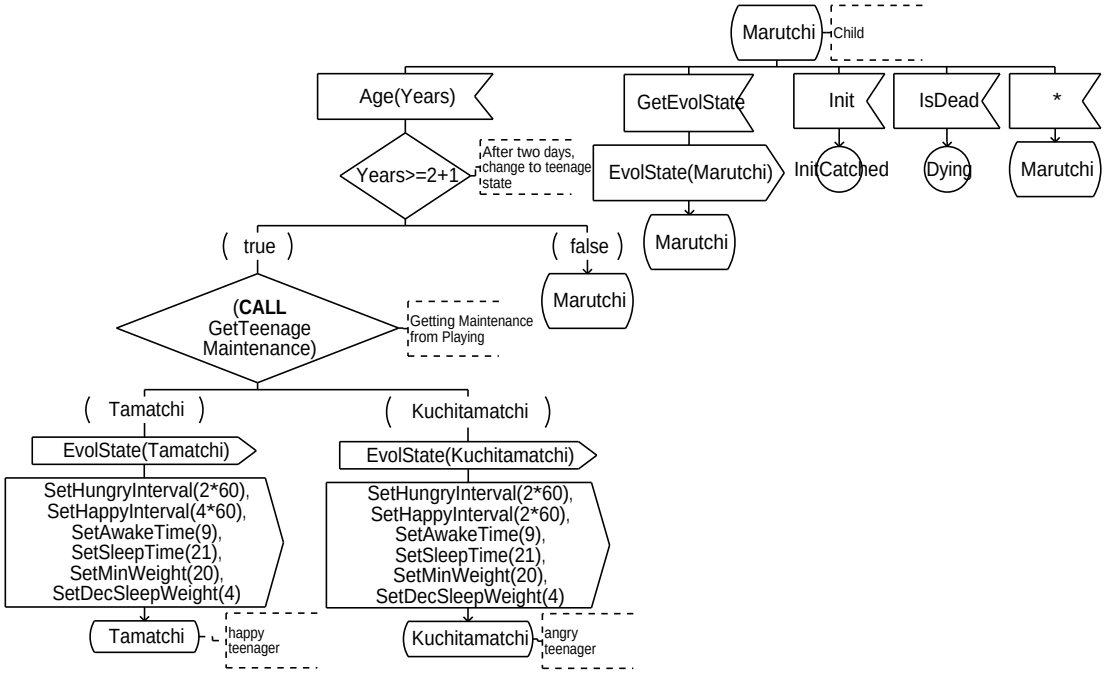

Figure 7: FSM of the component “Life Cycle” - State “Marutchi”

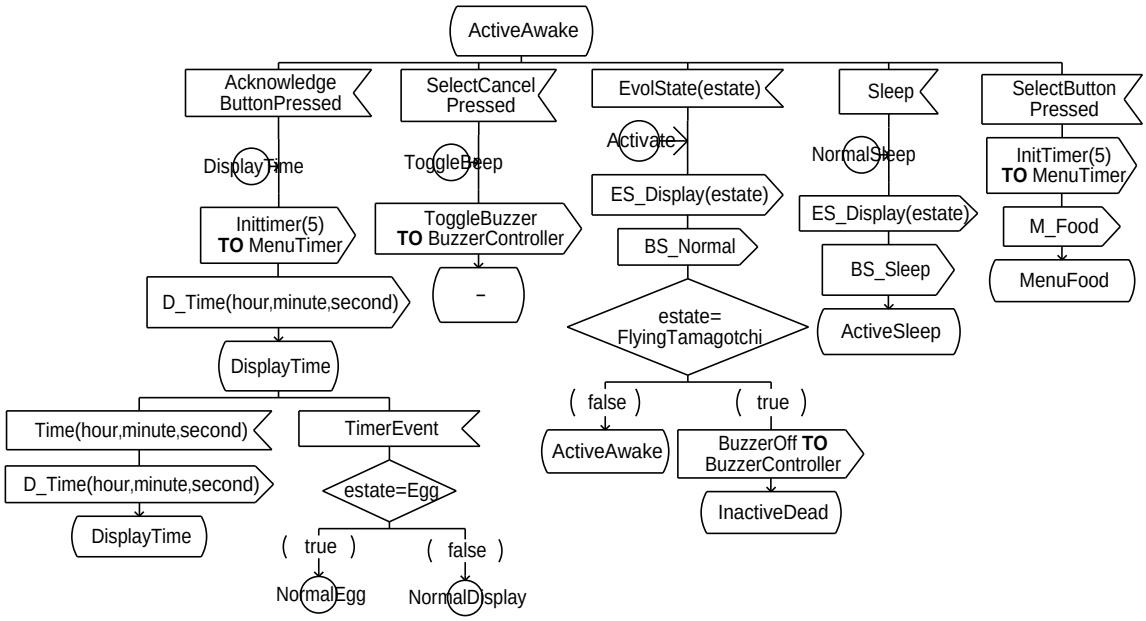

Figure 8: FSM of the component “Menu” - State “Active Awake”

If the user presses the “Acknowledge Button” the current time is displayed for 5 seconds. The display gets the signal “D_Time” containing the time in hours, minutes and seconds. To implement the 5 seconds time out, we use the external process “Menu Timer”.

The signal “Select Cancel” is generated when the user presses both buttons, select and cancel at the same time, and enables or disables the buzzer by sending the signal “Toggle Buzzer” to the “Buzzer Controller” process.

When the “Life” component changes the evolution state of the creature, the “Life Cycle” FSM sends a signal “Evol State” to the “Menu” process of the “Controller” block. This signal is caught by a transition starting at the “Active Awake” state to update the display. The display gets the signal “ES_Display(estate)” from the “Menu” to visualize the new stage. The signal “BS_Normal” indicates that the “Tamagotchi” is just awake. That means it is neither playing, eating nor sleeping.

If the new evolution stage is “Flying Tamagotchi”, our creature is dead. In this case we disable the menu by changing in the “Inactive Dead” state and the buzzer (signal “Buzzer Off” to “Buzzer Controller”).

“Life” can also send a “Sleep” signal to the “Controller” block, meaning that our creature falls asleep. Thereby the “Menu” process must update the display by sending the signals “Evol State(estate)” and “BS_Sleep”. If the creature is sleeping, the user menu is disabled by changing in the “Active Sleep” state.

The select button is used to select an item from the user menu. If this button is pressed in the “Active Awake” state, the first menu item “Food” is displayed. The signal “M_Food” is sent to the display and the “Menu Timer” is initialized with 5 seconds to allow a time controlled menu exit.

The new state is “Menu Food” (Figure 9). In this state the signal “Evol State” and the signal list “Sleep State” are not processed. To avoid the loss of these essential informations, these signals are saved in the signal queue of the “Menu” process to be treated in the “Active Awake” state.

