# Extracting Simulation Models from Complex Embedded Real-Time Systems

Johan Kraft*, Joel Huselius and Christer Norström
Department of Computer Science and Electronics
Mälardalen University, Västerås, Sweden
{johan.kraft, joel.huselius, christer.norstrom}@mdh.se

Anders Wall
ABB Corporate Research
Västerås, Sweden
anders.wall@se.abb.com

## Abstract

*A modeling process is presented for extracting timing-accurate simulation models from complex embedded real-time systems. The process is supported by two complementary methods for tool-supported model extraction, Model Synthesis and Hybrid Model Extraction. The generated models enable impact analysis for complex real-time systems with respect to dynamic system properties, such as timing and resource usage. This can make software maintenance more predictable with respect to time-to-market and development costs, since timing errors can be identified early and avoided. The contribution of the paper is the modeling process, the Hybrid Model Extraction method and an interactive modeling tool, MASS, designed to support Hybrid Model Extraction of large implementations in C.*

## 1 Introduction

Managing the evolution of software systems is important in order to preserve the large investments associated with the software. Complex embedded software systems, such as industrial control systems or automotive systems, often consists of several million lines of code and are maintained over many years, sometimes decades, during which the software is exposed for changes continuously. Due to the many changes made over the years, the software systems become larger and more complex. As a consequence, the perspicuity of the system decreases, i.e., it becomes increasingly harder to predict how a change, e.g. a new feature, will impact the behavior of the system. Thereby, it becomes harder to maintain the software, which reflects in development cost and time-to-market. For embedded software systems, which often are multitasking real time systems, an impact analysis is especially difficult as a change's impact on the system's temporal behavior may cause timing errors.

We aim to enable impact analysis for complex embedded real time systems with respect to properties of the system's temporal behavior, such as response time and utilization of limited logical resources. The impact analysis should not only predict the extremes of the system's temporal behavior, but also predict the system's typical temporal behavior. This allows for predicting the performance impact of a change, an important issue for e.g. industrial robot control systems.

Our approach for impact analysis is based on probabilistic, timing-accurate models of the software, intended for simulation based analysis. By prototyping a proposed change on such a model and running simulations of the updated model, negative side effects of the change can be identified early. This saves a lot of time and money, since timing errors often are expensive. In general, they can not be detected until late phases of testing, when the complete system runs under realistic load conditions.

The accuracy of a simulation-based impact analysis can be very high when predicting the system's typical behavior, given that the model and the prototype implementation of the change on the model are correct and accurate. However, since simulation is not an exhaustive analysis method, it is not ideal for predicting worst case behavior. It can give estimations regarding the values and frequencies of extremes cases, but these are not guaranteed to be the actual worst case values.

An alternative to simulation are formal methods, like *model checking*, where models are analyzed exhaustively. There are several tools for model checking of timed automata models [1], which can safely determine if a model violates specific requirements on e.g. maximum response time. Examples are e.g. UppAal [8, 11, 28] and Kronos [9, 12, 18]. Model checking unfortunately suffers from an inherit problem known as the *state space explosion*, i.e. when state space of the model is too large to search, due to resource limitations (time or memory). For models of a complex industrial systems this would be a major problem.

Another alternative are the different analytical methods for response time analysis, e.g. rate monotonic analysis [6, 7, 19, 20, 26]. These are however too restrictive in their assumptions to be applicable on the systems that we have

*Formerly Johan Andersson

studied. Moreover, the expressiveness of the temporal models defined by these methods is not enough to capture the complex behavior that our targeted systems exhibit, e.g. interprocess communication, which would make an analysis extremely pessimistic.

Simulation based analysis should not be regarded as a replacement for rigorous methods such as model checking or analytical response time analysis. Simulation is more related to testing in the sense that it is about finding potential problems rather than giving guarantees. For many systems, simulation is the only feasible analysis method.

In previous work we have presented a modeling language, ART-ML, for probabilistic modeling of complex real time systems for the purpose of analyzing timing and resource utilization properties. We also presented supporting analysis tools: a simulator, and a query language, PPL, for examining the simulation results [31]. When we evaluated the approach of simulation based analysis on a real industrial system [29, 30] we realized that the main problem is how to obtain a valid model in a reasonable effort. This requires a modeling process that structures the modeling work and tool supported methods for model extraction and model validation. Tool support is absolutly necessary due to the vast size and the complexity of the type of systems considered. To perform the modeling manually is too tedious and error prone to be applicable in an industrial context. The tool support reduces the effort of modeling to an acceptable level and the resulting model will be of higher quality.

