Model-driven Specification, Analysis, and Realization of Assisted Living Systems

András Balogh, András Pataricza, Gergely Pintér, Áron Sisak
Budapest University of Technology and Economics, Department of Measurement and Information Systems
Magyar Tudósok krt. 2., Budapest, H-1117, Hungary
Phone: (36)-(1)-463-3594, Fax: (36)-(1)-463-2667, {abalogh,pataric,pinter,asisak}@mit.bme.hu

Michalis Anastasopoulos, Jaejoon Lee
Fraunhofer Institut Experimentelles Software Engineering
Fraunhofer-Platz 1, Kaiserslautern, 67663, Germany
Tel: (49) (631) 68-00-0, {Michail.Anastasopoulos,Jaejoon.Lee}@iese.fraunhofer.de

Abstract - Ambient intelligent systems continuously gain importance in a variety of application domains. Nowadays one of the most important ambient intelligence domains is assisted living, which involves devices and services supporting elderly people during their everyday life. Typical assisted living applications include health monitoring, remote nursing, and alarm systems.

Model-driven design methodologies are gaining more and more importance in several application domains including traditional e-commerce systems and recently also embedded safety critical systems.

The characteristics of ambient systems (low computing power, strong resource limitations) arise challenges for the developers. The current paper introduces new techniques and tools that can help to overcome the design difficulties using modern, model-driven methodologies, furthermore interactive simulation and formal analysis tools are also integrated in order to increase the productivity of the development process.

I. INTRODUCTION

Ambient intelligence systems in general are usually composed of multiple networked computing nodes, which are typically resource-constrained due to the limited physical space and electric power available. Such devices include smart wireless sensors, handheld devices, or display devices that frequently interact with traditional personal computers with broadband network access.

The hardware resource constraints and the additional non-functional requirements especially in the case of assisted living (i.e. dependability including real-time behavior, high availability, safety) form a complex challenge for application development. A proper tool support is crucial both from the point of view of productivity and quality.

Model-driven development gains recently a high popularity in enterprise systems development, and is continuously evolving in the embedded systems domain. Existing tools and methods offer support even for such demanding fields, like automotive, aerospace, and industrial process control applications. However, the affordable and consequently the strict cost and resource constraints for ambient systems make the development even more complex compared with industrial applications.

Our paper introduces a tool framework that implements model-driven specification, analysis and realization of ambient intelligence systems and is tailored to the specific needs of assisted living applications. Our tools are built on the standard Eclipse tool integration platform that enables the integration with existing tools, like UML[2] modeling tools, source code management systems, compilers, debuggers, and so on. The result of the integration is a complex yet easy to use environment supporting all aspects of the software development process.

The model-driven specification is performed using the Unified Modeling Language (UML), which is specialized by a domain-specific profile. The latter captures both functional and non-functional aspects of the target system and platform. Class diagrams are used to model the architecture of the system, while statecharts describe its behavior.

For the model-driven analysis several analysis and simulation methods are integrated into the tool suite. In fact interactive simulation and model checking of the statechart specifications are used for automatically assessing the functional correctness. To that end an established model checker is being used. The input to the model checker is generated automatically.

Qualitative analysis tools are used in order to estimate several parameters of the system under design, like temporal behavior, availability or power consumption. The set of analysis tools can be easily extended using a simple plug-in mechanism.

The model-driven realization is supported by an application-level simulator that enables the interactive execution of the complete application. The simulation platform is Java, and the major part of the source code is generated automatically. The simulator substitutes the sensors and actuators of the embedded target system with graphical user interface elements that simulate inter-component message exchange.
Application level simulation also includes the execution of UML statechart specifications in order to be able to test and validate the internal behavior of the software components.

The paper also shows how the implementation of the final system (written in ANSI C portable to a variety of embedded tiny operating systems) can also be automatically generated by the tool chain. The resulting C code contains the skeleton of the RTOS tasks and also the implementation of the UML statecharts [3].

Our proposed tool framework includes several novel components that complement existing tools in supporting model-driven development with analysis, simulation, and automatic synthesis capabilities. The tools are integrated into a solid framework integrated into the Eclipse [4] tool platform.