The user can exit the menu by pressing the cancel button. In this case the “Menu Timer” is deactivated and the “Tamagotchi” is displayed in the current evolution stage. If the user does nothing for 5 seconds, the same procedure occurs initiated by a “Timer Event” signal .

Pressing the acknowledge button the sub-menu “Meal” is reached. On the other hand the user can press the select button to get from the “Food” menu item to the play selection.

Playing

The “Tamagotchi” has a game facility. The requirements for playing with the “Tamagotchi” are listed below:

BA-F-23 (a) You can play with a Tamagotchi. This is done by selecting the menu item “Play” in the main menu. (b) The display shows a playing creature in the current evolution stage. (c)

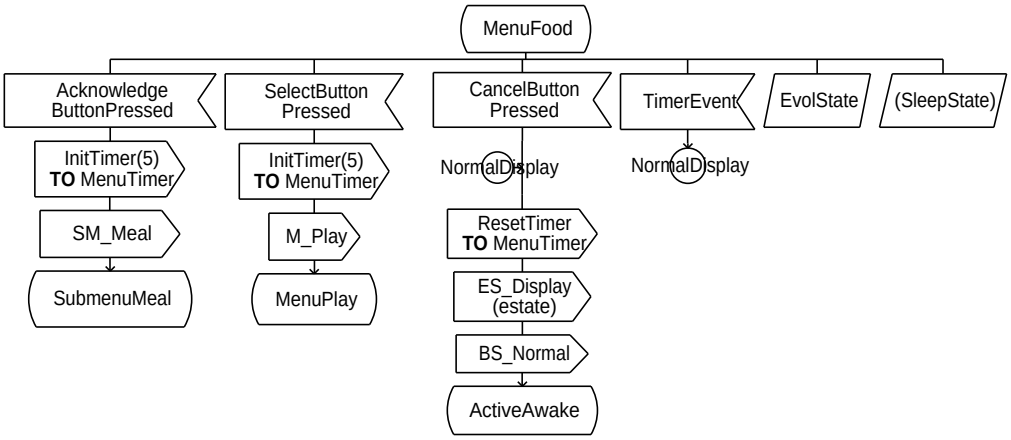

Figure 9: FSM of the component “Menu” - State “Menu Food”

The buzzer beeps frequently during the game. (d) The game only quits if the user presses the cancel button. (e) Without canceling the game, after each finished game a new one starts. The singe games can be won or lost. (f) A single game consists of 5 rounds.

BA-F-24 (a) The creature looks alternately to the left and to the right. (b) The user must choose one direction using the acknowledge or select button. (c) The “Tamagotchi” randomly chooses one direction. (d) If the user guessed the right one, the creature is laughing. Otherwise the “Tamagotchi” cries.

BA-F-25 (a) At the end of one game the score is displayed (number of right vs. number of false tips). (b) A game is won if at least three rounds are won. (c) The weight of the “Tamagotchi” decreases by 1 oz. per game. (d) If the game is won the happiness of the creature increases by 1 unit.

In our ObjectGEODE specification the game functionality is split into two main parts. The game parts associated with the life of the creature are implemented in the “Play” process of the “Life” component. Things dealing with the user interface and the display are part of the “Menu” process in the “Controller” block.

Playing and the Menu Process Figure 10 shows the menu state when the menu item “Play” is selected. In this state the user can start a new game by pressing the acknowledge button (BA-F-23a).

The transition calls the sub-procedure “Playing” of the “Menu” FSM. The return value of this procedure indicates whether a reset event caused by the reset stripe occurred during the game.

The Sub-Procedure Playing Figure 11 shows the start transition of the sub-procedure. First the signal “Start Game” is sent to the “Play” process of the “Life” block to indicate the start of a

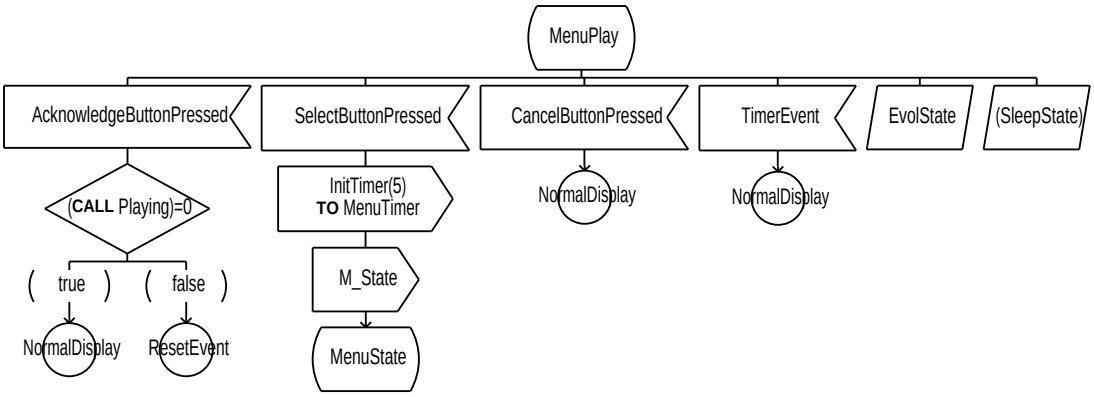

Figure 10: FSM of the component “Menu” - State “Menu Play”

new game. A one second time interval (“Init Timer” signal) is used to implement the alternating looking direction (BA-F-24a) and the frequent beeps (BA-F-23c). A playing “Tamagotchi” is shown at the display by sending the three signals “ES_Display”, “BS_Play” and “L_left” (BA-F-23b).

The local variable “lookdir” holds the current looking direction. This variable is of a boolean data type. The value false represents a left looking creature. Vice versa right corresponds to the value true.

After the initialization of the game, the system waits for user inputs in the “Play One Round” state (Figure 12). Every second a timer event triggers the change of the looking direction (BA-F-24a) and buzzer beeps (BA-F-23c).

Pressing the cancel button fires the transition to send out the “Cancel Game” signal from “Menu” to the “Play” process of the “Life” block. This signal indicates that the game is over. By returning an exit code 0 the sub-procedure jumps back to the “Active Awake” state of the “Menu” FSM (BA-F-23d).

The user can guess a direction by pressing either the acknowledge or select button (BA-F-24b). The decision whether the round is won or lost is made in the “Play” process of the “Life” component. The score of one round is randomly generated at the start of the round, meaning that the result is already fixed before the user chooses the direction. There is no difference whether the acknowledge or the select button is pressed.

After the user interaction, the “Menu” process is waiting for response from the “Play” process in the “Wait Round Result” state (Figure 13).

If you get the “Won” signal, the “P_Laugh Won” signal is sent to the display to show a laughing “Tamagotchi” (BA-F-24d) for 5 seconds. Waiting 5 seconds is done by the sub-procedure “Wait”. Then the start of a new round is initiated by a “Next Round” signal.

