Early Analysis of Ambient Systems SysML Properties using OMEGA2-IFx

xxx removed for blind review purpose xxx

Keywords:
Requirements, Formal Verification, Observers, SysML/Kaos, Relax, Goals

Abstract:
Formal methods provide tools to verify the consistency and correctness of a specification with respect to the desired properties of the system. This verification is important as the development of an AAL (Ambient Assisted Living) system involves different technologies (medical services, surveillance cameras, intelligent devices, etc.) requiring a strong consistency checking between models. We illustrate in this paper how we prove some of the properties of the system before the development even starts. To model the AAL system, we use the SysML language. In terms of tools, we used Rational Rhapsody in combination with the OMEGA2 profile which is an executable Uml/SysML profile used for the formal specification and validation of critical real-time systems. This profile is supported by the IFx toolset which provides mechanisms for the model simulation and properties verification of the AAL system.

1 INTRODUCTION

Formal methods are intended to systemize and introduce rigor into all the phases of software development. This helps us to avoid overlooking critical issues, provides a standard means to record various assumptions and decisions, and forms a basis for consistency among related activities. By providing precise and unambiguous description mechanisms, formal methods facilitate the understanding required to coalesce the various phases of software development into a successful endeavor [1].

In this paper we asses the use of formal methods for the modelling and verification of Ambient Assisted Living (AAL) systems. We use the OMEGA-IFx approach which has been applied for the verification and validation of industry-grade models [2] providing interesting results. The OMEGA Uml/SysML Profile [3] defines the semantics of Uml/SysML elements providing the means to model coherent and unambiguous system models. In order to make the models verifiable, it presents as extension the *observer* mechanism for specifying dynamic properties of models. The OMEGA Uml/SysML Profile is implemented by the IFx Toolbox [4] which provides static analysis, simulation and timed-automata based model checking [5] techniques for validation.

Ambient Systems are highly adaptive. They modify their behavior at run-time in response to changing environmental conditions. For these systems, Non Functional Requirements (NFRs) play an important role, and one has to identify as early as possible the requirements that are adaptable. The distributed nature of Dynamic Adaptive Systems and changing environmental factors makes it difficult to anticipate all the explicit states in which the system will be during its lifetime. As such a DAS needs to be able to tolerate a range of environmental conditions and contexts, but the exact nature of these contexts remains imperfectly understood. For the system properties of our AAL case study, we use two types of properties: Relax-ed and Invariant. These requirements are obtained by applying a process called Relax [6] on traditional requirements. Relax is a requirement engineering language for self-adaptive systems that incorporates uncertainty into the specification of these systems.

In our previous work [7], we have find a link between Relax and SysML/Kaos [8] which is a goal oriented approach, based on Kaos, which extends the SysML meta-model with goal concepts. In our view, these two approaches are complementary for each other and Relax can benefit

from the ContributionNature and ContributionType concepts of SysML/Kaos. This paper enriches our work in integrating the formal aspects.

Paper structure. This paper is organized as follows: Section 2 presents other approaches defined for the verification of AAL models, Section 3 introduces the OMEGA Uml/SysML Profile and the IFx Toolbox that are used for modelling and properties verification, Section 4 describes the case study: its specification, system architecture and behavior, Section 5 presents the properties, the AAL system has to satisfy, while Section 6 contains the verification and simulation results, Section 7 concludes the paper and highlights the future work.

2 Related work

Similar work can be found in literature. [9] introduces a profile named TURTLE (Timed Uml and RT-LOTOS Environment). With its formal semantics given in RT-LOTOS and its toolkit, TURTLE enables a priori detection of design errors through a combination of simulation and verification/validation techniques. By simulation the authors mean a partial exploration of the system state space. It is often used for debugging purposes and to quickly increase confidence in a design. For finite state space systems, exhaustive analysis is also possible. Verification relies on the exploration of the whole system state space in order to prove absence of deadlocks for instance and other general properties that should be satisfied by any system. Validation also relies on exhaustive analysis to demonstrate that a model meets specific requirements, or exhibits a certain behavior such as the validation of a design against the requirements.

[10] introduces Uppaal which is a tool box for validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems. It consists of two main parts: a graphical user interface and a model-checker engine. The idea is to model a system using timed automata, simulate it and then verify properties on it. Timed automata are finite state machines with time (clocks). A system consists of a network of processes that are composed of locations. Transitions between these locations define how the system behaves. The simulation step consists of running the system interactively to check that it works as intended. Then the verifier check the reachability properties, i.e., if a certain state is reachable or not.

