Seminar

Werkzeuggestützte Modellierung des Tamagotchi

Ralf Hettesheimer & Kizito Ssamula Mukasa (WS1998/99)
Faculty of Computer Science
University of Kaiserslautern – Germany

1.1 Introduction

1.1.1 The SCR Tool

We have used the Software Cost Reduction specification tool for modeling the Tamagotchi and the Safety Injection Device (SID). The Tool was developed at the Naval Research Laboratory of the US Navy. The purpose of this Tool is to assist software developers to specify software at low costs and without needing much knowledge of mathematical and theoretical modeling[1].

1.1.2 Notation

1.1.2.1 SCR Concepts

The notation as supported by the SCR Tool has two basic concepts. The first is the finite state machine concept. This explains the possible states the system can take. The second concept is of connecting the system with its environment. This is done through variables. We will go through these concepts in modeling the SID.

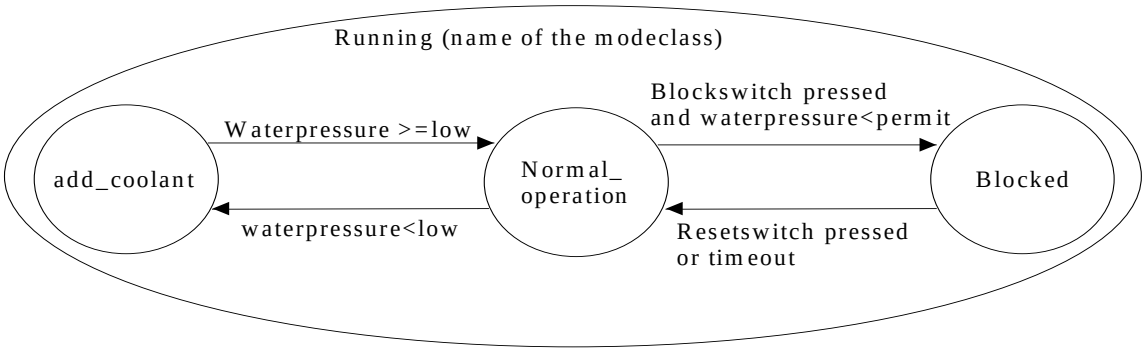

Fig.1 Our model of the safety injection device as a finite state machine

1.1.2.2 Defining Variables

As described above, we need variables to enable the system to communicate with its environment. The Tool knows three classes of variables. These three classes are: monitored, controlled and term. Monitored variables can be measured or monitored by the system, but it can not affect their values. A variable that a system can affect or control is a controlled

[1](SCR is) ... a formal method that software developers can apply without theorem proving skills, knowledge of temporal and higher order logic, or consultation with formal methods experts.

variable. Terms are functions of monitored variables modes and other terms. We will not need terms in modeling the SID.
We therefore need the following variables for our SID model:

Monitored variables: waterpressure, blockswitch, resetswitch, timer
Controlled variables: injection

Variables are defined in the "Variable dictionary" shown in the table below.

Table 1: Monitored Variable Dictionary

Name	Type	Initial Value	Accuracy	Comment
Blockswitch	Switch	off		
Resetswitch	Switch	off		
Timer	Time	0		
Waterpressure	Pressure	0		

Table 2: Controlled Variable Dictionary

Name	Type	Initial Value	Accuracy	Comment
Injection	Switch	off		

There is also a "Type dictionary" for defining or renaming variable types. We have defined three types in order to have a direct connection between the variable name and its meaning. The following table shows how this can be done.

Table 3: Type Dictionary

Name	Base Type	Units	Legal Values	Comment
Pressure	Float	bar	[0,100]	
Switch	Enumerated	N/A	on, off	
Time	Integer	min	[0,10]	

We also have two constants: low and permit. Their definition is done in the following "Constant dictionary table". The values of low and permit where set on the values shown, because the requirements did not contain specific values. The same happened to the legal values of the variable types in Table 3.

