ROOM-Method and ObjecTime Tool

Authors: Markus Fröhler, Michael Gnatz, Marcus Lanzl

Table of Content

1. Introduction

1.1. *Purpose*

The ROOM-Method is supposed to cover all steps in the development of „Real-Time Distributed Systems“. Those systems ROOM may be applied to can be characterized by the following properties:

- Timeliness: minimized service time and system latency
- Dynamic internal structure: need of system reconfiguration due to changes in the external environment
- Reactiveness: continuous interaction with environment; system responses depend on input, system state and time
- Concurrency: several simultaneous threads of control interacting through process-communication and synchronization
- Distribution

1.2. *History and Authors*

ObjecTime was developed by the Telos Group at Bell-Northern Research. The Group was founded 1985 and aimed to investigate advanced system architectures for integrated voice and data telecommunications applications. The telecommunications sector is where ROOM comes from and is oriented to. From the beginning there was the goal of a persisting architectural model, which should evolve throughout the whole development life cycle. Another goal was to make the models executable as early as possible to enable developers to detect errors and omissions in early design stages, and also to test alternatives easily. Finally the automated transformation of the models to implementations on real-time operating systems was focused from the very beginning. The ObjecTime tool was ready for usage within Bell-Northern Research since January 1990. Two years later, an independent spin-off (ObjectTime Limited) was founded, which brought the tool to the open market.

The ROOM Method has its origin in the work of three authors: Bran Selic, Garth Gullekson and Paul T. Ward. It is based on practical experience of the authors in developing real-time systems in different industries. It was especially influenced by the design of a large distributed telecommunication system. Bran Selic is now vice president of Research and Development at ObjecTime Limited. He has experience in many application sectors from over 20 years of design and project management, including the fields of telecommunications, aerospace and robotics. Garth Gullekson is vice president of Marketing at ObjecTime Limited. He spent 14 years with the design and management of real-time telecommunications software at Bell-Northern Research. Paul T. Ward is principal of Paul Ward & Associates, a company providing training and consulting for real-time system developers.

2. Main features

- ROOM is inherently object oriented.
- The modeling concepts are specific to the real-time domain, which enables the construction of accurate and concise system models.
- ROOM provides for the explicit capture and documentation of system architectures.
- ROOM produces executable models at all levels of abstraction. This allows early detection of errors and omissions.
- ROOM supports an incremental and iterative development process. There are no paradigm shifts and discontinuities as in many traditional development models.

2.1. Notation

There are two major views of actors . They form the key items for modelling:

- Structure view
- Behavior view

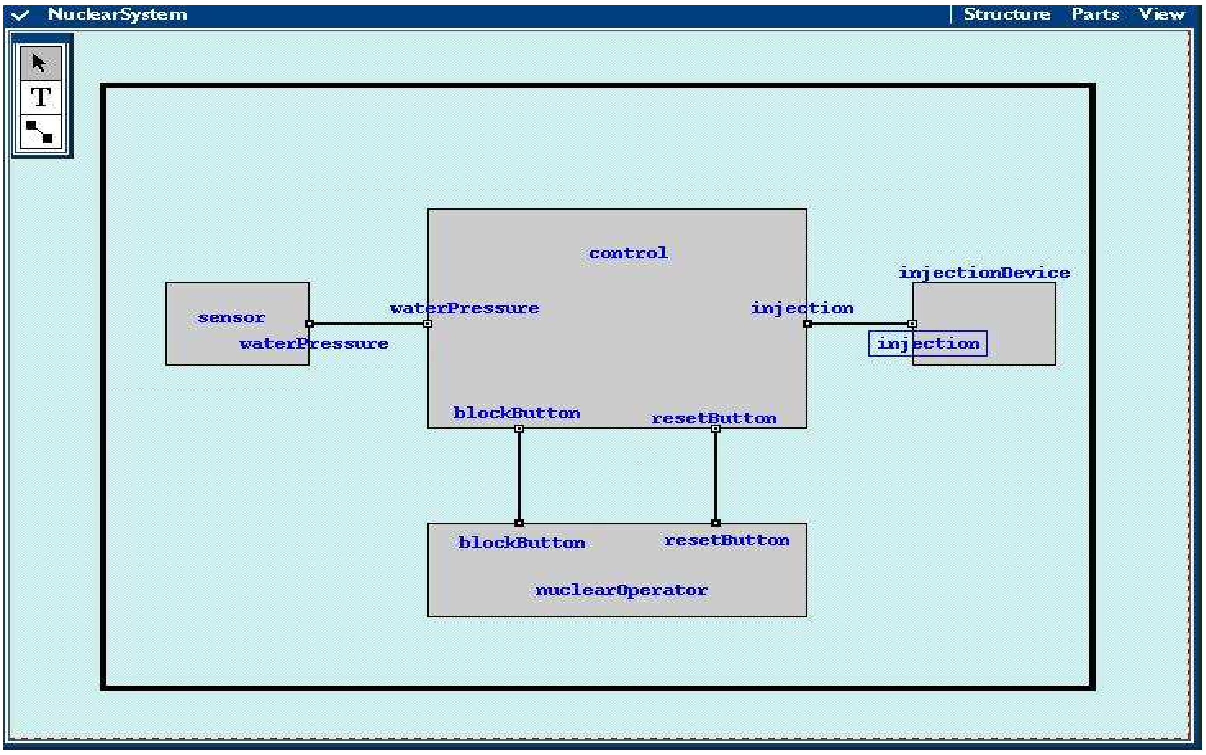


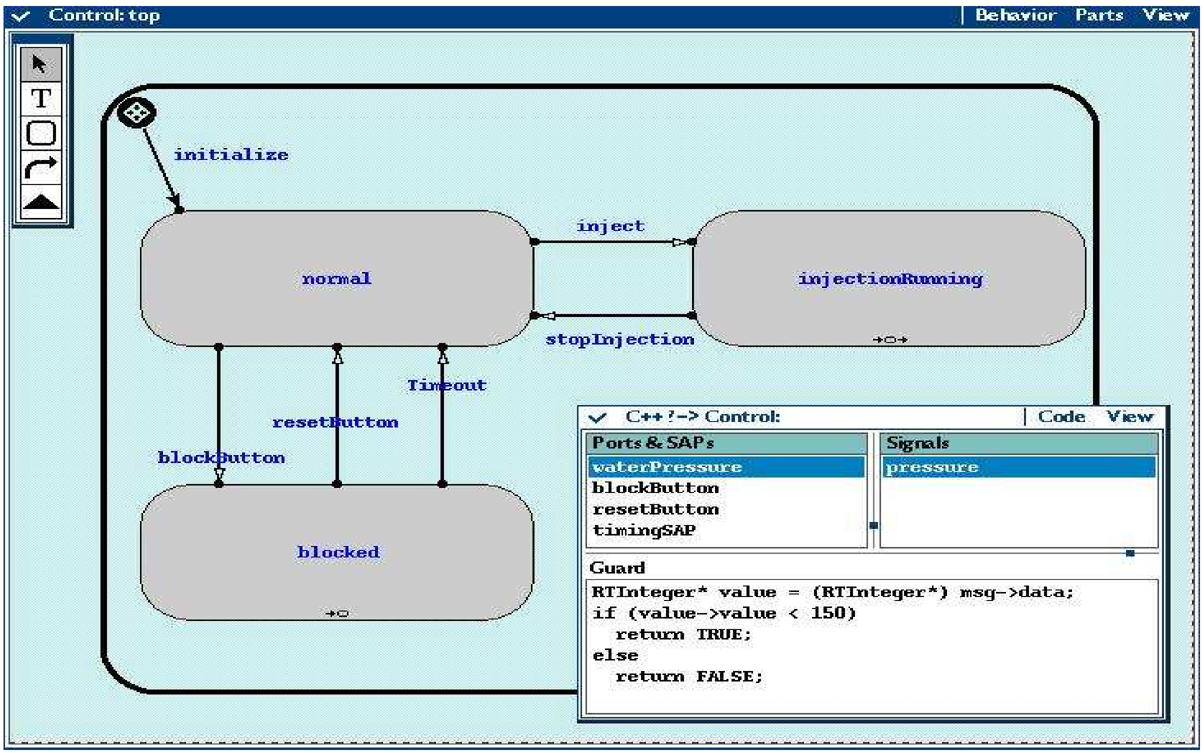
Figure 1: Structure-View of the reactor model

The structure view resembles a black box view. It describes communication with other components (actors) via ports, bindings and protocols. It also supports the encapsulation of actors within other actors.

Figure 1 shows the structure of the reactor case-study. The boxes represent actors, communication takes place via well defined channels (black lines). Each channel links two actors and has two different ends (ports). This allows to specify in- and outgoing messages within the attached protocol.

The behavior view can be compared to a glass box view. Behavior is represented by extended state machines (ROOM-charts). At any given time the actor is in one of a set of pre-defined states. Transitions between those states are triggered by incoming messages and may depend on specified logical conditions.

Figure 2 shows the behavior of the reactor case-study. The bubbles represent states, the arrows between them transitions. A transition is triggered, when one of the specified signals arrives and the (optional) guard condition equals to TRUE. Then the extended state machine changes into the state pointed to by the transition. Upon entry and exit of states there may be pieces of code executed (indicated by little arrows at the bottom of a state).



```
RTInteger* value = (RTInteger*) msg->data;
if (value->value < 150)
  return TRUE;
else
  return FALSE;
```

Figure 2: Behavior-View of the reactor model

States as well as actors can be nested, this allows different levels of detail. Valid messages are defined in protocol classes, they can carry along standard or self-defined data-types. Due to the object paradigm actors as well as protocols can be inherited and may be generated or deleted dynamically. Actors resemble independent distributed components and may therefore be executed independently.

The ROOM virtual machine – a runtime environment which hosts the simulation of models – provides additional services. Those, like the Timing Service for example, can be accessed via Service Access Points (SAPs) using the respective protocols. Message Sequence Charts can also be included into the model. They specify in detail the sequence of messages for concrete scenarios and are used by ROOM for model-validation.

2.2. Method

Generally, a modeling-advance from an abstract level towards more concrete levels is recommended. The ObjecTime Tutorial provides a list of steps that can be done repetitively. The first step resides on a very abstract level describing the system only roughly. Whereas following steps take place on ever more concrete levels resulting in a model as detailed as desired.

Those development steps are:

- Definition of actors
- Definition of protocol classes
- Declaration of ports and bindings between the actors
- Connection between bindings (communication channels) and protocols
- Description of behavior by ROOM-charts

The model is complete, as soon as it meets all user-requirements. This can be checked for example by a simulation of the whole model for specific scenarios.

2.3. Tool

The ObjecTime tool itself comprises the following basic components:

- **The Model Management** constitutes a library system that provides for formal storage of classes - including check-in, check-out and version control. Versioning and storage are provided by the configuration management of the main development environment.
- **The Model Editor** allows creation, modification and browsing of all elements of ROOM models. There may be Documentation objects associated with each element of the model. Graphical and textual representations can be exported for documentation purposes.
- **The Model Validator** is a powerful component which checks the model during construction, compilation and execution. For example, invalid bindings of actor ports and invalid actor containment relationships are prohibited while editing. Invalid message reception is revealed during simulation.

- **The Model Compiler** translates model into a high-level source code (C++ or RPL). This code can be compiled to run on the ROOM virtual machine.
- **The ObjecTime Run-time** system implements the ROOM virtual machine and provides capabilities for model execution and debugging. The testing of models is supported in various ways. So individual actors can be stopped and single stepped, messages can be injected or traced at any actor port. Trace or break points can be placed at structure and behavior charts. Furthermore there can be delays associated with event processing and message transmission, in order to simulate real conditions. The execution of a model can be observed in the same charts that have been edited using the model editor.
- **Cross Reference and Navigation** provide a good help, when browsing the model or looking for errors. There is an online cross-reference which enables keyword search within the whole model. Several navigators allow quick switching from one part to another, e.g. from a compilation error message to the place it occurred.