II. DEMONSTRATOR APPLICATION

A simple ambient assisted living demonstrator application has been chosen to validate the methodologies and tools developed during the project.

The application is an intelligent cup (iCup – Fig. 1) that supervises the drinking of elderly persons. As they do not feel dry, they often forget to drink that can cause serious health problems.

The iCup has a tilt sensor that monitors the angle of the cup and detects drinking this way. It also detects if it is near to a liquid source (for instance a water tap) and updates its liquid content based on the information coming from the liquid source.

If the owner of the cup does not drink enough, or does not drink for a period of time, the cup starts to beep using its internal beeper. It also sends an alarm to the central home monitoring computer through RF network.

If there are multiple cups nearby, they communicate to synchronize the drinking information. That way, false alarm signals can be avoided.

The prototype cup contains an 8 bit (Atmel AVR) microcontroller equipped with 4 Kbytes of RAM, 128 Kbytes of flash ROM, and a radio module. It is battery powered. This means a very constrained environment for application development.

III. MODEL DRIVEN DEVELOPMENT

Model-driven development (MDD) techniques collect several different system development tools and methods. The Object Management Group (OMG) [5] defined a standard, model driven development methodology called Model-Driven Architecture (MDA) [6]. MDA establishes a common nomenclature in the model-driven community (Fig. 2).

The development starts with the platform-independent model (PIM) of the system under design. This model describes only the core functional (and recently extra-functional) aspects of the system and does not contain implementation-related information.

The platform description (PD) is also a model that describes the execution platform that has been chosen to implement the system. It typically contains information about the runtime hardware-software environment (resources, nodes, networks, operating system, middleware, etc.). Most typically this model is reusable as the runtime platform is the same for several applications.

A (semi) automatic model transformation maps the PIM using information from PD to a platform-specific model (PSM). This model describes the integration of hardware and software components and contains all information required to build and deploy the software on the target hardware.

The source code of the system is derived from the PSM using the combination of automatic code generation and manual coding.

The main advantage of the MDA approach is that it separates the application-related and platform-related modelling aspects, that results in better portability of application models, and improved reusability of platform models.

Through the extensive utilization of automatic synthesis methods (model transformation and code generation) the productivity of the development process will increase, together with the quality of the implementation – thanks to the automatic, well-tested code generators.

However MDA is an everyday practice in business system development, and a promising solution in several other domains, recent research and development experiments have pointed out that it might have problems in
constrained and complex platforms like distributed embedded systems [7][8].

The main disadvantage of the original MDA approach is the automatic PIM to PSM mapping. In order to be able to automatically map software to hardware, all relevant information should be integrated into the input models. In case of a business application this only involves functionality, performance, and in several cases reliability requirements. In case of a distributed embedded application this should be complemented with power consumption, physical distribution, etc. This is practically impossible, or would require huge and complex models that are impossible to handle.

To overcome this problem embedded model-driven tools introduce an iterative and interactive mapping process for the PIM-PSM transformation (such as DECOS [8]) where the developer can customize the mapping and make human engineering decisions instead of trying to formalize each aspect of the system under design.

BelAMI utilizes the experiences collected during the development and evaluation of the DECOS tools and builds on the existing knowledge in the MDD field.

The main modelling language is UML (Unified Modelling Language [10]), extending it with a domain specific profile. UML is used both for software and hardware modelling, and also for behaviour specification. A UML profile is a set of stereotypes and attributes that are used to tag existing UML elements to express specific meaning/characteristics of the given element.

Figure 3 illustrates part of the profile (software functionality). A BelAMI system executes services that are composed of ServiceComponents. These are software components that can be hierarchically refined (parent-child relationship). Software components communicate using messages and may use resources (sensors and/or actuators). Service components can have internal behaviour specified by UML statecharts.

There are several other parts of the profile. Dependability and performability related properties are also defined for software model elements in order to support extra-functional requirements modelling.

Dependability describes the possible error modes and rates of components, whether redundant structures have to be implemented, and the overall availability requirements for the services. This attributes are used by the analysis tools during model validation.

The performance package contains execution time and frequency related information about the software components and transmission periods for messages. These data is also required for model validation.