3 THE VERIFICATION TOOLS

In this section, we introduce the OMEGA2 Profile which we used for modeling the AAL system and its properties and the IFx toolset used for the verification and simulation of these properties.

3.1 The OMEGA UML/SysML Profile

OMEGA is an executable Uml/SysML profile dedicated to the formal specification and validation of critical real-time systems. It is based on a subset of Uml 2.2/SysML 1.1 containing the main constructs for modelling system structure and class/block behavior and for which provides a clear and coherent operational and timed semantics.

The architecture of an OMEGA model is described in Class/Block Definition Diagrams by classes/blocks with their relationships (association, generalization and composition) and their interfaces. Each class/block define properties and operations, as well as a state machine. the hierarchical structure of a model is defined in composite structures/Internal Block Diagrams: parts that communicate through ports and connectors. The Uml/SysML Profile leave open several semantic variation points for which OMEGA defines a set of well-formedness rules that result in a strong typing language. For further details on the rules, their rationale and formalization, the reader is referred to [11].

The behavior of a system is given by the modelled state machines that use asynchronous operation calls and signal outputs for communication. The profile owns a textual action language compatible with Uml 2.2 action metamodel from which implements the main constructs: object creation/destruction, expression evaluation, variable assignment, signal output and control flow structuring statements.

The operational semantics of OMEGA relies on an asynchronous timed execution model. Each class/block is represented by a timed input/output automata, potentially executing in parallel with other blocks and communicating via asynchronous operation calls and signals.

The OMEGA Profile can model timed behavior, where the model time base can either be

discrete or continuous and it is specified by the user at verification. The time model is controlled by primitives from automata with urgency [12]: clocks, time guards and transition urgency annotations. The *clock* is represented by a *Timer* block on which we can perform actions as: *set* for setting the clock a delay and *reset* to restore the clock to 0. Time guards are either described as inequalities or specified via the *timeout* operation that verifies that a certain delay has elapsed. With respect to time progress, transitions can also define a particular semantics based on their stereotype: *eager* defines that time progress is disabled in a state (i.e., the actions on a transition are executed as soon as possible), *delayable* means that the time progress is enabled but it is bounded by a limit and *lazy* specifies that time progress is enabled and unbounded (i.e. time can progress to infinity). Based on these notions, one can also model synchronous communication in OMEGA model.

For specifying and verifying dynamic properties of models, OMEGA uses the notion of *observers*. *Observers* are special classes/blocks monitoring run-time state and events. They are defined by classes/blocks stereotyped with `<<observer>>`. They may have local memory (attributes) and a state machine describes their behavior. States are classified as `<<success>>` and `<<error>>` states to express the (non)satisfaction of safety properties. The main issue in modelling observers is the choice of events which trigger their transitions, and which must include specific UML/SysML event types. One can observe:

- Events related to signal exchange: `send`, `receivesignal`, `acceptsignal`.
- Events related to operation calls: `invoke`, `receive` (reception of call), `accept` (start of actual processing of call – may be different from `receive`), `invokereturn` (sending of a return value), `receivereturn` (reception of the return value), `acceptreturn` (actual consumption of the return value).
- Informal events explicitly specified by the modeller using the informal action.

The trigger of an observer transition is a `match` clause specifying the type of event (e.g., `receive`), some related information (e.g., the operation name) and observer variables that may receive related information (e.g., variables receiving the values of operation call parameters). Besides events, an observer may access any part of the state of the UML model: object attributes and state, signal queues.

3.2 IFx Toolset

OMEGA models can be simulated and properties can be verified using the IFx toolset [13]. The following terminology is used:

Verification: It designates the automatic process of verifying whether an OMEGA2 UML/SysML model satisfies (some of) the properties (i.e. *observers*) defined on it. The verification method employed in IFx is based on systematic exploration of the system state space (i.e., enumerative model checking).

Simulation: It designates the interactive execution of an OMEGA2 UML/SysML model. The execution can be performed step-by-step, random, or guided by a simulation scenario (for example an error scenario generated during a verification activity).