To add up to a complete development environment, ObjecTime can be used together with other software engineering tools providing:

- Requirements Specification Management
- Graphical Interface Development
- Project Management
- Document Creation
- Programming Language Environment

3. Approach and Result

We followed the method described in section 1.1.5. First we discussed the general design of the Tamagotchi model. We agreed on splitting the specifications into three independent actors. One should represent the Tamagotchi´s lifecycle, another should be responsible for keeping important variables (e.g. age, happiness or number of games played today) and the third actor would handle user interaction via the menu. We all worked together on one version of the model, since there were only two licences. Besides, it took almost the whole time for the project to understand some features of ObjecTime, including export for documentation purposes and the check-in/check-out mechanism.

We didn´t cover all of the specifications, but most of them. Display and Buzzer have not been modeled completely. ObjecTime does not support to check, whether two signals arrive at the same time. So BA-F-12 (turning the buzzer on and off) could not be modeled, because this would have required testing both keys (L and R) simultaneously. However we made great use of the ObjecTime Run-Time system. Simulation in early design stages discovered some errors and sometimes led to slightly new designs. Thanks to validation and simulation it didn´t take us long to understand what could be modeled and how it should be done.

Our model comprises 32 diagrams (structure and behavior views), but many of them are just empty actor frames. We declared 58 signals for the communication between the three main actors. For simulation the whole model was translated to 5.500 lines of C++ Code.

4. Specification of the Functionality „Playing“

4.1. *Structure and Behavior*

This chapter is about the feature of playing games with Tamagotchi. The specification of this functionality in ObjecTime is given here.

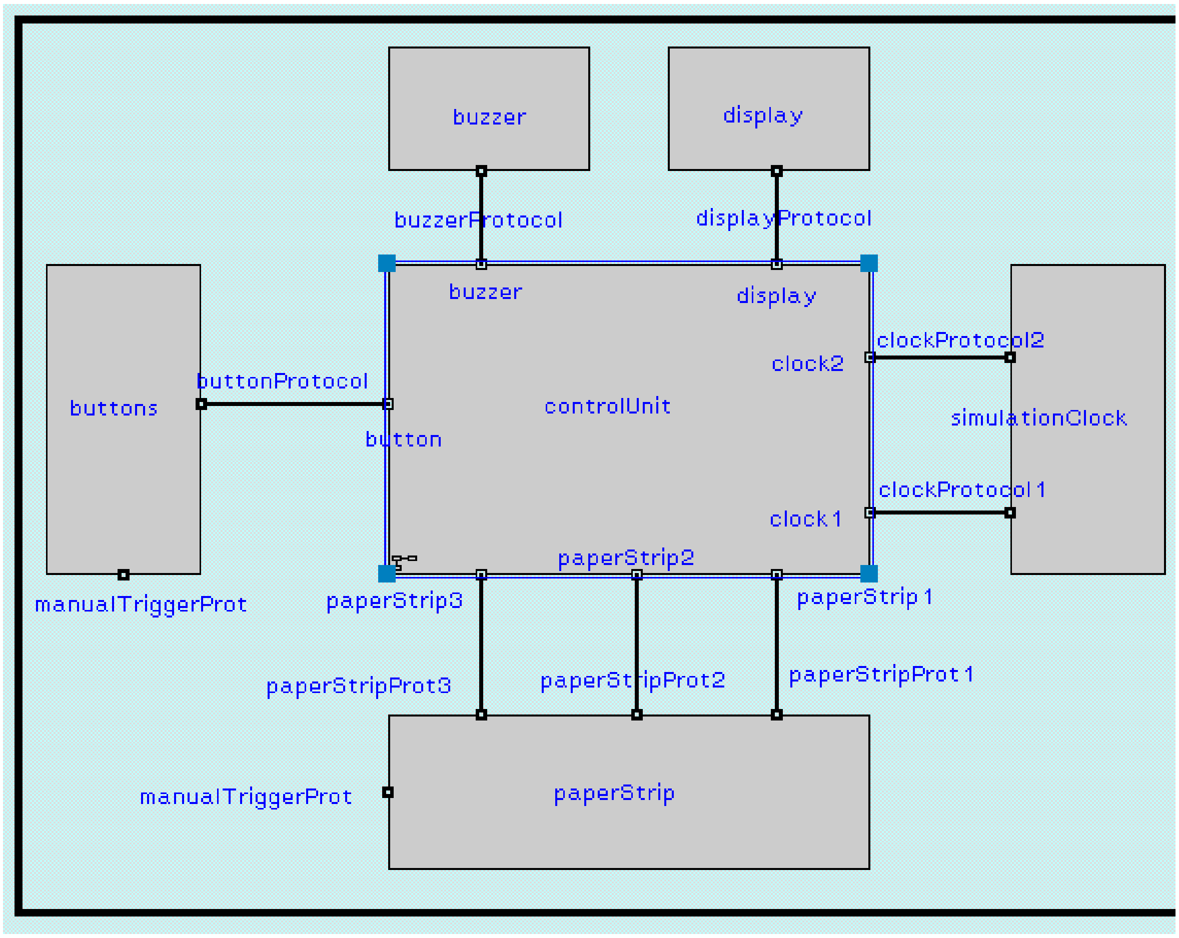


Figure 3: Structure of the whole system

Figure 3 gives an overview of the structure of the whole system. The six actors shown are paperStrip, simulationClock, display, buzzer, buttons and controlUnit, which is the main part of the specification.

The actor **simulationClock** is repeatedly sending signals to the **controlUnit** after variable time intervals. The controlUnit counts the number of received signals and thus knows the exact time. This actor is the only one, which is not directly related to the user requirements. By setting the time intervals in the **simulationClock** one can speed up or slow down the Tamagotchi's life during the simulation.