Table 4: Constant Dictionary

Name	Class	Type	Value	Comment
Low	Constant	Pressure	20	
Permit	Constant	Pressure	30	

1.1.2.3 The System States

According to our specification, the SID can take three different states: normal_operation, blocked and add_coolant. These states are defined using the "Mode class dictionary". These states belong to a mode class that we called “running”. The tool can handle different mode classes at the same time to model independent threads in the modeled system.
For our SID we need only one mode class. Its initial state is normal_operation. The table below shows the Mode class dictionary.

Table 5: Mode Class Dictionary

Name	Modes	Initial Mode	Table?	Comment
running	normal_operation, add_coolant, blocked	normal_operation	Yes	

After defining the states and mode classes, we have to define the events which cause the transition of states. An event is an instant in time in which a condition changes from true to false or vice versa. A condition is a predicate on the environment variables that holds for some period of time. Events and conditions are connected as follows:
@T(condition(s)) - an event indicating the change of condition(s) from false to true
@F(condition(s)) - an event indicating the change of condition(s) from true to false
@C(condition(s)) - an event indicating the change of condition(s)
@A - an event indicating the change of any monitored variable in the system

The following table contains the mode transitions for SID. It is the "Mode transition table" for our mode class “running”.

Table 6: Mode Transition Table for running

Source Mode	Events	Destination Mode
normal_operation	@T(waterpressure < low)	add_coolant
add_coolant	@T(waterpressure >= low)	normal_operation
normal_operation	@T(blockswitch = on and waterpressure < permit and waterpressure > low)	blocked
blocked	@T(resetswitch = on or timer > 9)	normal_operation

The most important part of the SID is setting the controlled variable “injection” at the right times to “on” and “off”. In our model this has to happen whenever the state add_coolant is entered or left. The following “Condition table dictionary” shows how the setting can be done.

Table 7: Condition Table for injection (Sheet of)

Name		Mode Class
Injection		running
Modes	**Conditions**	
add_coolant	TRUE	FALSE
normal_operation	FALSE	TRUE
Blocked	FALSE	TRUE
injection =	on	off

Whenever the add_coolant mode is entered, the injection goes to on. When any other mode is entered, the injection will go to off.

1.1.3 The tool

The Tool provides a checker which proves the consistency and completeness of the given specification. Syntax errors are localized and respective corrections are shown. Besides the syntax checker the tool provides a Dependency Graph Browser (DGB) which shows the dependency between variables. The arrows are from the dependent to the independent variable (see below). The tool also provides a simulator for a given specification. So the user can see how his model will operate.

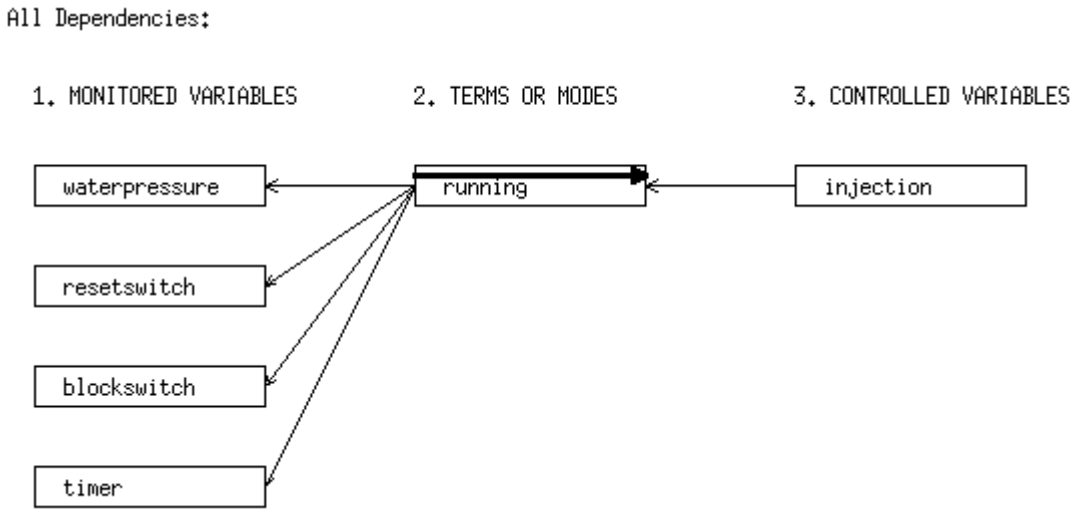

Fig.2 The dependencies in our SID model

1.2 Approach and Result

1.2.1 Modeling in team

To model the Tamagotchi in teamwork we decided to split it. Based on the informal requirements we discovered two big parts of the system. The first one, which we called backend, is responsible for all internal processing, i.e. the development of the Tamagotchi, timebased behavior and so on. The second part which we called front-end is handling all the requirements that involve user interaction, i.e. the main menu, playing, feeding etc.
We took this decision because it seemed to divide the work in two nearly equal big parts and the needed interface between the two submodels seemed very small. The two submodels were developed separately by us.