The IFx toolset relies on a translation of UML/SysML models towards a simple specification language based on an asynchronous composition of extended timed automata, the IF language[1]. The translation takes an input model in XMI 2.0 format. The compiler verifies the set of well-formedness rules imposed by the profile and generates an IF model that can be further reduced by static analysis techniques. This model is subject to verification that either validates the model with respect to its properties or produces a list of error scenarios that can be further debugged using the simulator. The overall workflow of IFx Toolset is shown in Fig. 1 [14].

4 MODELING THE AAL SYSTEM WITH OMEGA PROFILE

In this section, we explore the AAL system specification, the system architecture showing its structural and behavioral diagrams i.e. its block definition diagram, internal block diagrams and state machine diagrams.

4.1 System Specification

First, we start by taking into account the structural part of the AAL system. We consider those

[1] `http://www-if.imag.fr/`

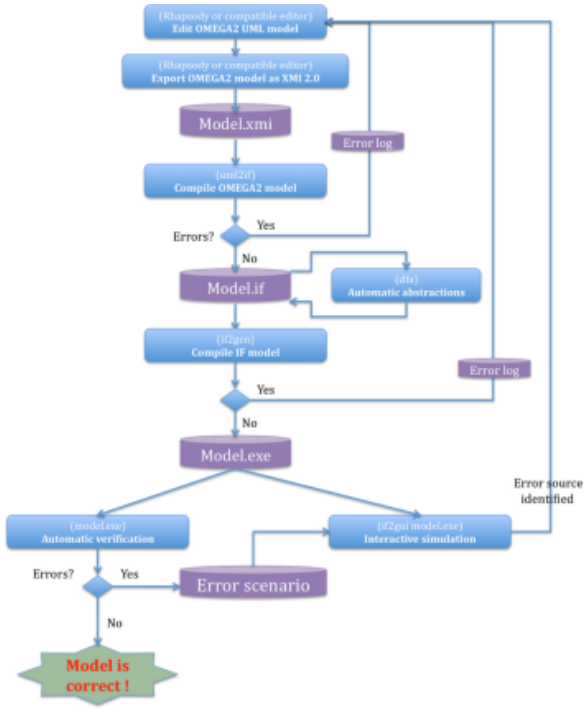

Figure 1: IFx Workflow

parts that are concerned with the daily calories intake of the Patient in the AAL house. The AAL system is composed of Fridge and Patient. We would like to model these parts and the interaction that takes place between them. The Fridge partially contributes to the minimum liquid intake of the Patient; it also looks at the calories consumption of the Patient as the Patient needs not to exceed it after a certain threshold.

4.2 System Architecture

Fig. 2 shows the Main internal block diagram. The important parts of the AAL system are Patient and Fridge. A Fridge in turn is composed of Food, Display, Alarm and Controller blocks. The block Food contains information about the food items in the Fridge, the calories contained by each item, the total number of calories the patient has accumulated and the calories threshold that should not be surpassed. The Fridge Display is used to show the amount of calories consumed by the Patient. The Alarm is activated in case the Patient's calories level surpasses a certain threshold. The Controller transfer the information from the Patient to the concerning elements and back to the environment. The communication between different blocks takes place through ports. A port bears a type. In OMEGA2, the type of a port must be an Interface. The type specifies the set of requests (operation calls and/or signals) that are transferred between parts (components) by means of ports and connectors. In Fig. 2, the Patient block has a standard port named pToFridge. This port has a contract named Patient2Fridge and is acting as a provided interface of the Patient block. The Interface Patient2Fridge defines an operation *eat(int item, int quantity)*. This interface is then used as a type of pToFridge port. At the same time the Patient block has a required interface named pFromPatient. The full system architecture of the AAL system can be found in [15].

Rational Rhapsody Developer v7.5.2. [16] is used to create OMEGA2 models. OMEGA2 models use a profile and a predefined library provided with the tool (OMEGA2.sbs and OMEGA2Predefined.sbs). Any other UML2.2 or SysML1.1 editor supporting profiling and exporting in the XMI2.0 standard compatible with Eclipse ecore can be used for OMEGA models.