An incoming signal “Lost” is treated similarly to the “Won” signal. The difference is, that in this case a crying creature is displayed (BA-F-24d).

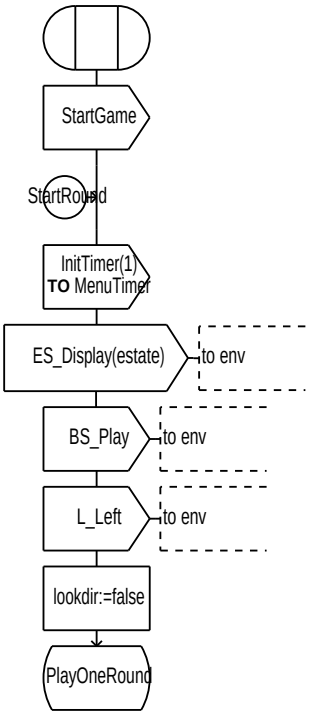

Figure 11: FSM of the component “Menu” - Sub-procedure “Playing” - Init

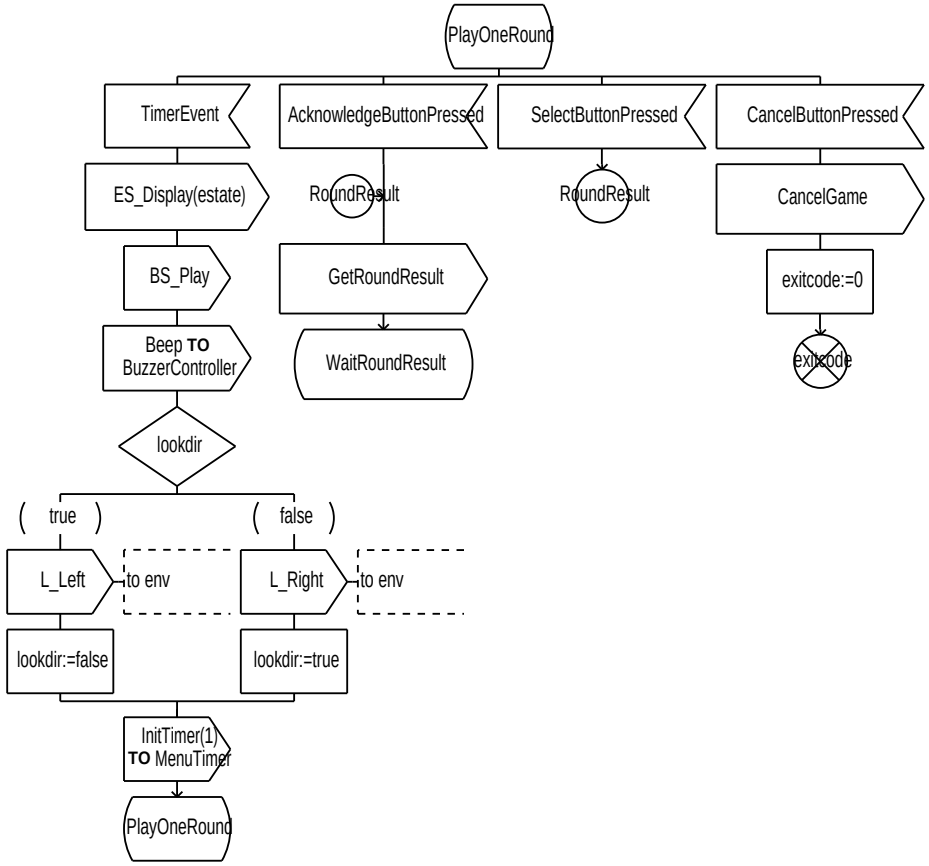

Figure 12: FSM of the component “Menu” - Sub-procedure “Playing” - State “Play one Round”

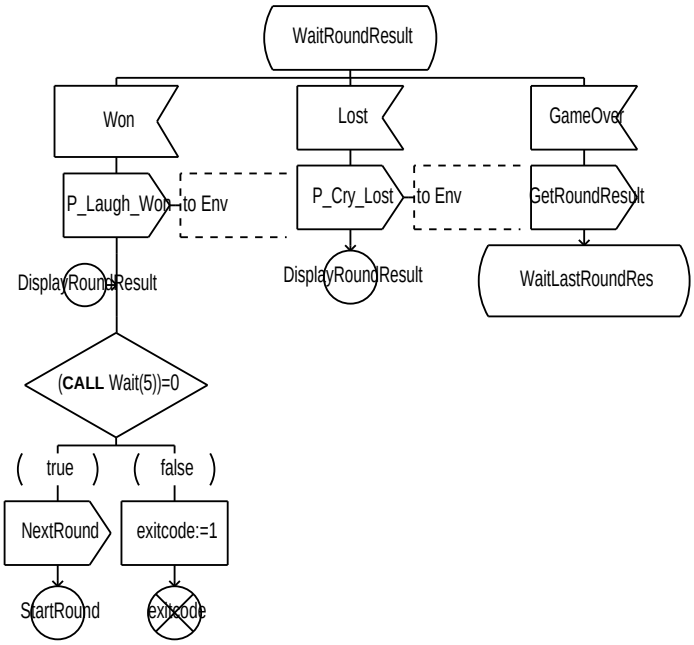

Figure 13: FSM of the component “Menu” - Sub-procedure “Playing” - State “Wait Round Result”

If the “Play” process sends the “Game Over” signal, a whole game consisting out of 5 rounds is over. In this case our “Playing” sub-procedure branches to the “Wait Last Round Res” state (Figure 14) and retrieves the last round result and the total game result.

After the “Won” or “Lost” signal from the “Play” process of the “Life” component arrives, a crying or laughing creature is displayed. Because the last round of the game is reached, the total game score is requested by the signal “Get Game Result”. The FSM waits for an answer in the subsequent state “Wait Game Result”. The following signal “Game Result” from the “Play” process contains the number of won game rounds. The number of lost rounds is computed and both numbers are sent to the display using the “P_GameRes” signal (BA-F-25a). The next game starts automatically by sending the “Start Game” signal to the “Play” process of the “Life” block (BA-F-23e).

The Play Process of the Life Component The “Play” process implements the core game functionality. At the initialization of this process (Figure 15), the two local variables “gamewon” and “roundwon” are set to the value false. The variable “gamewon” indicates whether a whole game has been won or lost. Similarly “roundwon” is responsible for the result of a single game round.