1.2.2 Support of the tool

The tool itself does not support any groupwork. So after the development of the two submodels we had to combine the two specifications. The tool does not even support importing a specification in an existing one. So we had to manually input the smaller specification in the bigger one. Based on the fact that the file-format used by the tool is plain ASCII, it could be possible to insert one specification in another one with a normal editor. But because we were not sure about the consistency of that method we did it by retyping.
Another problem in developing models separately concerns the “connecting” terms. These variables have to be terms in the finally connected specification. But in the development and testing of a submodel they need to be handled as monitored variables to simulate the interconnection because there is no support of the tool to manipulate terms in the simulator. So we had to develop the two submodels using monitored variables and then reimplement them as terms in the final specification.
Another problem concerns the flat modeling used by the tool. There is no capsulation of local variables in a modeclass. All the variables used in the specification are visible from every other modeclass. This can lead to overlapping definitions in separately developed specifications. We had to get around this problem by using a separate list of the variables used in the two submodels.
Using the discussed term/monitored variable substitution it was possible to test the two submodels using the simulator provided by the tool very early in the development process. The tool makes it possible to test the specification whenever there are no consistency problems. Situations prohibiting the simulation include inconsistent initial states, dependency graph cycles (see 1.4) or overlapping mode transitions.

1.2.3 Final specification

The final specification developed includes nearly all the informal requirements. We did not implement the 5 second timeouts for the menu via an automatically generated term. As described earlier the tool does not provide timers itself. It can use a monitored variable called time to handle time-based requirements. The unit we used on this variable is a minute. So it was not possible to model the second-based requirements. Instead we implemented a monitored variable that can be seen as the output of an external 5 second timer to control the timeouts.
Random based requirements, i.e. winning or losing a game, were modeled using monitored variables that can be seen as the outputs of external random generators.
The final specification does not contain a simulation frontend. The user or tester of the specification has to imagine the graphical output from the textual output via the controlled variables in the simulator.
The final model is consistent and complete in itself. This has been proved by the CC-Checker provided by the tool. A problem with the CC-Checker made it impossible to test the disjointness of the mode transitions from a certain size of the specification on. It seems that the Checker hangs in an endless loop while performing this special test. All other tests have been executed and resulted in a “no error” message.
There are no more errors in relation to the informal requirements as far as we know. The specification was reviewed against the informal requirements and no requirements were found that did not have an equivalent in the model. In the other direction nothing in the model does not correspond directly or indirectly to the informal requirements.
Because of the complexity of the modeled system and the non-interactive simulator (and the free time of the developers) not all allowed and possible situations in the life of a Tamagotchi

could be tested. The simulator needs an event-queue to be generated before it can simulate the model. The tool (simulator) does not support starting the model in a predefined state. It would be possible to change the initial state of the model to reflect the needed starting point but as the tool does not provide any help in doing that a lot of tables would have to be manipulated manually.
The model has been tested against the informal requirements and reworked until no more errors were found. We can therefore say that the final specification contains no errors.
There are two versions of the final specification relating to the two different solutions for the cycle-problem discussed in 1.4. The first one using the "userbased" solution consists of 71 tables. The second one using the "time displacement" solution consists of 66 tables.

1.3 Specification of Functionality "Playing"

Please note that we will be using the word "Tamagotchi" throughout this section to mean the artificial creature in the spefication we are modeling, regardless of its development stage.

1.3.1 Customer Requirements (CRs)

First of all we would like to give a short description of the CRs regarding playing with Tamagotchi.