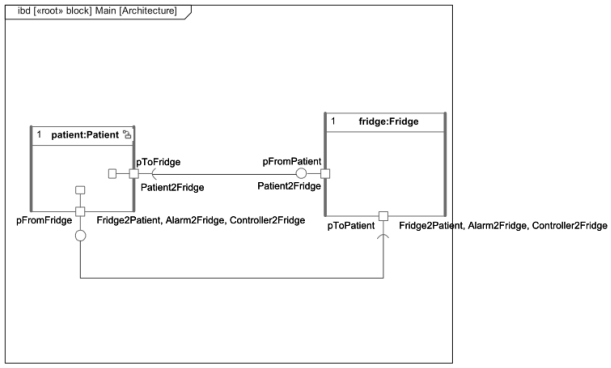

Figure 2: Main Internal Block Diagram

The important part of the AAL system is the intelligent Fridge. The Fridge interacts with the AAL system. Fig. 3 shows the internal block diagram for the Fridge block. Each of the four blocks behaviors is modeled in a separate state machine diagram.

Fig. 4 shows the state machine diagram for the Patient block. Here the exchange of information between Patient and Fridge takes place. We identify the number and quantity of each item present in the Fridge. If a certain product still present in the Fridge is chosen by the Patient then the information is communicated with the Fridge. Otherwise the Fridge is empty and the Patient will wait to be refilled. Also, if the Alarm of the Fridge is raised due to high intake of calories, the Patient stops eating and waits for the system to be un-

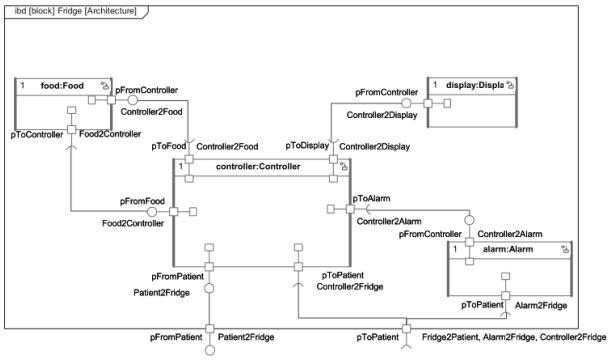

Figure 3: Fridge Internal Block Diagram

blocked.

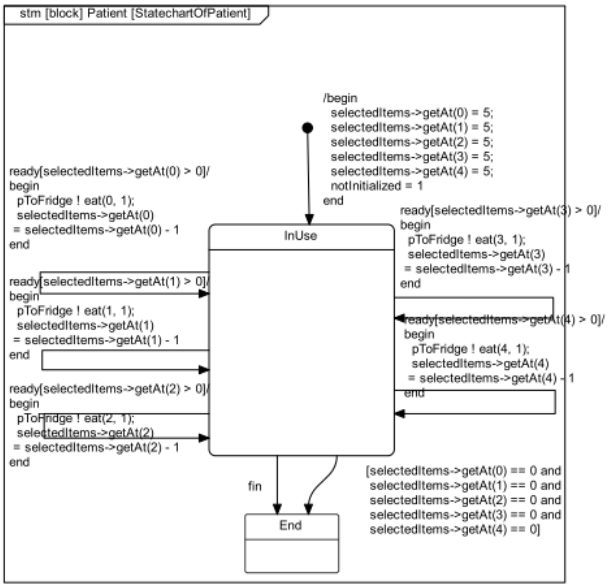

Figure 4: Patient State Machine Diagram

The Food block models the knowledge of the Fridge about what it contains. Here we define the number of items and the amount of calories associated with each item present in the Fridge. We then calculate the total number of calories accumulated by the Patient. If the total number of calories is greater or equal to the maximum calories allowed for the Patient, then a message is send and the alarm is raised or if the total number of calories is greater than the maximum calories allowed minus 500, then the Patient is warned with a message that the calories level is approaching the maximum amount of calories allowed. Fig. 5 shows the state machine diagram for the Food block.