The profile also has a platform specification part that describes the node types, their resources (memory, CPU, communication controllers, sensors and actuators), and the network links of the system.

The hardware part has also a dependability related package that is used to describe availability, error rates, and other dependability related attributes of the runtime platform. This is a simplified dependability model for high level validation of the system models.

As the power consumption is a critical factor in ambient applications, the hardware power consumption is also modelled using a high-level model that contains the typical consumption of the hardware units for the most important operation states and an analysis tool can calculate the time spent in those states based on the software model.

It must be noted that there are more fine-grained models both for hardware dependability and for power consumption, but based on our experiments this resolution is enough to do conceptual validation of system models during the early design steps.

IV. BEHAVIOUR SPECIFICATION

The profile discussed in the previous Section described only the structural part of the software components. For behaviour specification UML statecharts are used.

The statechart language of UML is a hierarchical concurrent state machine notation. It contains several high level concepts (history states, transition guards and triggers, actions) that form a powerful specification environment.

A statechart implements an event-triggered behaviour. It waits for input events and tries to find transitions that are
A specific property holds on a model or not. It involves the algorithms. The technique is used mainly for protocol traverse of the state space of the system model. This is the component behaviour.

Validation requires sophisticated, memory and CPU time efficient current paper. We focused on the statechart specification of the demonstration application. The cup amount observer component is responsible for the maintenance of the liquid level inside the cup.

After starting, based on the actual angle the cup will decide whether to go into the empty, some, or full state. If the angle is updated, the maximal amount will be updated (if the cup is tilted, it can hold less water).

If the cup receives waterPoured signal from a liquid source, it will update the actual amount.

In both of the previous cases, the statechart will end in the VariablesUpdated state. This has an immediate transition to the first pseudo state that leads to a cyclic behaviour (this transition is not shown to keep the Diagram readable).

This simple example shows that the behaviour specification can be simple and visual by using UML statecharts.

IV. MODEL CHECKING UML STATECHARTS

Formal verification of software behaviour is a wide research and development topic in the computing theory. We targeted a single part, namely model checking in the current paper. We focused on the statechart specification of the component behaviour.

Model checking [11] is a procedure that verifies whether a specific property holds on a model or not. It involves the traverse of the state space of the system model. This requires sophisticated, memory and CPU time efficient algorithms. The technique is used mainly for protocol validation [12], but also in several safety-critical system development [13], and processor design [14].

Most of the model checker tools use finite state machine like formalisms as input model. The properties that should be checked are specified using temporal logic [15] [16] expressions. These logics are usually based on first order logic enriched by temporal operators (next state, future state, all states, etc.).

The finite state machine notations do not contain concurrency and state hierarchies; therefore the UML statechart models have to be converted to flat, simple state machines. In order to do this, a formal semantics of the UML statecharts have to be defined. Unfortunately, the UML standard itself is ambiguous and informal; therefore a custom, formal operational semantics has been defined [3] [17].

The main elements of the formalization are the following ones:

- Unambiguous definition of refinement-related concepts (states, regions, sub-states, sub-regions)
- Formalism for compound transition structures – possibly involving multiple transitions and pseudo states
- Formalism for compound activity structures
  - Any number of activities (state entry, exit, transition affects)
  - Specification of subsequence relations between activities
  - Enables exploitation of parallel computing resources (if supported by the implementation platform)
- Formal definitions of the operational semantics
  - Based on Kripke transition system
  - Declarative specification mapped to easy to implement algorithms

The formal semantics is a common solid foundation for achieving multiples goals, like model checking, automatic source code synthesis, runtime error detection, and behaviour simulation.

The model checking is implemented using SAL (Symbolic Analysis Laboratory) [18], an open-source widely used model checker suite developed at SRI International. It implements several model checking methods (classic and bounded model checking), and also contains simulation and deadlock analyser tools.

The language of SAL is based on the mathematical transition systems. A model contains states, transitions, and state variables. The type system for variables is rich, as it contains both primitive types (e.g. natural numbers) and composed structures like arrays and lists.