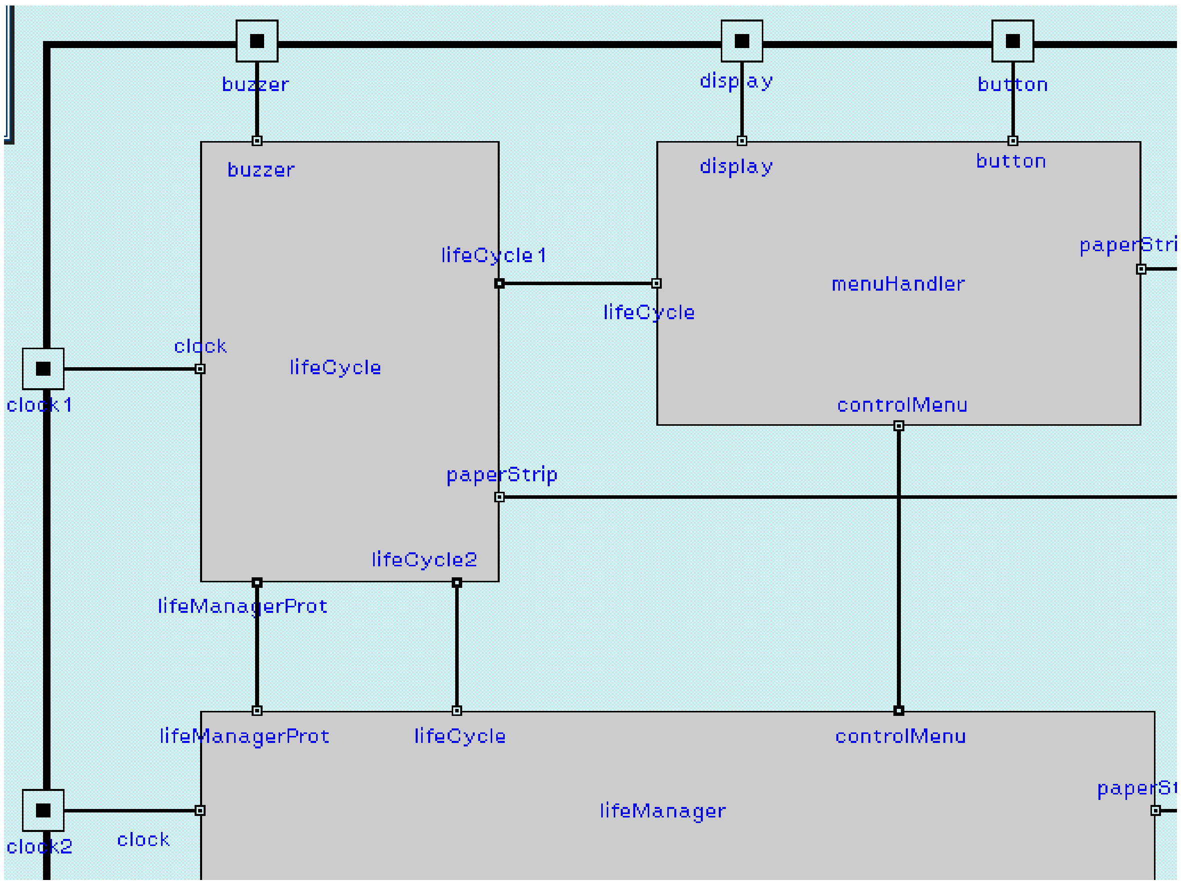


Figure 4: Structure of the actor controlUnit

Figure 4 shows the structure of the actor **controlUnit** - it contains three actors. The **lifeCycle** calculates Tamagotchi's development from egg, to Babytchi, Tamatchi or Kuchitamatchi, ... to state flying.

The **lifeManager** controls the states „*awake*" and „*asleep*". Furthermore it collects all variables from the different actors and also provides the actors with variables on demand, e.g. it provides **lifeCycle** with those variables needed to calculate Tamagotchi's further development.

The actor **menuHandler** receives the signals originating from the **buttons** actor in the previous figure. It controls the different menu states. It also contains the implementation of playing games. The relevant variables are sent to **lifeManager**, e.g. in case Tamagotchi's happiness has to increase after playing a game, the variable is transmitted to **lifeManager** over the controlMenu protocol.

The protocol connections to the higher-level structure as shown in the previous figure are displayed, too. The **simulationClock**, for example, is connected to both **lifeCycle** and **lifeManager** over the clock protocol.