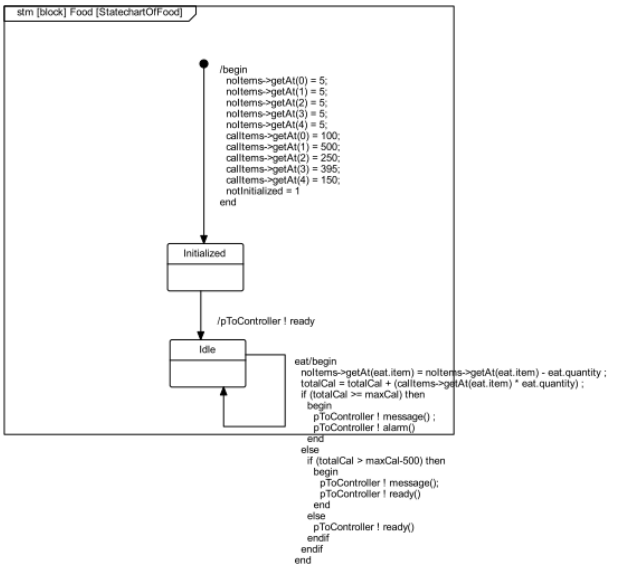

Figure 5: Food State Machine Diagram

5 PROPERTIES VERIFICATION OF AAL SYSTEM

The properties of AAL system that we modeled and verified are obtained after RELAX process is applied on its traditional requirements. RELAX is a requirement engineering language for self-adaptive systems that incorporates uncertainty into the specification of these systems. Typical textual requirements prescribe behavior using a modal verb such as SHALL that defines functionality that a software system must always provide. For self adaptive systems such as AAL however, environmental uncertainty may mean that it is not always possible to achieve all of those SHALL statements; or behavioral uncertainty may allow for trade-offs between SHALL statements to RELAX non-critical statements in favor of other, more critical ones. Therefore RELAX identifies two types of requirements: one that can be RELAX-ed in favor of other ones called variant or RELAX-ed and other that should never change called invariant. Below are the properties to be verified:

5.1 Traditional/Relax-ed Requirement

- The fridge shall detect and communicate with food packages

RELAX-ed version of this requirement is as follows:

- Property 1 : The fridge SHALL detect and communicate information with AS MANY food packages AS POSSIBLE

Below are the uncertainty factors associated with the given RELAX-ed requirement.

ENV: Food locations, foot item information (type, calories), food state (spoiled and unspoiled)

MON: RFID readers, Cameras, Weight sensors

REL: RFID tags provide food locations and food information; Cameras provide food locations (Cameras provide images that can be analyzed to estimate food locations), Weight sensors provide food information (whether eaten or not)

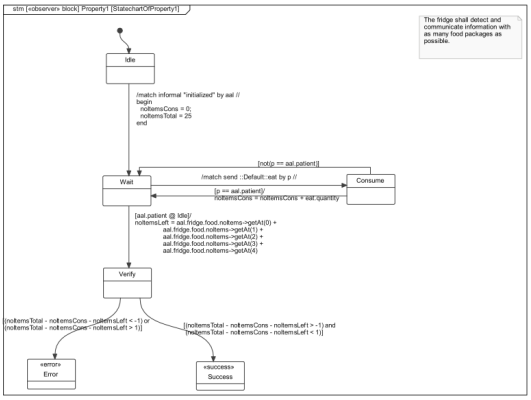

Figure 6: Property1 State Machine Diagram

The satisfaction of this requirement contributes to the balanced diet of the Patient. We would like to verify this property as it is important for the AAL system to know about as many food items as possible present in the fridge. Fig. 6 shows the state machine diagram of the Property 1. In this property, we identify the number of items consumed by the Patient and the total number of items in the Fridge. First of all, we verify the identity of the Patient, if the person is identified as the Patient, then we calculate the number of items consumed. We then calculate the number of items left in the Fridge which is equal to the sum of all the items present in the Fridge. Then in the last step, we calculate if the total number of items minus the number of items consumed minus the number of items left is greater than -1 and the total number of items minus the number of items consumed minus the number of items left is less than 1, it means that we have reached the `<<success>>` state by having all the information about all the items present in the Fridge, .i.e. it should be between -1 and 1 (there is no information loss). Inversely, if the total number of items minus the number of items consumed minus the number of items left is less than -1 or the total number of items minus the number of items consumed minus the number of items left is greater than 1, then it means that we are missing some information about some of the items present in the Fridge and the *observer* passes into the `<<error>>` state.