SAL has multiple equivalent representations for its programs. XML is used for model interchange between SAL and other applications and the textual syntax provides direct programming interface. SAL is basically meant as a target language for analysis, thus it is optimal for code generation.

The generated SAL code that represents the statecharts consists of two parts. The data structures describing the concrete statechart are generated by a code generator module. It describes the states, regions, and triggers that are implemented as enumerations.

The implementation of guards and effects is generated by customizing generic code templates. The actual effect specifications can also be copied from the UML model (if specified).

The statechart engine implementation is the direct implementation of the formal operational semantics described earlier. It receives events and calls behavioural
operations that use the data structure described above. The engine code is generated according to the model using generic templates.

The model checker also needs property definitions that can be evaluated on the model. These should be specified using linear temporal logic (LTL) formulae.

There are several generic properties that are important for all systems in our domain. The first one is the safety property. Safety means that “something bad never happens”. More formally, if the system reaches a given state (S_1), it should never reach an other (S_2). For instance, if a system completes the initialization, it should not reach the boot-up state again.

The other usual property is the liveness of the system. The informal meaning is “something good will happened eventually”. More formally, if the system reaches a specified state (S_1), it should eventually reach an other one also (S_2). For instance, if the iCup is switched on, it will reach the Empty, Some, or Full states.

These generic criteria can be generated automatically, if the developer tags the states (or state sets) using UML stereotypes <<init_s>> and <<unsafe>> for safety analysis, and <<init_l>> and <<live>> for liveness analysis, respectively.

The developer might also want to specify custom and application specific properties. Currently, the only possibility is to use the LTL formalism directly. Unfortunately this is a low-level mathematical language that might be hard to learn for engineers; therefore we are working on higher level specification languages.

V. SOURCE CODE SYNTHESIS

![Fig. 5: Source Code Synthesis Overview](image)

One of the key parts of model-driven development is the automatic code synthesis. Most modelling tools offer code generation capabilities. However, these are standard features; the generator only uses the structural model elements (classes, associations, etc.); therefore the resulting source code contains only the code skeletons (methods, parameters, classes, etc.) but no behaviour.

Our goal was the generation of behaviour from statecharts besides the structural code parts. This generation is also based on the common operational semantics introduced earlier in this paper.

The automatic implementation of a such complex formalism as a statechart is definitely a nontrivial issue. The usual naïve approaches (e.g. implementing the state-transition logic by nested switch statements or state-transition tables) are unable to handle such fundamental constructs as state refinement or parallel execution, not even the well-known State design pattern [19] is capable of supporting these concepts.

The solutions published in various research papers are unfortunately also restricted to a subset of UML statechart features. Event the best-known Quantum Hierarchical state machine (QHsm) implementation technique explicitly proposed for embedded systems by Samek [20][21] is restricted to non-concurrent statecharts.

Having taken into consideration the lack of a full-featured embeddable solution we decided to adapt the Model Driven Architecture initiative for this challenge. In our case, the platform-independent modelling phase of MDA corresponds to the metamodel of precise statecharts, and the definition of the algorithms implementing the operational semantics.

The platform-specific phase we had to (i) identify those characteristics of resource-constrained embedded systems that may require some modifications both in the metamodel and the algorithms. The dominant properties of such systems are the following ones:

- low computing power
- serious memory constraints
- lack of hardware support for parallel execution
- need for deterministic or even real-time operation

In correspondence to these observations we carried out modifications in the metamodel and the algorithms specifying the operational semantics as follows:

- we substituted the complex algorithms (that calculate a a possibly parallel execution order of various activities) with simple algorithm that calculate a single valid sequence of activities (reducing this way the processing power requirements and taking into consideration the lack of parallel execution possibilities)
- substituted the recursive or mutually recursive function structures with iterative algorithms (supporting this way the pre-calculation of execution times for real-time operation)
- introduced compact representation of configurations and similar data (reducing this way the memory consumption)

The resulting platform specific language consists of a modified metamodel and a set of modified algorithms. We proved the semantics equivalence of the PIM and PSM representations by comparing the corresponding algorithms line-by-line and discussing the modifications and their correctness.