CR-23:
The user selects "Playing" from the main menu and a playing Tamagotchi is seen on the display. A peep tone ist to be heard regularly. If the game is not aborted, another game follows. The game can be aborted by pressing the R-Button. Each game consist of five rounds. A game can be lost or won.
We have added sleeping time as another condition for breaking the game, so that Tamagotchi can automatically go to sleep when it's time without needing to break the game first.

CR-24:
In each round Tamagotchi looks left and right. Then the user chooses one of the directions and presses the L-Button for the left direction or the M-Button for the right direction. Thereafter chooses Tamagotchi one of the directions randomly. If it's the same as that of the user, a round is won, and a laughing Tamagotchi is seen on the display, otherwise a crying one.

CR-25:
At the end of each game, lost and won rounds are dispalyed. A game is won, if the number of rounds won is not less than three. Tamagotchi loses one unit of weight per game, and if the game is won, his happyness increases by one unit.

1.3.2 The Modes

In order to meet the CRs, we had to model Playing as a Mode of the Mode Class selectedAction (sa), which is responsible for the menu Functionality. Names of all modes which belong to this class have a prefix sa. The mode **saGames** corresponds to Playing in
CR-23. But with this mode alone, the L-Button could not have been used for playing, because pressing it could lead to a change to the next menu point "saInformations" (see CR-4). In this way the game could have been aborted against CR-23. We have solved this problem by introducing another mode called **saPlaying**. This mode is reached by pressing the M-Buton after choosing the saGames mode. Being in this Mode, playing can take place als required.
But this alone is not enough because according to CR-23, the next game has to follow after each game, if no abortion occurs. This means, that the variables used to count rounds and scores have to be reset. We need an expression of the form:

if(x=c) then x = 0,

where c is the value of any variable at the end of the game (after five rounds).
But the tool does not allow such expressions, i.e including the variable whose value is to be changed in the condition. This forced us to introduce another mode called **saReset**. We have made the change from **saPlaying** to **saReset** and vice versa as smooth as possible. That means after every five rounds the system changes automatically to the saReset and pressing the L or M-Button brings the user back to saPlaying. In this way the user does not lose his playing rhythm.

The table below shows mode transitons for the modes mentioned above. See 1.1.2.3 for the Notation used.

Part of the Mode Transition Table for selectedAction

Source Mode	Events	Destination Mode	Corresponding CR
saNoction	@T(m_lbutton = pressed)	saFoodAndSnacks	
saFoodAndSnacks	@T(m_lbutton = pressed)	saGames	
saGames	@T(m_rbutton = pressed)	saNoAction	CR-23
saGames	@T(t_awake = FALSE)	saSleep	CR-23
saGames	@T(m_mbutton = pressed)	saPlaying	mCR-23
saPlaying	@T((m_mbutton = pressed OR m_lbutton = pressed) AND (c_playedrounds < 5))	saPlaying	mCR-24
saPlaying	@T(m_rbutton = pressed)	saNoAction	CR-23
saPlaying	@T(t_awake = FALSE)	saSleep	CR-23
saGames	@T(m_lbutton = pressed)	saInformations	
saPlaying	@T(c_playedrounds = 5 AND (m_mbutton = pressed OR m_lbutton = pressed))	saReset	mCR-24
saReset	@T(m_lbutton = pressed OR m_mbutton = pressed)	saPlaying	mCR-24
saReset	@T(t_awake = FALSE)	saSleep	mCR-23

The CRs have the prefix “m” for “modified” bacause the Events do not correspond 100% to the original CRs. We had to modify them for the implementation

1.3.3 The Variables

Variables belog to the class monitored, controlled, or term. In order to identify them easily, we have prefixed their names as follows; monitored with m_, controllled with c_ and terms with t_. While monitored variables can and must be set by the user, controlled variables and terms are set internally by the system according to rules specified in the Event or Condition Tables of the correspondig variables. For this reason, the user must create either an Event or a Condition Table for each controlled variable or term used in the specification, but not both. We have used Event Tables only.

But first, lets see variables definition in the Variable Dictonary.