5.2 Invariant Requirement

- Property 2 : The Alarm SHALL be raised instantaneously if the total number of calories surpasses the maximum calories allowed for the patient

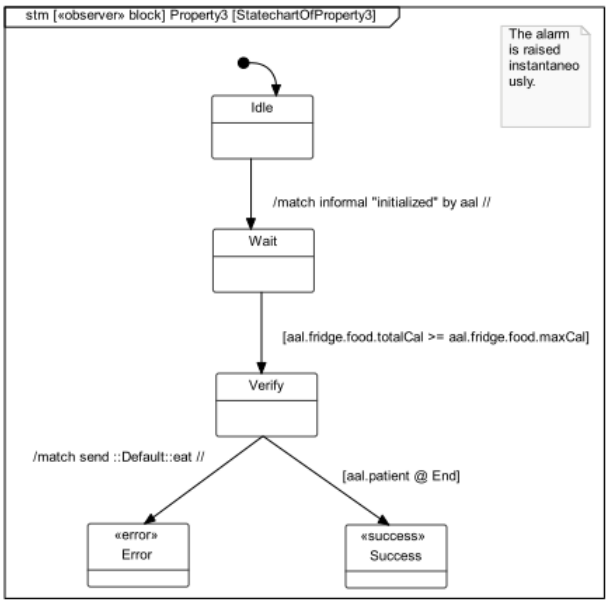

Figure 7: Property2 State Machine Diagram

This property ensures that the Patient should stop eating when the total number of calories surpasses the maximum calories allowed and that the Alarm should be raised. Fig. 7 shows the state machine diagram of the property 2. This requirement implies that the Alarm shall be immediately raised as soon as the total number of calories equals or surpasses the maximum calories allowed for the Patient. If it happens then the Patient should stop eating.

6 Verification Results

Until now, we have modeled the AAL system and the properties to be verified on the model. It is now time to verify these properties and in case if there is error during its verification, simulate it to find the error and then correct it in the model. Fig. 8 shows the snapshot of the compilation of the AAL model named AAL2. The AAL2 model is first exported into AAL2.xmi and then using the IFx toolset the AAL2.xmi is compiled into AAL2.if.

```
[ahmad@topcased ~]$ uml2if -sysml -rhapsody -rhplang -eager AAL2.xmi
uml2if (OMEGA2) v2.0.1  (c) Verimag,IRIT 2009-2011
Analyzing input.
Please wait...
Success
Generated...
Preprocessed...
Indented...
Done.
[ahmad@topcased ~]$
```

Figure 8: XMI to IF Compilation

Figure 9: IF to Executable file Compilation

Fig. 9 shows the snapshot of the compilation of AAL2.if into an executable file i.e. AAL2.x. For the properties verification part, we run the model-checker with the following options:

AAL2.x -dfs -po -me -ce ln

While verifying the AAL model, the model-checker has found several error scenarios, as can be seen in Fig. 10. Any of the error scenario can then be loaded through the interactive simulation interface of the IFx toolset to trace back the error in the model and then correct it.

```
[ahmad@topcased ~]$ ./AAL2.x -dfs -po -me -ce ln
00:11:34    2140556/s    5468814/t    326/d reached error state [2141463]
reached error state [2141465]
reached error state [2141467]
reached error state [2141474]
reached error state [2141476]
reached error state [2141478]
reached error state [2141488]
reached error state [2141490]
reached error state [2141492]
reached error state [2141515]
reached error state [2141517]
reached error state [2141519]
reached error state [2141532]
reached error state [2141534]
reached error state [2141536]
00:11:35    2143450/s    5472991/t    554/d reached error state [2143590]
```

Figure 10: Model Checker results in Error Scenarios

In order to debug a model, firstly we import it into the simulator as shown in Fig. 11 .

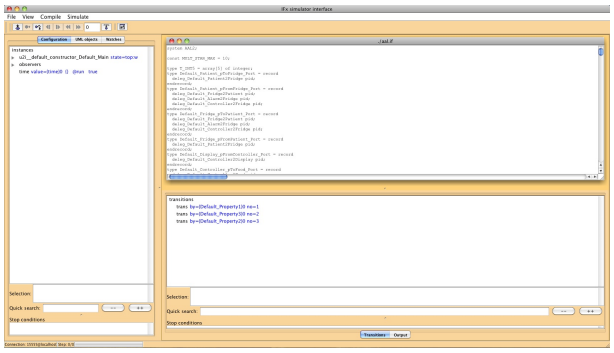

Figure 11: Initial Simulation Interface

We check the states of the *observers* in order to identify which property has not been satisfied. One can observe in Fig. 12 that Property 2 fails. While checking the state of the entire system for this property, we discover that the error state contained the maximal allowed number of calories for the total number of calories consumed and subsequently eat requests sent by the Patient. This implies that the Alarm function of the intelligent Fridge doesn't function properly. The Alarm function of the Fridge is strictly relied to its Food process. One can observe in the state machine of the Food block (Fig. 5) that the Alarm is raised only if the total number of consumed calories is strictly superior than the maximum allowed; condition which doesn't satisfy the request that the Alarm is raised as soon as possible. The correction consists in raising the Alarm in case the total number of consumed calories is equal to the maximum allowed threshold. Once this error is corrected the verification succeeds.