The final step of the design (according to the MDA concept) is the implementation phase. The goal of this step is to implement the platform specific model in a programming language that seamlessly fits the target platform. In our case this means the implementation of the PSM metamodel and algorithms in the ANSI-C language, as data structures and functions, respectively.

In order to achieve this, first we prepared an annotated metamodel of precise statecharts indicating how to implement the corresponding model element in C (e.g., by built-in data type, a structure, or an enumerated type) then implemented the data structures and algorithms.

Having built the theory behind the PIM, PSM, and implementation phases we implemented the process in an automatic code generator. Our implementation expects the
models in the CML metadata interchange (XMI) [22] format supported by most of the UML modeling tools, enabling this way the seamless integration into popular environments.

On user level, the process of source code synthesis is a relatively simple process (Fig. 5). The user creates the statechart model using a UML modeling tool. He can either specify the atomic activity implementations directly in the UML model, or later by filling in the function skeletons generated by the tool.

The code generator takes the UML model as input and generates the data structures and functions described earlier. Some glue code (drivers for peripherals, physical to logical event mapping, etc.) has to be completed manually. The resulting source code can be compiled using the C compiler of the target platform.

VI. SIMULATION ENVIRONMENT

While the behavioural analysis offers formal verification for the statechart models, the assessment of the complete system behaviour is not currently supported. As the target system is a distributed embedded system, it might be hard to test the software on the real hardware units and to debug the code.

An application level simulator has been developed in order to complement the existing formal analysis method with an interactive, user-directed simulation facility.

The simulation environment is executed on the developer computer and is a Java based application. The application relies on Eclipse Equinox [23], an OSGi (Open Services Gateway Interface) 0 framework implementation.

This environment supports the integration of loosely coupled components executed on multiple threads. The platform also has inter-component communication facilities (both synchronous and asynchronous). These characteristics are ideal for distributed application simulation.

The code synthesis tool described earlier in this paper has been modified in order to support Java code generation. According to the model-driven design of the code generator this results in the modification of the platform-specific metamodel and algorithms to meet the specifics of Java.

The statechart data structures are implemented using as classes, and the code generator only instantiates them according to the model. The actions, signals, and triggers are also classes that contain the implementation of their specific behaviour. Guard conditions and action

![Fig. 6: User Interface of the Simulator](image-url)
implementations should be implemented by the developer either by directly typing them into the UML modeller tool, or by extending the generated code skeletons.

The implementation of the algorithms forms a standard, out-of-the-box Java library that can be used in any Java program without modifications.

The common semantic background of Java and C code generators assures that the simulated and real behaviour will be equivalent.

VIII. RELATED WORK

Model-driven development has several applications in multiple IT domains. However, it is widespread in the enterprise and web development fields, it is still not widely used in the embedded domain.

There are several tools that support MDD in embedded environments, such as iUML [25] and the DECOS tool chain [8], but these environments typically target powerful 32 bit systems with robust backbone networks.

There are several strategies for UML statechart-based implementations ([19][20][21]). But –as discussed earlier – none of them supports the complete UML specification. Our proposed method not only covers the UML standard but it also defines a precise operation semantics for statecharts.

The targets of the code generator (SAL, Java, C) cover the whole development process from model analysis, verification, simulation, and implementation, and all targets rely on the same solid semantics.

IX. CONCLUSIONS

The model-based development methods and tools introduced in this paper support the development of highly resource-constrained embedded ambient systems.

A domain-specific profile of UML has been developed in order to support the development of distributed systems. The profile supports both platform and software modelling, and also the software-hardware integration. The standard UML modelling environment is extended with custom tools that provide code synthesis methods are tailored to the characteristics of such platforms.

Besides the code generation, formal analysis methods are also introduced in order to support early assessment of system behaviour that helps to discover problems in the earlier design steps that results in increasing productivity and decreasing design costs.

The environment is complemented by an application level simulator that is able to present the behaviour of distributed software applications. The simulator has an interactive, visual user interface in order to help to understand the behaviour of the system.

The concepts and tools have been validated using an ambient assisted living demonstration application from the BelAMI German-Hungarian joint project.

REFERENCES

[1] BelAMI Project http://www.belami-project.org/