Table 9: Table for Term Dictionary

Name	Type	Initial Value	Accuracy	Comment
t_deltaHappyness	Integer	0		offset. set to 1 if the game is won and added to the total happyness
t_deltaWeight	Integer	0		offset. set to -1 after every game and added to the total weight

Table 10: Table for Controlled Variable Dictionary

Name	Type	Initial Value	Accuracy	Comment
c_lost	Integer	0		number of round lost
c_playedrounds	Integer	0		number of round already played per game
c_playbeep	Boolean	FALSE		peep tone during playing
c_scores	Integer	0		number of rounds won

Table 11: Table for Monitored Variable Dictionary

Name	Type	Initial Value	Accuracy	Comment
m_lbutton	button	notpressed		L-Button
m_mbutton	button	notpressed		M-Button
m_rbutton	button	notpressed		R-Button
m_tamdirection	direction	middle		for Tamagotchi random direction

Because the tool lacks a random generator, the user plays this role. He or she has to set the variable m_tamdirection to left or right after pressing the L or M-Button. He or she then has to set it to middle before the next round begins, otherwise no event will occur if the value of m_tamdirection is the same in consecutive rounds. That’s why m_tamdirection takes three values; left, middle and right

After defining variables, the next step is creating tables.
An Event Table consists of the name of the variable and the name of the Mode Class, in which the variable is used. That makes the first two rows. Then follows a two column row for the Modes (States) and Events. The "Modes" column contains rows for all modes, which belong to the mentioned Mode Class, one row for each group of modes with common events. Events are specified in the "Events" column. This can have more than one column. Lastly is the row with the primed variable name and the value it attains, when specified events occur. The prime on the variable name indicates the new value of the corresponding variable.
We will explain the Event Table for the variable c_playedrounds in details. We then hope that the reader will be in position to understand other tables.
The variable c_playedrounds counts the number of correct rounds played in one game. A round is correct and counted, if it does not violet order of playing specified in CR-24 i.e the user must choose the direction before Tamagotchi does the random selection.

To ensure that this is obeyed, we had to use an event of the form

@T(condition_2**) WHEN (**condition_1**)**

This event occurs, when condition_1 occurs before condition_2, and both must be changing from false to true.

Table 12: Event Table for c_playedrounds (CR-24)

Name	***Mode Class***		
c_playedrounds	SelectedAction		
Modes	***Events (CR-24)***		
saPlaying	**@T(m_tamdirection = left) WHEN (m_lbutton = pressed OR m_mbutton = pressed)**	**@T(m_tamdirection = right) WHEN (m_lbutton = pressed OR m_mbutton = pressed)**	NEVER
saReset	NEVER	NEVER	**@T(m_lbutton = pressed OR m_mbutton = pressed OR m_rbutton = pressed)**
saSleep,saNoAction, saGames,saFood, saSnacks, saFoodAndSnacks,	**NEVER**	NEVER	@A

saInformations, saHunger, saHappyness, saWeightAndAge, saTime			
c_playedrounds' =	c_playedrounds + 1	c_playedrounds + 1	0

In the table above, we see that the value of c_playedrounds is incremented, if the user presses either the L- or M-Button and Tamagotchi chooses one of the directions. This happens in the mode (state) saPlaying. In all other modes incrementation is NEVER allowed. In the saReset mode we are able to set the value of c_playedrounds to zero. The same applies, if any event occurs (@A) in any of the mentioned modes, but NEVER in saPlaying.

The following are Event Tables for other variables.

Table 13: Event Table for c_scores

Name		***Mode Class***	
c_scores		SelectedAction	
Modes	***Events*(CR-24)**		
saPlaying	@T(m_tamdirection = left) WHEN (m_lbutton = pressed)	@T(m_tamdirection = right) WHEN (m_mbutton = pressed)	NEVER
saReset	NEVER	NEVER	@T(m_mbutton = pressed OR m_lbutton = pressed OR m_rbutton = pressed)
saSleep,saNoAction, saGames,saFood, saSnacks, saFoodAndSnacks, saInformations, saHunger, saHappyness, saWeightAndAge, saTime	NEVER	NEVER	@A
c_scores' =	c_scores + 1	c_scores + 1	0

Table 14: Event Table for c_lost

Name		***Mode Class***
c_lost		SelectedAction
Modes	***Events*(CR-25)**	
saPlaying	@T(c_playedrounds = 5)	@F(c_playedrounds = 5)
saReset	NEVER	@T(m_mbutton = pressed OR m_lbutton = pressed