The variable “alwaysplayed” is used to watch the condition, that the user plays twice a day with the creature during the “Marutchi” stage. Depending on this variable the “Life Cycle” FSM

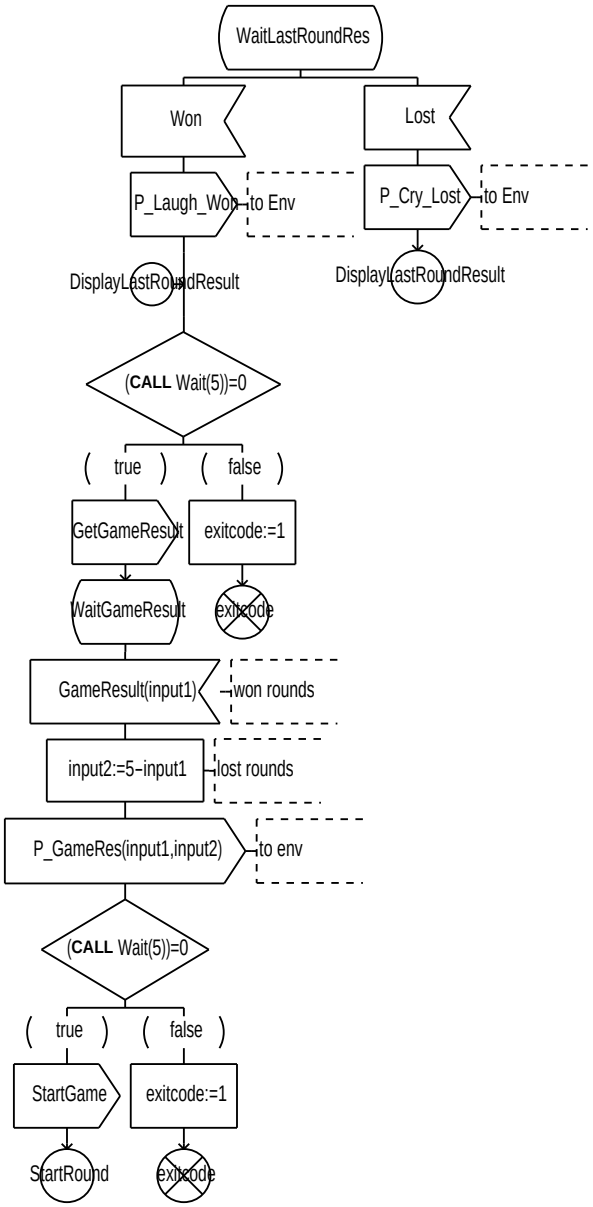

Figure 14: FSM of the component “Menu” - Sub-procedure “Playing” - State “Wait Last Round Res”

decides whether the creature becomes a good or bad teenager. Additionally “playedaday” counts the number of played games and is used in combination with the watch variable “alwaysplayed”.

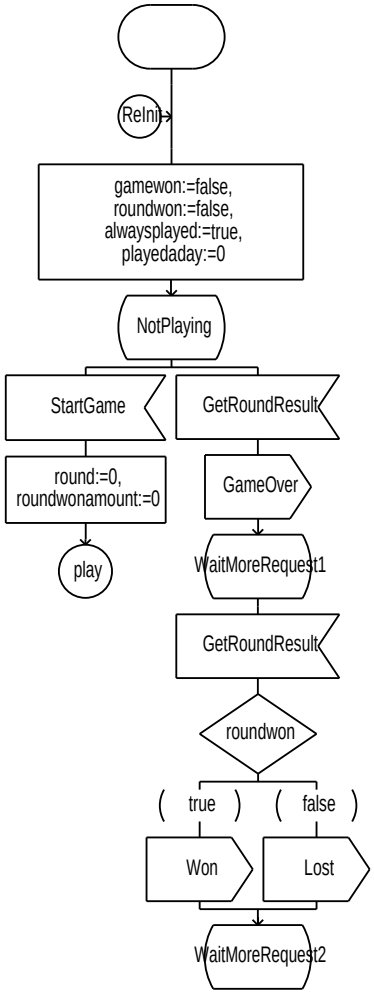

Figure 15: FSM of the component “Play” - Init and State “Not Playing”

After the initialization the FSM reaches the “Not Playing” state. In this state nothing is done until the “Start Game” signal is sent from the “Menu” process of the “Controller”. In this case the local variables “round” and “roundwonamount” are set to the value 0. The variable “round” counts the total number of rounds played so far in the current game. This variable is used to recognize the end of a game after 5 rounds (BA-F-23f). The local variable “roundwonamount” counts the total number of won rounds of a single game (BA-F-25b). The execution of the FSM is continued at the label “play”, within a transition starting at the “Playing” state. Figure 16 shows this state.

Every time a new game round starts, the “Menu” process sends the signal “Next Round” to the “Play” process. The round count variable “round” is incremented by the value 1 and the

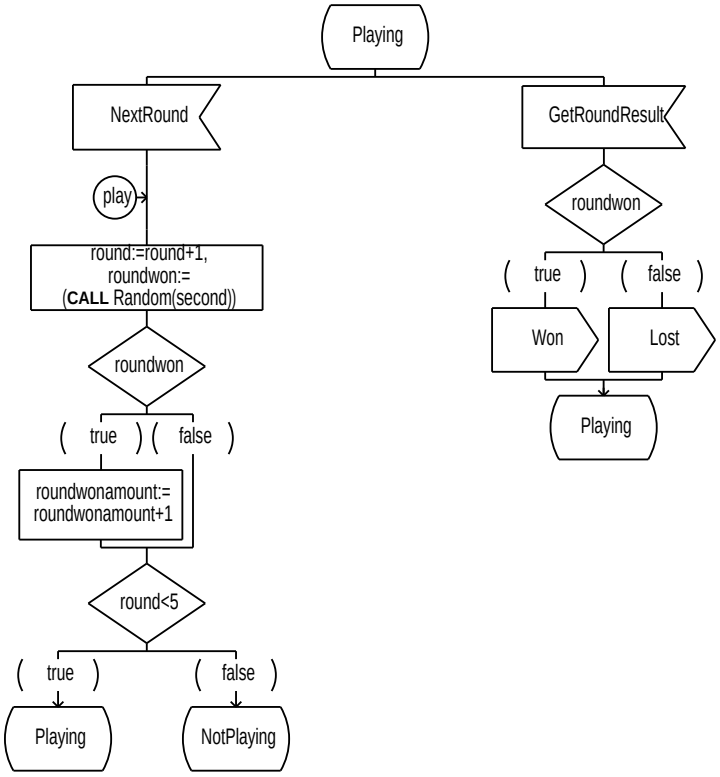

Figure 16: FSM of the component “Play” - State “Playing”

“Random” sub-procedure is called to generate a random round result (BA-F-24c). The round result is stored in the “roundwon” variable. If the round is won, the variable “roundwonamount” is increased by the value 1 to count the total number of won rounds (BA-F-24d). Finally the round counter variable is compared with the value 5 to decide whether the game is over (subsequent state “NotPlaying”) or the game is continued (subsequent state “Playing) (BA-F-23f).

After every game round the “Menu” process of the “Controller” component requests the last round result by sending the signal “Get Round Result”. The “Play” process returns “Won” or “Lost” depending on the last result stored in the “roundwon” variable.