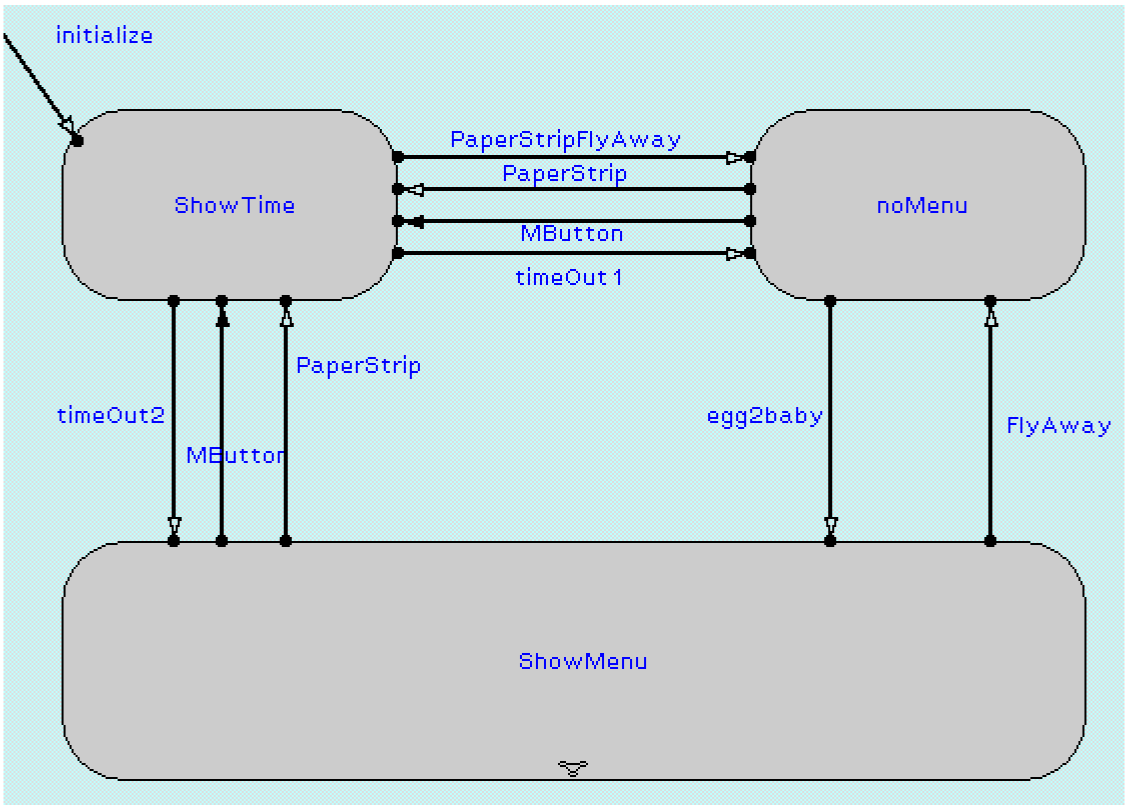


Figure 5: MenuHandler's behavior

Figure 5 shows the automaton which is associated with the actor **menuHandler**. The initial state is *ShowTime*. The transition PaperStripFlyAway is triggered by both the signal PaperStrip and the signal FlyAway and leads to state *noMenu*. PaperStrip signal originates at the actor **paperStrip** when the child is removing the paper strip. The FlyAway signal is sent by the **lifeCycle** in case Tamagotchi is dying.

After reaching the state *noMenu* for the first time **lifeManager**'s state is *egg*. In this state the menu functionality is disabled. Next **lifeCycle** sends the signal egg2baby indicating the development from state *egg* to *Babytchi* which leads **menuHandler** to state *ShowMenu*.

By pressing the middle button the Mbutton transitions originating at **buttons** leads back to state *ShowTime*. After 5 seconds either the timeOut1 transition in case the previous state was *noMenu* (which means the Tamagotchi is dead or still inside its egg) or the timeOut2 transition (while the Tamagotchi is living) is triggered, which leads back to state *ShowMenu*.

When the paper strip is inserted again the signal PaperStrip triggers the corresponding transition and **menuHandler** enters the state *ShowTime* which has also been its initial state.

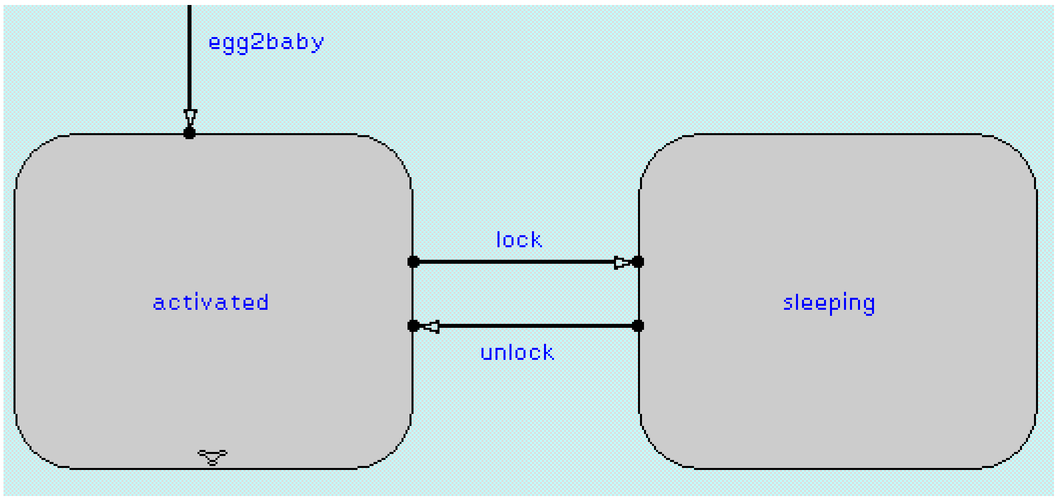


Figure 6: Behavior of state ShowMenu in actor MenuHandler

Figure 6 shows the behavior of state *ShowMenu* which is not shown in the previous figure. In this state the menu is either *activated* or inactive while the Tamagotchi is *sleeping*. The transitions lock and unlock are triggered by signals from **lifeManager**. The **lifeManager** sends these signals in correspondence to its own states *awake* and *asleep*.