		OR m_rbutton = pressed)
saSleep,saNoAction, saGames,saFood, saSnacks, saFoodAndSnacks, saInformations, saHunger,saHappyness, saWeightAndAge	NEVER	@A
c_lost' =	c_playedrounds – c_scores	0

Table 15: Event Table for c_playbeep

Name		*Mode Class*
c_playbeep		selectedAction
Modes	***Events*(CR-23)**	
saPlaying	@T(m_tamdirection = left OR m_tamdirection = right)	@F(m_tamdirection = left OR m_tamdirection = right)
saSleep,saNoAction, saGames,saFood, saSnacks, saFoodAndSnacks, saInformations, saHunger,saHappyness, saWeightAndAge, saTime,saReset	NEVER	@A
c_playbeep' =	TRUE	FALSE

Table 16: Event Table for t_deltaHappyness

Name		*Mode Class*
t_deltaHappyness		selectedAction
Modes	***Events*(CR-25)**	
saPlaying	@T(c_scores>=3 AND c_playedrounds =5)	@T(c_scores < 3)
saSnacks	@T(m_mbutton = pressed)	NEVER
saSleep,saNoAction,saGames, saFood,saFoodAndSnacks, saInformations,saHunger, saHappyness, saWeightAndAge,saTime	NEVER	@A
t_deltaHappyness' =	1	0

Table 17: Event Table for t_deltaWeight(CR-25)

Name		*Mode Class*		
t_deltaWeight		selectedAction		
Modes	***Events***			
saPlaying	@T(c_playedrounds =5)	NEVER	NEVER	NEVER
saFood	NEVER	@T(	NEVER	NEVER

		m_mbutton = pressed)		
saSnacks	NEVER	NEVER	@T(m_mbutton =pressed)	NEVER
saSleep,saGames,saNoAction, saFoodAndSnacks, saInformations,saHunger,saTime, saHappyness,saWeightAndAge	NEVER	NEVER	NEVER	@A
t_deltaWeight' =	-1	1	2	0

We have been able to implement all Customer Requirements, except those, which the tool does not support, i.e displaying Tamagotchi, producing a beep tone and random selection of directions. Despite of these shortcomings, it is possible to play with Tamagotchi. We wish you will enjoy it!

1.4 Experiences

1.4.1 Getting used to the Tool and the Notation

In order to get used to the tool and notation, we started with modeling a small Safety Injection System (SID). We spent about 40 hours on modeling the SID. The only information source we had was the Tool Guide. This User Guide to SCR* Toolset consists of 77 pages. However, it does not mention some important features. For example the use of the function Duration() for monitoring time. This Term is generated automatically by the tool, if the user defines a monitored variable "time". We just discovered it accidentally because it appears in a screenshot in the User Guide! Also the Events @A and @C are only shortly mentioned. This means, that one has to use them for try and error in order to know how they are properly used. Moreover, the Guide doesn't clearly state the use of Terms and the fundamental difference between them and the controlled variables. These drawbacks are big problems for the beginner. Another problem is that there is no description of the example used in the User Guide. One sees only some of the tables, without knowing their background. This makes it difficult to understand them. Including the requirements to the specification could have provided the background of the system model in the examples.

1.4.2 Modeling Tamagotchi

Our approach to modeling Tamagotchi is explained in 1.2. It took some 100 hours. We didn't have problems with the notation, but with the tool. Some of the problems have already been mentioned, and others follow now.

☹ **The cycle problem**

The tool has a problem with cyclic dependencies between mode classes. The following diagram shows what this means. It shows an excerpt of our first Tamagotchi specification.