Above we mentioned the sub-procedure “Random”, which delivers the round result of a single game round. This sub-procedure is illustrated in detail in Figure 17.

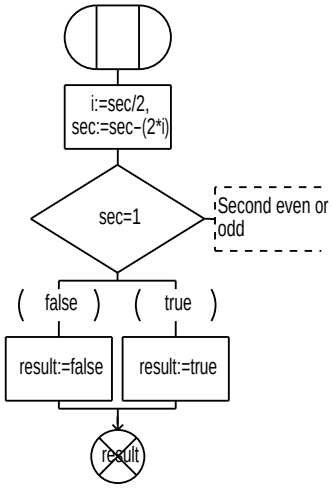

Figure 17: FSM of the component “Play” - Sub-procedure “Random”

The procedure takes the seconds of the current time as a parameter. The result is generated by the “Modulo 2” function. If the seconds value is odd, this round is won and the procedure returns the value true. An even seconds value results in a lost round with the return value false (BA-F-24c / BA-F-24d).

After the user has played a whole game, the “Play” FSM is already in the “Not Playing” state (Figure 15), now indicating that the current game is over. If the “Menu” process of the “Controller” block requests the round result by the signal “Get Round Result”, our “Play” FSM returns a “Game Over” signal to inform the “Controller” about the end of the current game (BA-F-23f).

Consequently after “Menu” receives the “Game Over” signal, it requests the round result a second time (FSM “Play” waits in the “Wait More Request 1” state). Depending on the last round result, a “Won” or “Lost” signal is sent to the “Menu” process.

In the next step the “Menu” requests the total game result from the “Play” process by sending the signal “Get Game Result”. Meanwhile the “Play” process waits in the “Wait More Request 2” state (Figure 18). After the game result request is received by “Play”, the signal “Game Result”

is sent back, containing the number of won rounds of the last game (BA-F-25a). If the number of won rounds is greater or equal to the value 3, the game is won (BA-F-25b) and the happiness of the creature is incremented by sending the signal “Inc Happy” to the “Happy” process of the “Life” block (BA-F-25d).

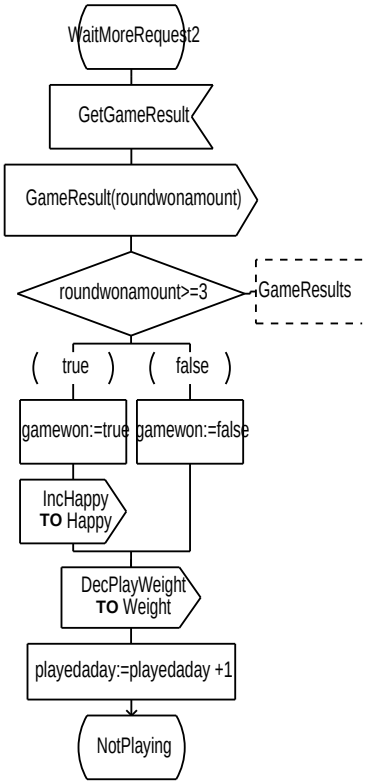

Figure 18: FSM of the component “Play” - State “Wait more Request 2”

The “Play” process decreases the weight by sending the “Dec Play Weight” signal to the Weight component (BA-F-25c). This decrease is independent of the game result. Finally the variable “playedaday” is incremented to watch the number of played games. The subsequent state is “Not Playing”. Here a new game can be started.

Figure 19 shows the asterisk state of the “Play” FSM. Transitions assigned to a asterisk state are valid in all states of the current FSM.

The “Menu” process can send the signal “Cancel Game” indicating that the user has pressed the cancel button to end the game. In this case the “Play” process jumps into the “Not Playing” state to end the game (BA-F-23d). If this event occurs, neither the weight of the creature is decreased nor the happiness is increased.

The transitions resulting from the input signals “End Day”, “Reset Play Watch” and “Get Play Watch” implements the play watch. This allows to validate the condition, that the user played twice a day during the “Marutchi” stage.

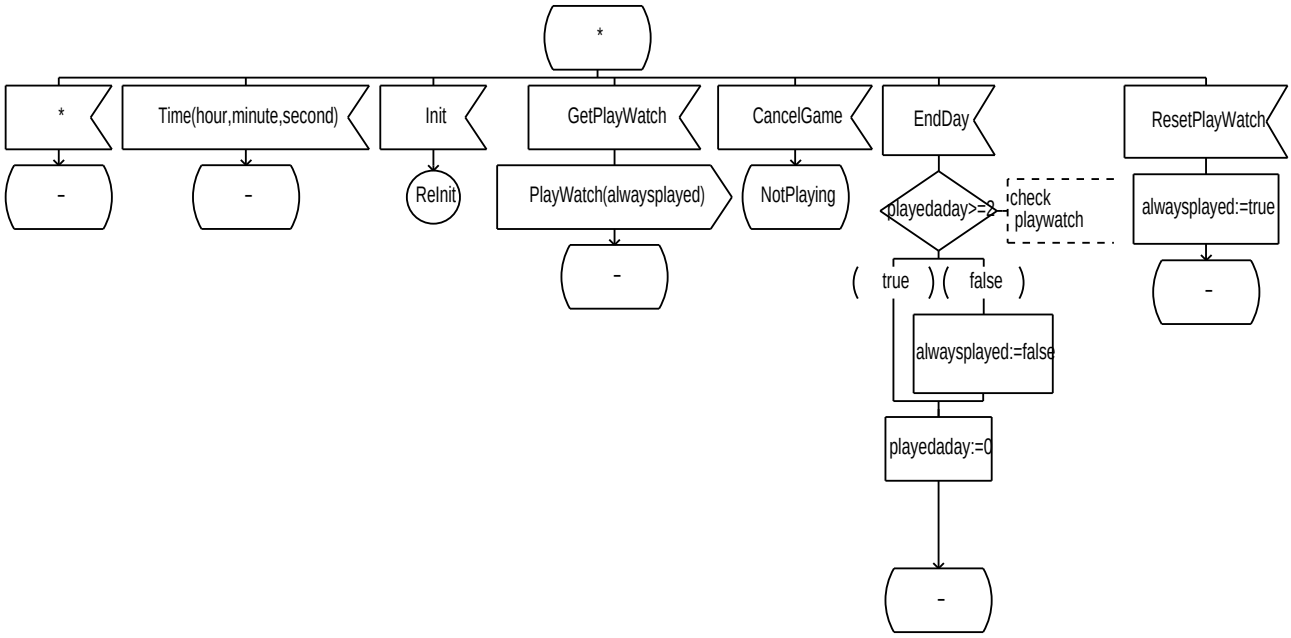

Figure 19: FSM of the component “Play” - Asterisk State

Sequence of signals Above the whole implementation of the play functionality has been illustrated. This has been done from the point of view of the single components. We can use MSCs to illustrate the sequence of the whole system. In this way we can show the temporal appearance of signals between the single components and the environment.