In this paper we propose a modeling process utilizing two complementary methods for tool-supported modeling; *Model Synthesis*, a fully automated method initially proposed in [16], complemented by a novel method named *Hybrid Model Extraction*. The paper also presents an interactive modeling tool, MASS, designed to support the method of Hybrid Model Extraction. This paper does not discuss methods for model validation, but we have discussed model validity in [3, 4, 2], which also contains references to works in the area of simulation model validity. Further investigation of model validation methods is part of future work.

Our work assumes systems based on real time operating systems (RTOS) with the possibility to record the task scheduling, i.e. the time and tasks of each context switch. This is possible in e.g. VxWorks, an RTOS from Wind River [33]. We assume that the tasks may be either event triggered or time triggered, may communicate with other tasks or external systems. The tasks may be servers that execute various *services* in response to incoming requests, from other tasks or other nodes. The services of a task may have very different functional and temporal behavior, but share a single thread of control. We assume online priority-driven scheduling, but make no assumptions on scheduling algorithm used, since we rely on simulation rather than analytical analysis methods.

The system domain that we are targeting is large and complex industrial control systems, e.g. industrial robot controllers and automotive control systems. We have two industrial partners in these domains:

- ABB - develops industrial robots and control systems for such robots - a complex real time system.
- Bombardier Transportation - develops trains, which contains distributed real time computer systems. This collaboration started 2006, which explains the focus on the ABB system in previous works.

The outline of this paper is as follows. Section 2 presents our approach for simulation model extraction, including the system modeling process (Section 2.1), Model Synthesis (Section 2.2) and Hybrid Model Extraction (Section 2.3). Section 3 presents a tool supporting Hybrid Model Extraction and in Section 4 references to related work is presented. Finally, Section 5 presents plans for future work.

## 2 Simulation Model Extraction

An impact analysis based on timing-accurate simulation requires an analyzable model of the system that describes both functional- and temporal behavior of the individual tasks on a proper level of abstraction. The model should focus on *task events* which significantly impact the temporal behavior of the system, i.e. what tasks that execute, in what order and for how long.

Examples of such task events are interprocess communication events, synchronization events and state change events that impacts the execution of other task events. Information regarding relevant task events can be obtained either from the source code (static analysis) or by analyzing data recorded from the running system (dynamic analysis).

To allow for timing-accurate simulation, the model also needs to describe the amount of CPU time required by a task to execute from one task event to the next, i.e. the execution time of the code between these model events. This is modeled not as a fixed value, but as an interval of possible execution times. The model also need to describe how the system's environment, e.g. other nodes, human operators, sensors readings etc., stimulates the system.

The desired model needs to be expressed in a rich modeling language, such as ART-ML, presented in a previous work [29, 30], or the modeling language used in the simulator Virtual Time, from Rapita Systems [23]. Such modeling languages are in essence imperative programming languages, extended with modeling primitives such as a global clock, which is advanced explicitly by the tasks in the model, and probabilistic selection. Such modeling languages also contain OS services such as task management

(start new task, change task priority etc), IPC communication (send message, receive message) and synchronization features (semaphores). The OS services of the modeling language need to have the same semantics as the corresponding OS services of the modeled system, which may limit the applicability of such modeling languages. The language we use, ART-ML, mimics the OS services of the VxWorks RTOS, from Wind River [33], while Virtual Time is adapted for OSE, an RTOS developed by ENEA [13].

## 2.1 The System Modeling Process

The extraction of a simulation model from a complex software system is far from trivial and requires a well-defined process that structures the modeling work and tool supported methods for the individual modeling activities. In this section we propose such a process, which utilize two complementary methods for the task modeling: *Model Synthesis* [16], for fully automated modeling of less complex tasks, and we propose a novel method, *Hybrid Model Extraction*, for modeling of complex tasks. In Hybrid Model Extraction, presented in Section 2.3, static analysis is used together with dynamic analysis in order accurately model tasks with complex behavior. The method can be automated to a large extent through appropriate tool support (see Section 3). The other method, Model Synthesis, is less suitable for modeling of tasks with complex behavior, since it relies solely on dynamic analysis. On the other hand it is fully automated, which makes it excellent for automatic modeling of less complex tasks, often numerous in a complex embedded system.