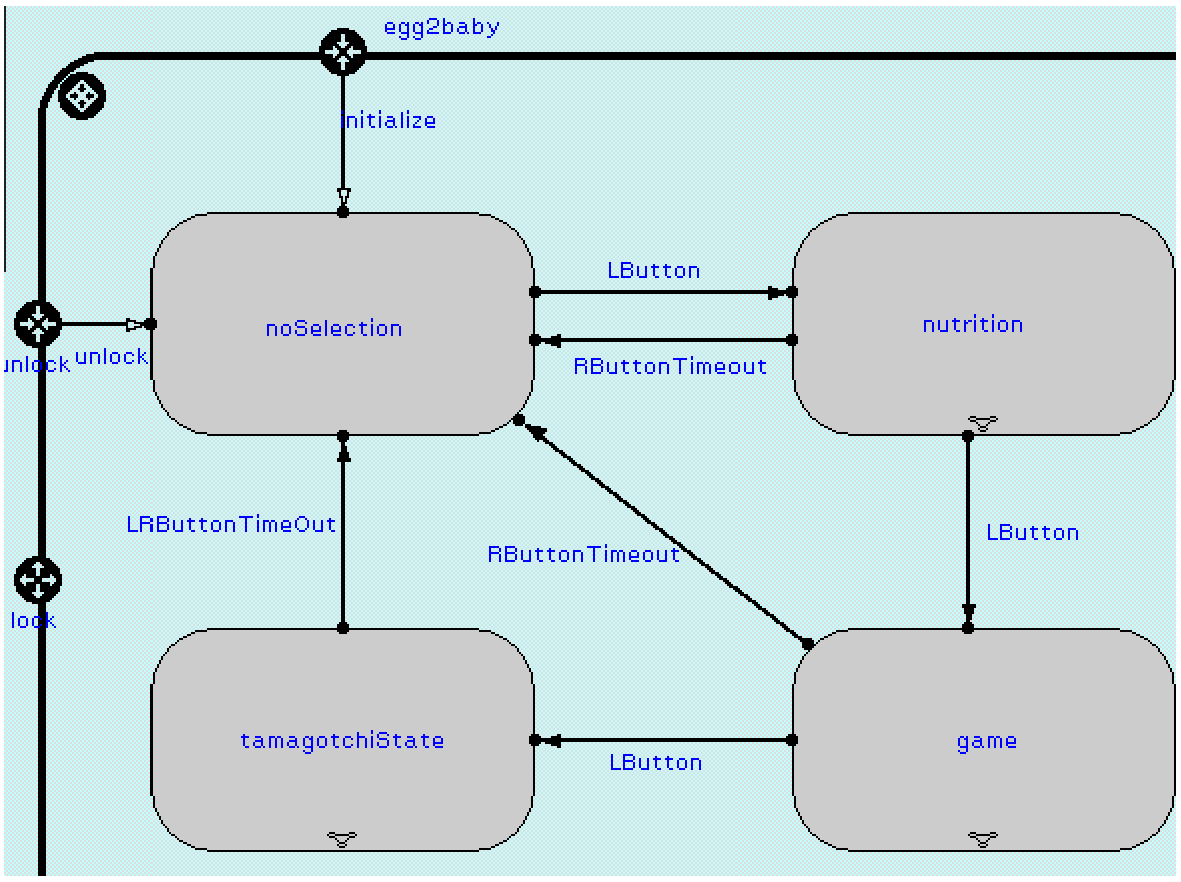
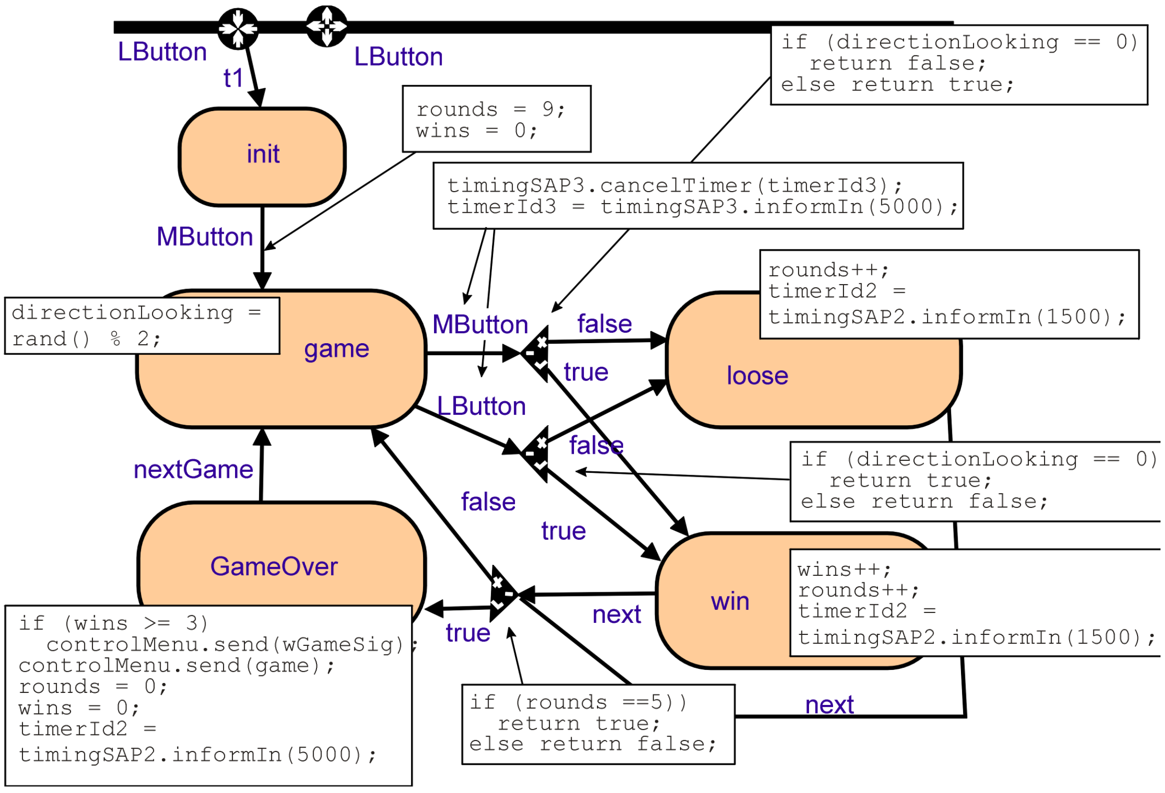


Figure 7: Behavior of state activated in actor MenuHandler

Figure 7 shows the behavior of actor **MenuHandler** in state *activated* which is hidden in Figure 6. The states *noSelection*, *nutrition*, *game*, *tamagotchiState* are the user's selection of a submenu, which either contains the functionality of feeding the Tamagotchi, playing a game with Tamagotchi or inspecting the Tamagotchi's state (or no selection). The transitions between the states are triggered by signals from actor **buttons**. Timeout signals are leading back in the initial state *noSelection*.



```
if (directionLooking == 0)
  return false;
else return true;
```

```
rounds = 9;
wins = 0;
```

```
timingSAP3.cancelTimer(timerId3);
timerId3 = timingSAP3.informIn(5000);
```

```
rounds++;
timerId2 =
timingSAP2.informIn(1500);
```

```
directionLooking =
rand() % 2;
```

```
if (directionLooking == 0)
  return true;
else return false;
```

```
wins++;
rounds++;
timerId2 =
timingSAP2.informIn(1500);
```

```
if (wins >= 3)
  controlMenu.send(wGameSig);
controlMenu.send(game);
rounds = 0;
wins = 0;
timerId2 =
timingSAP2.informIn(5000);
```

```
if (rounds ==5))
  return true;
else return false;
```

Figure 8: Behavior of state game in actor menuHandler

Figure 8 shows the behavior of state game which is hidden in Figure 7. The middle button is leading from state *init* to state *game*, where the game starts. The variables rounds and wins are initialised with zero and by entering state *game* the variable directionLooking is set randomly. Then the „player" chooses either the middle or left button and depending on the value of directionLooking this is leading to state *loose* or *win*. In both cases the variable rounds is increased, variable wins is just increased in state *win*. After waiting 1.5 seconds in state *loose* or *win* - Tamagotchi is happy or disappointed meanwhile - the transition next leads back to state *game* if rounds are less than five. Otherwise the state *GameOver* is reached. If wins is more than 3 the signal wGameSig containing the information that a successful game has been played is sent to actor **lifeManager**. After five seconds and initialisation of the variables transition nextGame is starting a new *game*.