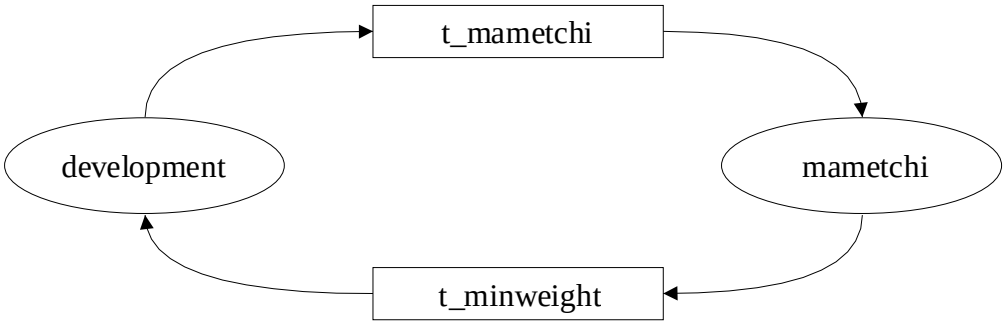

Fig.3 A dependency cycle

The arrows in the diagram show the direction of the dependency. I.e. t_minweight is set in the modeclass development and therefore depends on this modeclass. The modeclass mametchi is modeling the possibility of a Tamagotchi evolving into a mametchi. It has to check the actual weight of the Tamagotchi to t_minweight to fulfill CR-35 (The weight of the Tamagotchi must not fall below t_minweight to allow the development of a mametchi). So the modeclass mametchi depends on the Term t_minweight. In the mametchi modeclass the boolean variable t_mametchi is set to TRUE of FALSE and finally the development modeclass depends on this variable when it evolves the Tamagotchi. The dependency cycle is closed!
The problem with such a cycle is that the tool can not simulate a specification containing one or more of these cycles. The possible problem concerning dependency cycles is that specifications containing them could easily get into endless loops that the tools can not detect before execution.
Dependency cycles do not need to exist between two modeclasses directly. The cycle can include a lot of modeclasses which means the cycle is not always as obvious as in the above example. In our first approaches we included a lot of such cycles.
The tool developers know of the problem and say they are working on a solution. But until now no such solution is implemented in the tool. So it is up to the user to fix the problem. We discovered two possible solutions that both have their good and bad sides.

Solution 1: User interaction

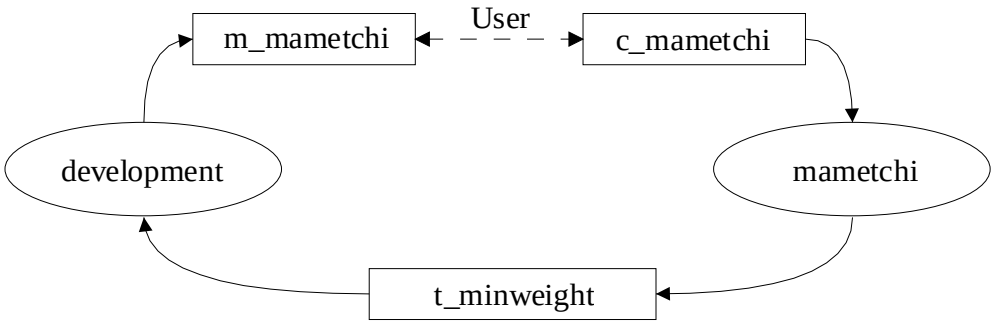

Fig.4 Opening the dependency cycle via user interaction

While simulating a specification based on this solution the user has to manually adjust the monitored variables, i.e. m_mametchi to the values presented to him in the controlled variables, i.e. c_mametchi. This makes simulation very difficult (some observers said nearly impossible) because the user has to look at a lot of variables at the same time and adjust the

monitored variables on any change. But it provides the possibility to react on any event immediately.

Solution 2: "Time displacement" solution

The second solution is based on the discovery that the tool can simulate a specification when the use of one of the variables in the cycle is displaced in time. Instead of using the connecting variable directly in a dependant modeclass it is used via the WHEN statement.
This means it is possible to use a dependency cycle like in Fig.3
Compare the two following definitions:

Definition 1

Source Mode	Events	Destination Mode
mam_idle	@T(t_weight < t_minweight)	mam_nomametchi

Definition 2

Source Mode	Events	Destination Mode
mam_idle	@A WHEN(t_weight < t_minweight)	mam_nomametchi

In definition 1 t_minweight is used directly. In definition2 the value of t_minweight in the state before the actual one is used.
The problem with this solution is that a change in an internal term can not be recognized immediately. It will only be recognized when any of the monitored variables change. So multiple events can be displaced. As the tool is targeted to real-time systems this solution is not really acceptable. But compared to solution one it provides an easy to use simulation possibility.