Based on previous experiences of modeling a complex embedded system [30], we propose the following process for modeling a complex embedded system, using Hybrid Model Extraction and Model Synthesis:

1. **Preparation** If not available, add timing recording functionality to system, typically in the form of a software recorder. The recording needs to capture the task scheduling, i.e. the context switches and calls to relevant OS functions, e.g. for interprocess communication, task priority changes or synchronization. It should also be possible to log variables in the application code, e.g. by inserting calls to the recorder (*probes*) at relevant locations in the application code. The recorder and any probes needs to be permanently added to the system and always be active, in order to avoid a *probe effect* [21] when removing or disabling them.

2. **Reference use-cases** Identify common system use-cases on which to focus the modeling and analysis. These *reference use-cases* can be obtained from e.g. system test specifications, end user documentation, or by asking experienced engineers at the company. The reference use-cases are necessary for all steps involving dynamic analysis, i.e. step 3, 5 and 6. Moreover, the reference use-cases are also of importance for the source code modeling, step 4, since they can be used to avoid modeling tasks or task services that are not executed in the reference use-cases.

3. **Model Synthesis** Make recordings of the system in the reference use-cases and apply Model Synthesis (Section 2.2) to automatically generate models for all tasks that has been observed in the recordings. Inspect the generated models and determine which tasks that are too complex for Model Synthesis and therefore should be modeled using Hybrid Model Extraction. A criterion to use is the number of probabilistic selections in a task model; models with fewer probabilistic selections can be used in the final simulation model while task models with many probabilistic selections should be modeled using Hybrid Model Extraction.

4. **Hybrid Model Extraction** Model the more complex tasks using Hybrid Model Extraction. The method is presented in Section 2.3.

   (a) **Source code modeling** Apply the source code modeling process described in Section 2.3.1 to one task at a time, starting with time-triggered tasks and tasks triggered by environmental stimuli. Continue modeling each task that is found to be triggered by a previously modeled task. In general, model only the task services (behaviors) that are executed in the reference use-cases to limit the modeling effort.

   (b) **Dynamic Analysis** Extract execution time data from the recordings made in step 3 for the tasks modeled using Hybrid Model Extraction. This can easily be automated through the appropriate tool-support. See Section 2.3.2 for more information regarding the dynamic analysis in Hybrid Model Extraction.

5. **Environment modeling** Model the environmental stimuli of each reference use-case based on the previously made recordings. The stimuli can be, e.g., commands from a human operator, messages from other external systems, interrupts caused by network traffic or I/O signals, or variations in input values from sensors. The environment can be modeled as tasks with highest possible priority that can preempt the simulation at any time in order to generate environmental stimuli. Such environment tasks should not consume CPU time, but should still describe the interarrival time of the stimuli events, e.g. through a loop containing an appropriate delay operation.

6. **Simulation Model Composition** Compose the submodels (task models and environment models) into a complete simulation model and validate the model by comparing execution traces recorded of the system with traces from simulations of the model. The traces has to be compared with respect to specific properties, e.g. observed response times of particular tasks or task services, or preemption patterns.

We have introduced a software recorder in the control system for industrial robots developed by our industrial partner ABB. We have also developed a supporting tool, the *Tracealyzer*, where recordings can be analyzed and vizualized [4, 3]. This does not only allow for model extraction but also facilitates general understanding and debugging of the software, a fact which was quickly realized by software engineers at the company. The recorder is today active by default in the system, also in the release version, and frequently used for debugging and performance testing. One recorded event requires 4 bytes of memory and around 0.8 $\mu$s of CPU time on a 500 MHz Intel Pentium III CPU, according to measurements [2]. This result in a CPU overhead of 0.4% and a data rate of 20.000 bytes/s at an event rate of 5 KHz, which is typical for the amount of code instrumentation proposed [2]. Thus, only 400.000 bytes is required for recording 20 seconds of execution, which is sufficient for both debugging and model extraction purposes. Since the systems considered have plenty of memory, 256 MB or more, using a few hundred KB is no problem.

## 2.2 Model Synthesis

Model Synthesis [16] is a fully automated method for obtaining models, where models are generated from recordings made on the system, i.e. a form of dynamic analysis. The method for Model Synthesis is complemented by an automated validation process [15] that ensures the recordings used are of sufficient length, number, and quality with respect to the system. The Model Synthesis process is computationally efficient; recordings containing thousands of task executions are synthesized in a few seconds. Although it may require an effort to perform the monitoring, models are obtained quickly enough not to interfere with continuous maintenance. We have investigated Model Synthesis on the robot control system from our industrial partner ABB. One of the conclusions made from that study is that, even though the models were reflecting the system operation properly in simulation based analysis, the models may be of a different structure than source code of the system. Model extraction based on source code is likely to produce more models more understandable to the human developer, in the sense that they are more similar to the source code. Moreover, to model tasks with more complex behavior, the method requires that the application code is instrumented to monitor state variables of importance for the temporal behavior. So far this has to be done manually, which may be a significant effort. Model Synthesis is however very useful for the modeling of less complex tasks, which are often numerous in complex embedded systems.