4.2. Functional User-Requirements

In our implementation of the Tamagotchi the actors **display** and **buzzer** (see Figure 3) have not been realised. This is not because of problems with the ROOM modelling technique or ObjecTime but because of too little time for the project. Many signals from actor **menu** to actor **display** are necessary to enable the display functionality.

The user requirements BA-F-23, BA-F-24 and BA-F-25 - not including display and buzzer functionality - can easily be found in the behavior views (see Figure 5 to Figure 8) explained in the previous chapter.

The described behavior has been implemented straight forward in the ROOM modelling technique. Generally the behavior is implemented with extended finite automatons which are enriched with C++ statements. A balance between modelling with automatons and using C-statements must always be found and our implementation is just one possibility.

To „structure" the required behavior ROOM provides so-called structure views and actors. Different automatons in different actors can be integrated in one system by communication via signals. The user requirements BA-F-23, BA-F-24 and BA-F-25 are implemented in actor **menuHandler** and are integrated in the whole system via the protocols (communication paths) shown in Figure 4.

5. Experiences

This chapter deals with the experiences that we made during the seminar. It is divided into three sections. The first one shortly describes the training phase with the tool, the second one shows our experiences during the actual modeling of the Tamagotchi and the third one contains both information about the simulation (e.g. in how far it is comprehensible for an outstanding person what happens) and the review with the „Autofocus"-group.

5.1. The Training Phase

The first time we sat down together with our tool, it was our goal to go through the tutorial. Unfortunately, this turned out to be a pure operating instruction. It did not impart any comprehension for the operations which had to be executed. Additionally, both the enclosed manuals and the online-help already proved to be incomplete in this early phase. More details about this fact can be found in chapter 6.3. To a certain extent, this deficiency could be avoided by using [Sel94], which has already been mentioned earlier in this draft.

So, after spending one afternoon (which means about 10 hours, distributed on three shoulders) with getting used to the fundamental capabilities of ObjecTime, we spent three further sessions (approximately 35 hours, thereby altogether 45 hours) with the reactor case study. That way, we got familiar enough with the possibilities of the tool that we could enter the next phase: the modeling of the Tamagotchi.

5.2. The Modeling Phase

This was divided into three parts. First of all, the specification had to be turned into a rough design of the whole system on paper. Once again, it was the work of one afternoon (10 hours) and was followed by the implementation in ObjecTime. After some weeks (50 hours) in which we encountered a couple of problems, we had implemented the Tamagotchi so far that we could turn to the third part: testing and improving. This took us another 40 hours including the search for and removal of errors as well as the clarification of further ambiguities in the requests, which had not been discovered during the implementation phase. One extra session (again 10 hours) was invested in the review (more to it in 5.3), which does not directly count towards the modeling. Thus, the time spent on those phases adds up to approximately 110 hours.

As already described in the first chapter, the notation (ROOM) permits almost everything that was needed for the Tamagotchi and a lot more, which was not necessary in this case. The only major deficiency for our implementation was the absence of the simultaneousness of signals, which can – with a certain expenditure – be simulated by using the available means. Therefore, we did not transfer request BA-F-12, where the buttons „L" and „R" should be pressed at the same time to trigger an event. Possibilities to realize this functionality nevertheless would have been to either define an extra signal „buttons L and R pressed“ or to divide the signal for each button into two separate ones „button pressed“ and „button released“. This was the only exception, where we wished for an extra feature of the notation.

A weakness of the tool are some lacks in the implementation, which have no influence on the stability. But they contributed to the fact that some things did not work the way, we expected it. More detailed information about this - including some examples - is given in chapter 6.3.

Despite that, we could finish our implementation in time without having to solve any severe problems. The implementation went so far that a simulation of the whole Tamagotchi could take place. Additionally, we made a review with the „Autofocus"-group, mutually examining the results of the other group´s efforts.

5.3. The Simulation and Review Phase

On the one hand this part covers the simulation with the associated questions of comprehensibility for an external person, the possibility of understanding the transfer of the requests and the chances for testing and searching for errors (or at least inconsistencies). On the other hand it includes the results of the - previously mentioned - review.

The nice layout and presentation of both actors and machines (animation of switching transitions, marking of the active state, etc.) contribute to a considerable degree to the comprehensibility. Without being familiar with the implementation, one has the possibility to understand a lot of what is happening by viewing the simulation. A negative aspect on the other hand are the many code fragments, which remain hidden during simulation. Therefore it is hardly comprehensible in many cases, why certain transitions switch, where and why signals are sent and what causes a decision at a choice point.

The offered glossary word search makes a substantial contribution to the possibility of finding a transferred request. As result of a call of this function, ObjecTime presents a list of all occurences of the entered word within the implementation (code, signals, etc.). This can for example be used to find out, where and when a signal is sent or received. One further aid is the possibility of influencing the running simulation, using predefined signals.

Besides, the window concept of ObjecTime is not difficult to understand, windows are easy opened and closed with self-describing meanings. Unfortunately, the arrangement and number is not so nice. One large deficiency is however the very high nesting depth of actors and machines that can be reached (within the Tamagotchi there are already six levels), because the number of open windows increases with each level and thus also the confusion of the arrangement.