In Appendix A an MSC of a game is shown. This MSC indicates the signal sequence between the processes “Menu” of the “Controller” block and “Play” of the “Life” block. Furthermore all communication between this two processes and other processes of the system and its environment is shown. At the beginning of the MSC, the user menu item “Play” is already displayed. The user selects this menu item and starts playing. The MSC shows a whole game sequence, consisting of 5 play rounds. The user wins two and looses three rounds. After the game is over the score is displayed and a new game starts. Finally the user cancels the game.

1.4 Experiences

Training

The training in ObjectGEODE took 25 hours. In this phase we have read parts of the ObjectGEODE manuals [Ver98]. The main activity during the training phase was the specification of a simple “Safety-Injection” system. Real traps for beginners were the bad quality of the manuals and the strange user interface of the different graphical editors. On the other hand SDL is a simple

description technique. We did not experience any problems during learning this technique.

Modeling

The whole modeling process of the "Tamagotchi" system involved 70 hours of time. This contains the specification of the system in SDL, adapting the simulator to our needs and the simulation of the entire model. The architectural design of the structure took about 15 hours. 30 hours where required for describing the behavior of the single components in FSMs. We have needed the remaining 25 hours for simulating the specification and solving the found inconsistencies.

All time specific statements in this paragraph are related to one person. Normally these numbers should be multiplied by three. But teamwork was not possible because of the one-user license. We have built the specification all together at one single workstation. This way time amounts multiplied by three would not be representative.

Review

The review took part with the group who specified their system with the PEP tool. PEP is a tool prototype where the user can describe systems using high level Petri-nets. This notation is totally different from the finite state machines used in SDL. Therefore we had some problems in understanding the notation. A real problem is, that a system can not be split in single small independent parts. A system is always described as one large Petri-net. The only mechanism to cope with the complexity allows to hide single parts of the large Petri-net. But the implementation of this mechanism has still a lot of bugs.

In contrast to PEP, SDL allows the description of hierarchical structures. Therefore we think that it is more suitable for the specification of distributed and embedded systems.

Due to the vagueness, the traceability of PEP specifications is also difficult. There are a lot of 1:n relations at different places of the Petri-net. We also use many 1:n relations to implement the different requirements. But the strict separation between the single components and the possibility to generate MSCs makes it easier to find the different requirements in the SDL specification.

Like ObjectGEODE, the PEP tool offers a simulator to test the behavior of the specified system. PEP interprets the Petri-net and animates the execution directly in the editor. This is more convenient than the simulator of ObjectGEODE. ObjectGEODE uses compiled C code. Therefore it is not possible to edit the specification during a simulator session.

The PEP group did not find any real specification errors during the review. However, some differences in the requirement interpretation has been found.

1.5 Conclusions

As mentioned in Section 1.2, we could not work at the project together because of the one-user license. This means that a parallel modeling process could not be carried out. The strength of ObjectGEODE is the simulator, which is very useful to verify the specified system. It is possible to configure the simulator for special user needs. This means that you can hide irrelevant signals in a simulation scenario. The simulator offers an interface to other programs. Therefore it is possbible to adjust the simulation environment in respect of the users needs. Disadvantages of the tool are the bad documentation, the fussy editor framework and insufficient export functions for diagrams.

Positive features of SDL are the support of internal timers and signal buffers. Negative aspects of SDL are, that only point to point communication is allowed, that signals can not be queued once more in the priority queue after usage and the lack of hierarchical finite state machines. Another weakness of SDL is the huge space requirement of the notation.

SDL is well-suited for specifying the “Tamagotchi”. The developer gets a better understanding of the whole system. Even detailed aspects are precisely defined. In addition the ability of simulating the specification is a valuable help in finding out inconsistencies and faults in the requirement specification. The SDL specification describes almost all aspects of the later implementation. In this way an automatic code generator can take over a lot of implementation work.

Finally if you are searching a tool to create SDL diagrams, we would prefer a graphical editor like “Visio”. But the powerful simulator justifies the use of ObjectGEODE. Because of the possibility of generating code and simulating the specified embedded system a rapid prototyping process is possible. Because of these aspects and the practical oriented approach of SDL we would use the tool again for specifying embedded systems.