## 2.3 Hybrid Model Extraction

Hybrid Model Extraction relies on two information sources, both static and dynamic analysis. The static analysis models dependecies between tasks based on the source code, such as interprocess communication and shared state variables. In the dynamic analysis, recorded execution traces are analyzed in order to obtain empirical data on e.g. execution time (see Section 2.3.2). The Hybrid Model Extraction is better suited for modeling more complex tasks compared to Model Synthesis, but is on the other hand not fully automatic. The philosophy behind Hybrid Model Extraction is to automate as far as possible, but leave important modeling decisions regarding abstractions to an experienced software engineer with a deep understanding of the system. This ensures high-quality task models and by automating the time consuming and "boring" parts of source code modeling, a simulation model can be extracted from very large software systems in a reasonable effort.

When the modeled system is changed, to update the model it is only necessary to re-model the functions that have been directly changed and functions that become model-relevant as a sideeffect of changes in other functions.

If re-modeling is made on a regular basis, e.g. after each major release, the effort of maintaining the model is kept small, and even if the model is not updated for a long time, the effort of re-modeling the system is still reasonable. Howewer, as we have not yet evaluated this approach, we have no data on the effort required. An evaluation investigating this issue is part of future work.

The model of a task is extracted from the source code in an iterative process, where each iteration of the process adds additional details to the model. This makes the model larger and more complex, but also more accurate. To separate this process from the system modeling process presented in Section 2.1, we refer to this process as the *source code modeling process*. The process terminates when an iteration of the process results in a fix point, i.e. when the output equals the input. However, unless appropriate abstractions are made in the modeling, the process may not terminate. We refer to this issue as the *termination problem*. We intend to investigate the severity of this problem in future work when we evaluate the method on a real system.

### 2.3.1 Source Code Modeling

The source code modeling process is performed for one task at a time and is depicted in Figure 1, where the grey boxes

correspond to the activities and the white boxes represents information.

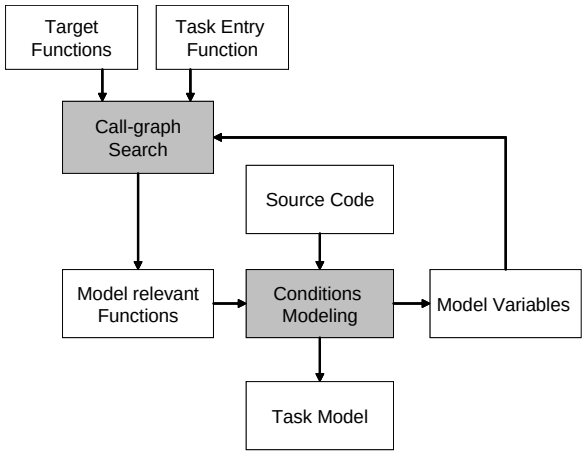


**Figure 1. The source code modeling process**

We define the source code modeling process formally in definitions 1-7 and Algorithm 1.

**Definition 1.** ***Target function*** *- A function that when executed may impact the task execution order or the utilization of logical resources.* □

Typical target functions are OS services for communication, synchronization services or task management (e.g. starting of new tasks, suspending tasks, or changing the scheduling priority of tasks). Another example of target functions are services for allocation/deallocation of application-specific resources.

**Definition 2.** ***Model variable*** *- A variable explicitly represented in the model.* □

**Definition 3.** ***Model event*** *- A program statement that*

- *calls a target function, or that*
- *assigns a model variable, or that*
- *calls a function containing at least one model event.*

□

**Definition 4.** ***Model-relevant function*** *- A function containing at least one model event.* □

**Definition 5.** ***Model-relevant condition*** *- A condition of a loop or a selection that determines the exection of a model event.* □

**Definition 6.** *The call-graph search is represented by the function*

$$F = cgSearch(s, T, V)$$

*, where $s$ is the entry function of the search, $T$ a set of target functions, $V$ a set of model variables. The function returns a set $F$ containing all functions reachable from $s$ that are model-relevant with respect to $T$ and $V$.* □

**Definition 7.** *The conditions modeling is represented by the function*

$$V = cModel(F)$$

*, where $F$ is a set of model-relevant functions. The function returns a set $V$ containing the model variables identified in the model-relevant conditions of the functions in $F$.* □