☹ **Variable definition.**
The tool does not allow more than one mode classes to change the value of a variable. That means, that only one Mode Class is allowed to alter the value of a variable. But in our Tamagotchi Model, we have variables which must be set in diefferent modes classes, for example weight, happyness, hunger and life expectation. As a solution we had to define some variables as offsets, e.g t_deltaWeight, in each mode class. The values of these offsets can be read and added to the main variable ,e.g t_Weight, by the mode class responsible for changing the respective variable. This results in variable explosion, but it works!

1.4.2 Review and Simulation

We reviewed the specification modeled by the UML / Rhapsody group. Our specification was very large compared to theirs, although we had not yet combined our two parts of the

specification. Probably because of redundant variables and mode classes we had (see above). It was difficult for them to understand our specification because it has a lot of tables and one graph, the Dependency Graph (DG). This graph was also very large and confusing. It could not be displayed completely on the 19” screen!
Simulating our specification was also difficult. The simulator displays all variables with sub windows for monitored variables, controlled variables and terms. Because we had many variables, they could not be displayed all at once. So one had to scroll on. With a desert of variables on one screen, it is very difficult for the user to know which one to use, when and how. In this way, it even more difficult to trace the effect of an event and to test the CRs. By the UML / Rhapsody, it was easy to specify the variables one needed to observe, and to trace their effect. There was no error found in our specification besides the fact that we had not yet merged the part specifications.
At the time of the review we had not yet discovered the possibility of the “time displacement” solution. As this makes simulation a lot easier we think our partner team would have get a better impression of the system we used if they revied this version of our specification.

1.5 Conclusions

1.5.1 Teamwork

The teamwork worked very well on the given specification. This results on the format of the informal requirements that could be split in two relatively equal parts worked out by us in separate submodels. The modeling of the submodels could be paralleled very well compared to the sequential work by other teams including the model we reviewed (UML / Rhapsody). The combination of the two submodels had to be done manually because the tool does not provide any help on this. The problems concerning the combination were discussed explicitly in 1.2.

1.5.2 Modeling

Modeling Tamagotchi was helpful to get a deeper understanding of the informal requirements. It also led to a structured overview of the dependencies in the system.
Because of the tricks we had to use to fulfill the requirements a later implementation of the specification would need a lot of rework. A new specification of a bigger system would lead to better results in a shorter time. This is based not only on the experiences we made during the modeling of the Tamagotchi but mainly on the better understanding of the special problems with the tool that should be described in the user guide.

1.5.3 The tool

The tool is working very well on smaller systems like the Safety injection device described in 1.1 because it provides very good modules (CC-Checker) to test models on consistency and completeness. Because of the weaknesses described earlier bigger models like the Tamagotchi are getting more complicated and difficult to survey than they need to be.
Areas in which the tool should be extended include the support of groupwork. The tool should at least support the import of subspecifications into an existing one. Corresponding to that it should support testing and simulating of submodels. The developers of the tool should consider implementing a feature in the simulator that allows changing of terms manually. We know this could lead in an inconsistent state but it would support subspec development a lot.

The developer could be warned and notified about a possibly inconsistent state when changing a term so that the gain of the feature is bigger than the loss of consistency.
Exchanging parts of the model is not supported by the tool. If for example a modeclass is deleted all the variables defined in this modeclass lose their context but are not removed from the specification automatically. The user has to delete them manually to avoid consistency problems.
The tool is a prototype. A really good and complete user guide should not be demanded. But the user guide coming with the tool is really bad. It does not even describe the complete notation. For example we discovered the duration functionality by ourselves through trial and error because the duration-function appeared in a screenshot in the user guide.. This was very helpful for modeling the Tamagotchi. The duration2 functionality, helping to solve a special cycle-problem, was only mentioned in an email from the tool-developers.
Moreover, the Guide doesn’t clearly state the use of Terms and their fundamental difference from controlled variables. These setbacks are big problems for the beginner.
We would use the tool again on modeling smaller systems like the safety injection device but not on bigger systems like the Tamagotchi until the described problems are solved by the developers.