Appendix

A MSC

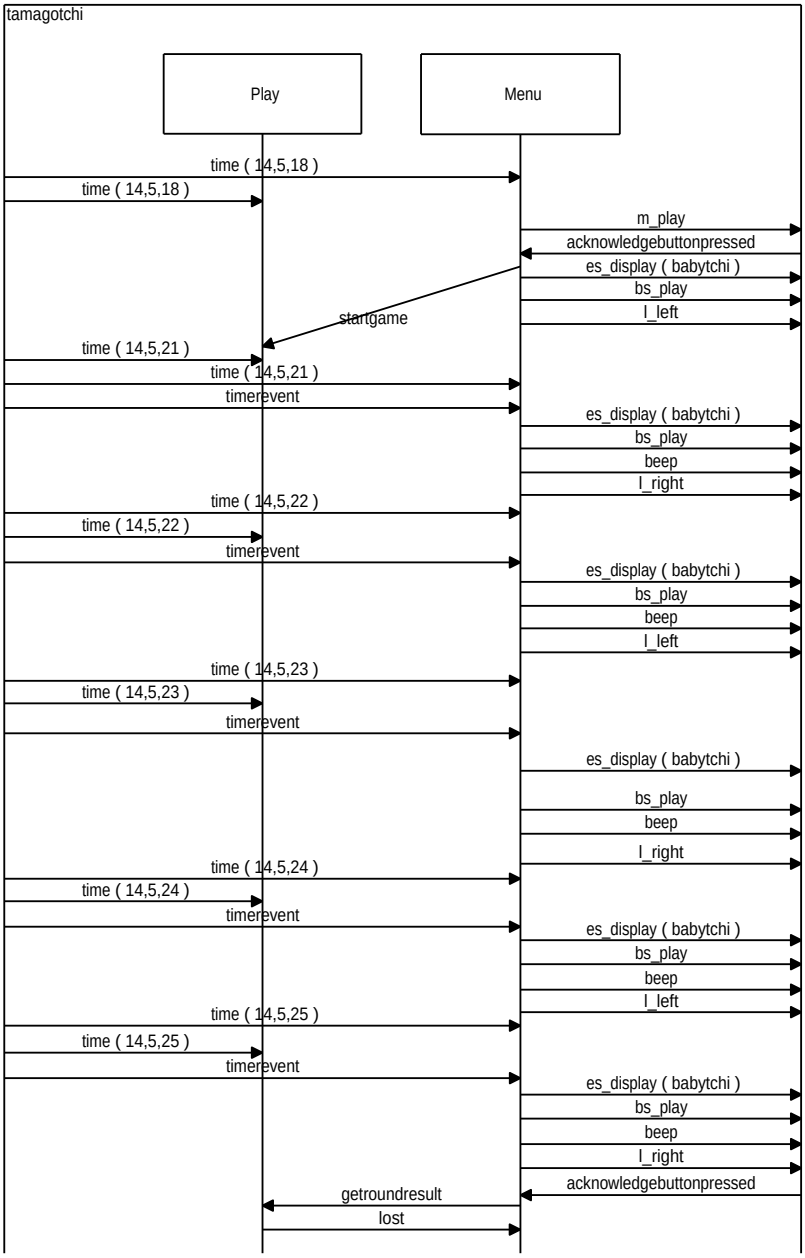

Figure 20: MSC “Playing” scenario - Part 1

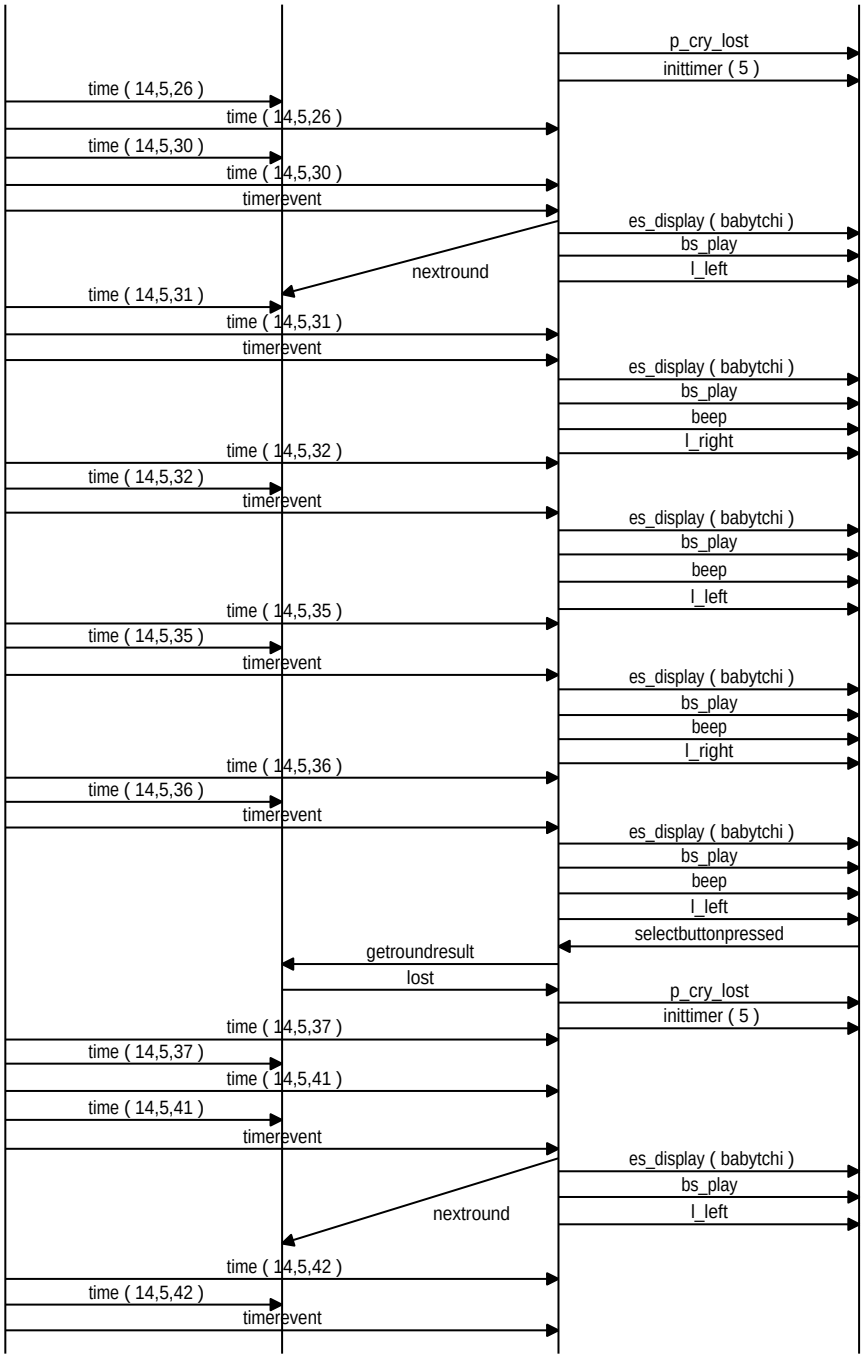

Figure 21: MSC “Playing” scenario - Part 2

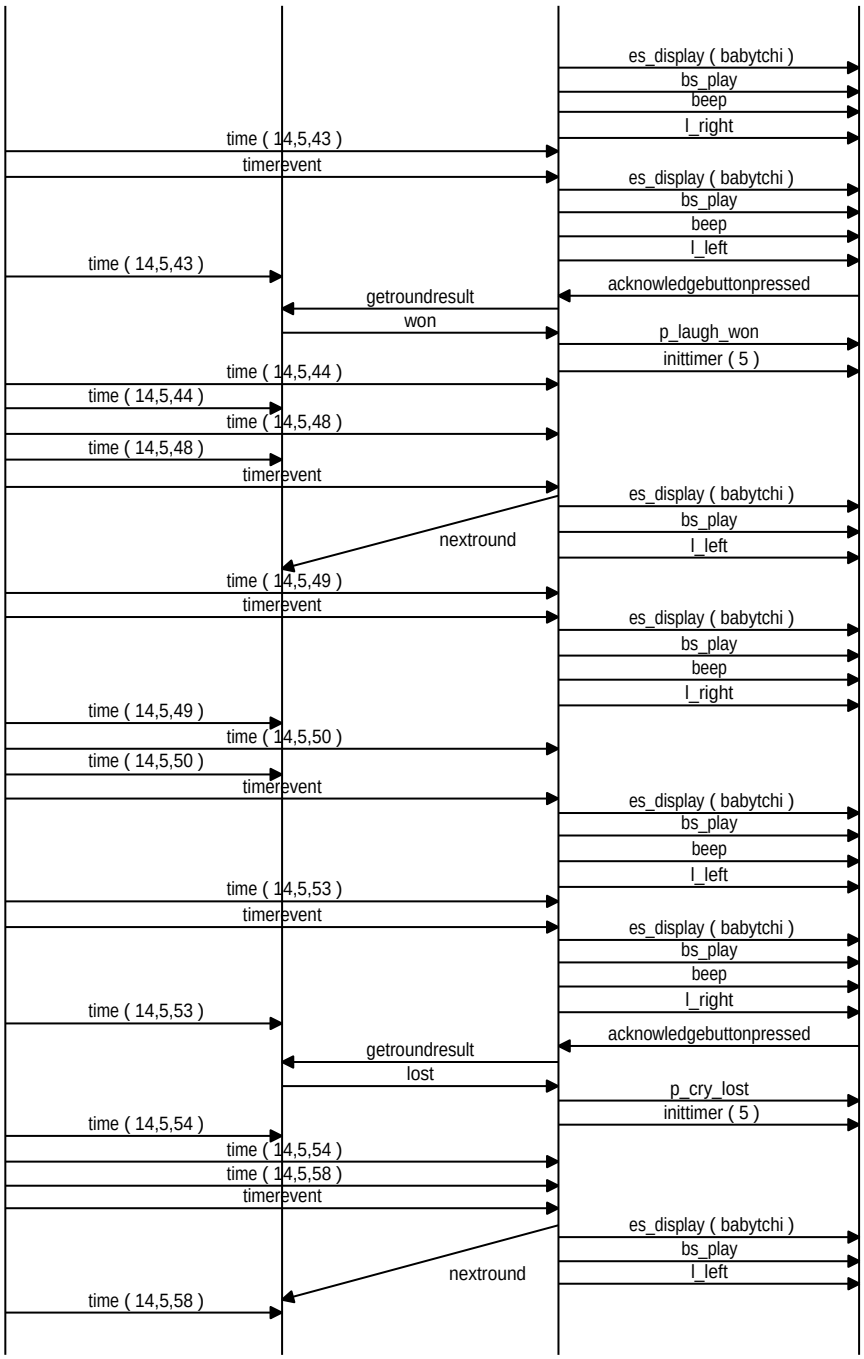

Figure 22: MSC “Playing” scenario - Part 3

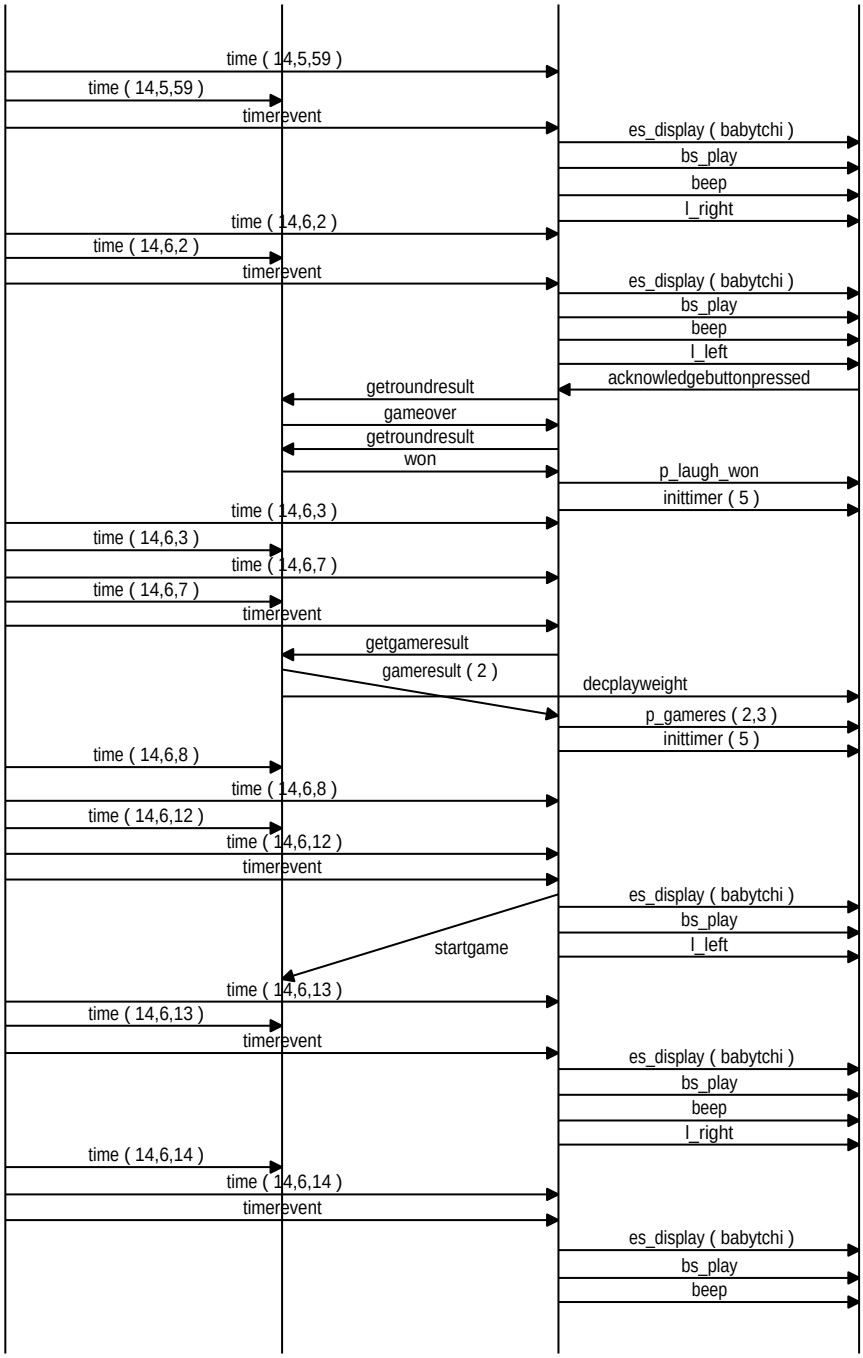

Figure 23: MSC “Playing” scenario - Part 4