Finally, the question arises, whether it is possible to check the correctness of the implementation. In this context, the consistency checks of ObjecTime, which already go on during editing, turned out to be very helpful. They prevent the user from creating „wrong" connections (e.g. forbidden arrows in the machines, signal lines between two entries or exits of actors, etc.) and – in most cases – produce an error message, telling you quite precisely, why the connection is not allowed. Unfortunately, these checks only recognize errors that are not part of the code. If the code itself is wrong and the errors are found during compilation - which is done by an external compiler - there are two possible results. Either you receive a useful error message window from ObjecTime, including a list of all errors with links to the relevant places within the implementation. Or one receives an error message directly from the compiler, which usually doesn´t help much for debugging. Further problems emerge with growing size of the implementation. Although the Tamagotchi is not really extensive, there is no practicable way of testing its completeness. This is connected to the fact that a simulation of one complete lifecycle of the Tamagotchi is not possible within reasonable time.

Next topic is the review we made together with the „Autofocus"-group that used a tool, developed and realized at the chair of Professor Broy at the TUM. There haven't been found any modeling errors in our implementation, but as mentioned, we did not transfer all the requests. Besides the request, where we would have needed simultaneous signals, we did not implement the external devices display and buzzer. A further result of the review was that both tools are quite similar in many ways and that it is relatively easy to understand the implementation in one tool, when you are familiar with the other. Additionally we found out that our experiences with ObjecTime were a lot better than those of the others with Autofocus. Why and in which way is one of the subjects of the last chapter.

6. Conclusions

Like already chapter four, the last one is divided into three parts. The first and shortest one describes the possibilities for teamwork, the second one deals closer with weaknesses and strengths of notation, method and tool while the concluding one treats their usefulness for the modeling.

6.1. Teamwork

According to the manual, ObjecTime offers the possibility of integrating independently developed components into the main model. As it is quite difficult to make use of this functionality and with the chair having only two licenses for the tool, we decided to implement the Tamagotchi in one piece and to do it without any parallelism in the working routine. So most of our implementation was realized by all three of us working together.

6.2. Notation, Method and Tool

The first one can be treated in one sentence. It was self-describing to a great extent and after a short habituation time, easy to understand.

Substantially more complex is however the method (ROOM). As mentioned earlier it originates from the telecommunication sector and is thus particularly suitable for distributed systems, a field in which it reveals it’s strengths. One of these is the possibility of dividing one monolithic system into a number of subsystems (actors). Those communicate by exchanging signals - taken from developer-defined protocols - over predefined interfaces. Going along with that is another advantage: the consideration of temporal aspects. Therefore you can rely on the fact that the order of the signals sent over the same line is preserved.

Despite - or perhaps even because of – that the method was not ideal for the Tamagotchi. One reason for this is that a division into several actors would not have been necessary and was created rather artificially, as it contributed to keeping an overall view of the implementation. Another reason is the absence of global variables - in this case, variables which are declared once on top level and remain valid in all actors - that would have relieved us of a lot of work. To sum it up, the method offers far to much functionality for the Tamagotchi in most areas, while other features that would have been necessary are not part of it.

It is obvious that this also affects the evaluation of the tool. Principally, it offers everything that the method offers and displays its strengths in several areas. First of all, it preserves you from typing a lot of code by generating it, an aspect that should not be underestimated. In our case, ObjecTime made about 5.500 lines of code out of some hundred (and of course the actors, machines and protocols) which we wrote (and created). A further strength is the error recognition in the input phase. This was already mentioned in connection with transitions and signal lines in 5.3, but there are comparable checks regarding the code itself. It is for example not necessary to start the compilation in order to find out, whether all used variables have been declared somewhere. The tool already checks their existence when the relevant code is saved.

One of ObjecTime's greatest strengths remains however the visualization. Besides the already mentioned aspects (e.g. 5.2) it is displayed, whether a machine state has entry or exit code, the final points of signal lines are marked for incoming or outgoing signals and several more things. Other strengths and some weaknesses can be found in the concluding chapter.

6.3. Usefulness of Method and Tool

A lot of things that could have been part of this paragraph have already been mentioned and explained before. This fact shows that it is not easy to find a unique allocation and that this last chapter can be seen as a kind of summary, still including some new things.

Certainly, the stability of ObjecTime makes a substantial contribution to the usefulness of the tool. Although this should be a matter of course it is explicitly mentioned here, because most other tools had obvious backlog demand in this area. For good reasons, the intuitive structure of actors and machines as well as the communication by signals have been praised before. Without having much knowledge about method and tool, it is possible to understand what's going on and even use the basic functionality of ObjecTime. Just as important is in this context the timer concept which simplifies the creation and application of time signals a lot. Besides all that, the manual intervention into the simulation can be very helpful both for testing and demonstrating purposes.

But in most cases advantages are accompanied by disadvantages and thus ObjecTime has some weak points, too. The first one is the insufficient documentation, once mentioned at the beginning of chapter 5. A concrete example for this is our search for an explanation of the „Guard"-function, which was neither explained in one of the enclosed books nor could be found in the so-called „master index“. Both in that case and in some other situations, [Sel94] helped us to solve our problems. Some implementation lacks represented a further problem. One of them was the absence of the operators „<" and „>" in the class Real (real numbers). Additionally there occur naming conflicts between variables and signals, which are defined in different and from each other independent machines and protocols.

Another weakness is the copy&paste functionality of ObjecTime which only cuts out and reinserts everything that can be marked. So by using it, you lose all the code which belongs to the part you want to duplicate. Finally one must mention the number of windows which is constantly growing with the implementation getting more and more complex. Thus more than a dozen windows can belong to a simple machine, consisting of three states and just as much transitions (for exit and entry codes, code at the transitions, a variables window, etc.). In fact this brings the advantage that everything is clearly separated, but nevertheless one code window - with an appropriate subdivision - would have been completely sufficient for each machine.

Despite the discussed lacks, ObjecTime is a very useful tool and has obviously proved it's worth in practice. Even though the suitability for the Tamagotchi may be doubted for several of the reasons given above, we would use it again, for example if we had to model a distributed system considering temporal aspects. In this context, the strengths of ObjecTime would fully show to advantage and simplify the development. The second reason for working again with the ObjecTime is the fact that it turned out to be one of the best tools among those used at the seminar.

7. Literature

[Sel94] Bran Selic, Garth Gullekson, Paul T. Ward: Real time object oriented modeling, New York 1994