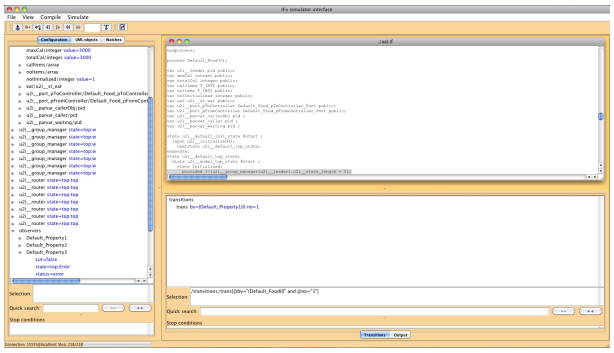

Figure 12: Error State Food Observer Simulation Interface

```
[ahmad@topcased ~]$ ./AAL2.x -dfs -po -me -ce ln
01:46:03    17058583/s    42971642/t    179/d
```

Figure 13: Model checking successful

Fig. 13 shows the result of the model-checker

on the correct model.

7 Conclusion

We have modeled the structural and behavioral parts of an AAL (Ambient Assisted Living) system. The modeling is done using Rational Rhapsody 7.5.2 with OMEGA2 profile which is used for specification and verification of dynamic properties of models through *observers*. For the verification and simulation part, we have used IFx which is a toolset used for the simulation of OMEGA2 models and the verification of properties defined on these models. We have verified two properties of the AAL system using the IFx toolset. At first, the verification results in errors which can then be simulated through the interactive simulation interface of the IFx toolset in order to identify the source of the error and then subsequently correct it in the model, after correcting the error in the model, the verification results in the fulfillment of all the two properties.

The future work is centered around the use of *observers* in the context of our integrated approach [17]. This work motivated us to take benefit from the OMEGA2/IFx. In our integrated approach, we are interested in RELAX requirement which we obtain by using the RELAX process. We then refine the RELAX requirement with the ContributionType and ContributionNature of SYSML/KAOS which is a SYSML profile with the goal concepts integrated into it, The RELAX requirement is then verified by a test case and then the test case can be refined by observer and then observer is allocated to a state machine diagram. This whole sequence of steps constituite our meta model.

REFERENCES

[1] `http://www-2.cs.cmu.edu/~svc/`

[2] **xxx removed for blind review purpose xxx**

[3] **xxx removed for blind review purpose xxx**

[4] **xxx removed for blind review purpose xxx**

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking. MIT Press, 1999.

[6] **xxx removed for blind review purpose xxx**

[7] **xxx removed for blind review purpose xxx**

[8] Christophe Gnaho, Farida Semmak. Une extension SYSML pour l'ingénierie des exigences non-fonctionnelles orientée but. Revue Ingénierie des Systèmes d'Information, Vol 16/1, 23 pages, 2011.

[9] Ludovic Apvrille, Pierre de Saqui-Sannes, Ferhat Khendek. TURTLE-P: a UML profile for the formal validation of critical and distributed systems, Software and System Modeling 5(4): 449-466 (2006).

[10] `http://www.it.uu.se/research/group/darts` /uppaal/

[11] **xxx removed for blind review purpose xxx**

[12] S. Bornot and J. Sifakis, "An algebraic framework for urgency," *Information and Computation*, vol. 163, 2000.

[13] **xxx removed for blind review purpose xxx**

[14] OMEGA-IFx for UML/SYSML v2.0 Profile and Toolset, User Manual Document version 1.1

[15] `https://wwwsecu.irit.fr/FILEX/get?k=n21pV3UgsOtvJC4PSvx`

[16] IBM, Rational Rhapsody v7.5. reference manuals. Available via `http://www.ibm.com/developerworks/rational/`.

[17] **xxx removed for blind review purpose xxx**