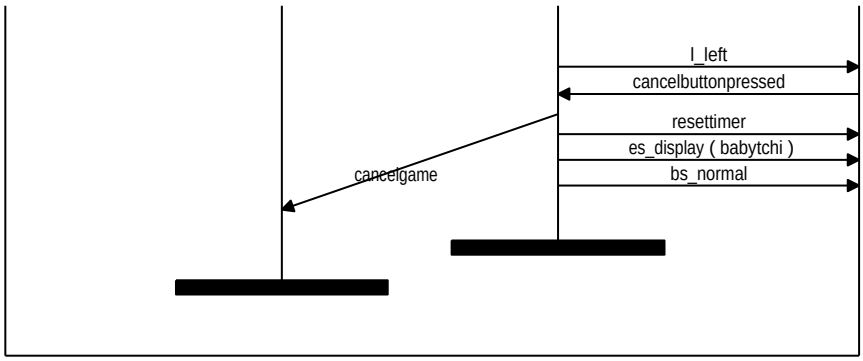

Figure 24: MSC “Playing” scenario - Part 5

References

[BH93] Rolv Braek and Oystein Haugen. *Engineering Real Time Systems*. Prentice Hall International, 1993. ISBN 0-13-034448-6.

[Soc98] SDL Forum Society. *SDL Forum Homepage*, december 1998. URL: HTTP://www.sdl-forum.org.

[Ver98] Verilog, France. *ObjectGEODE 1.2 Documentation*, 1998.