**Algorithm 1.** *Based on definitions 1 to 7, we define the source code modeling process as the algorithm*

1. $i = 0, V_0 = 0$
2. *repeat*
   - *(a)* $i = i + 1$
   - *(b)* $F_i = cgSearch(s, T, V_{i-1})$
   - *(c)* $V_i = cModel(F_i)$
3. *until* $Vi = V_{i-1}$
4. *Generate model based on $V_i$ and $F_i$*

*where T is the set of target functions, $F_i$ is the set of model-relevant functions after process iteration $i$ and $V_i$ is the set of model variables after process iteration $i$ .* □

Applying the source code modeling process requires 6 steps:

1. **Definition:** Initially, define the set of target functions, i.e. known OS functions that correspond to model events, such as IPC communication, synchronization, and task management services like starting a new task or changing a task's priority. The set of model variables is only empty in the first iteration when modeling the first task. It is never cleared since we like the set of model variables to be global for all tasks, since model variables might be shared between tasks.
2. **Call-graph Search:** The call graph of the task's entry function is searched for model events, i.e. calls to model-relevant functions or assignments to model variables. This corresponds to the $cgSearch$ function in the formal definition. We have used a commercial reverse engineering tool for this step, see Section 3.2. The output of this step is a set of model-relevant functions.
3. **Conditions Modeling:** Using an appropriate tool (see Section 3), construct a model of each model-relevant function. This corresponds to the $cModel$ function used in the definition.

(a) **Generate a model skeleton** by filtering the source code of the function with respect to the model events, by removing all statements that are not model events or encapsulate model events (i.e. a loop containing a model event should remain in the model).

(b) **Model the conditions** of selections and loops in the model skeleton manually. Appropriate tool support can facilitate this step by showing the corresponding source code, storing modeling decisions made (conditions and model variables) and in the end generate the resulting model of the function. A condition of a loop or selection can be modeled in three ways:

i. **As a logical expression** on variables included the model. If the variable is not in the model already, it is added to the model and the set of model variables used in the next process iteration.

ii. **As a constant value**, e.g. by modeling a loop using a fixed number of loop iterations or by modeling a selection condition as always true. This is useful abstraction for removing behavior that is out of scope for the model, like "unnecessary" error handling caused by defensive programming.

iii. **As a probabilistic expression**, which is useful when the underlying cause of the selection is out of scope for the model due to abstractions made. Probabilistic selections may be required in order to terminate the source code modeling process, but should not be used too often, as the model becomes less accurate. The "probabilities" are later derived from the dynamic analysis (see Section 2.3.2).

4. **Termination:** If no new model variables have been found, the source code modeling process has reached a fix point and is thereby finished, continue to step 5. Otherwise, the process is repeated from step 2, with the updated set of model variables. This corresponds to step 3 in Algorithm 1.

5. **Verification:** When all tasks have been modeled, verify that all assignments to model variables have been modeled by repeating the modeling process for each task using the latest version on the model variables list. This is necessary since the list of model variables have grown during the conditions modeling. Tasks may contain undiscovered assignments to model variables identified in later modeling of other tasks. However, the number of state variables shared between tasks is typically fairly low, so this step will only require a minor modeling effort.

6. **Composition:** Compose the function models into a task model. We see two alternative methods for this. A very simple solution is to write all function models to a single code file, in a format suitable for the simulator in mind, e.g. ART-ML. This will probably generate a lot of small function models, many containing just a call another model function, perhaps with a simple condition. Models looking like this may be hard to overview and comprehend. A more attractive, but more complex, solution is to compose small model functions into larger ones, to get a better overview. A technique for composition of function models if part of our future work. This step corresponds to step 4 of Algorithm 1.

### 2.3.2 The Dynamic Analysis

To enable timing accurate simulation, the model also needs to contain data on execution times, i.e. the amount of CPU time needed to execute from one model event to the next in the real system. The execution times are modeled as intervals rather than fixed values. This information is obtained using dynamic analysis, but information from worst case execution time (WCET) analysis could also be used to extend the execution time intervals up to a safe worst case estimate. If not all modeled tasks, or task services, are found in these recordings, there are two possible causes:

1. The recordings were not sufficiently long and/or did not stimulate the system according to the identified reference use-cases. This requires additional recordings to be made and/or an adjusted system configuration and/or an adjusted procedure for stimulating the system during the recording.

2. The source code modeling included tasks that never execute in the reference use-cases. In this case, either remove the extra task models and references to them in other task models, or add new reference use-cases that involves these tasks, if desired.

Since the recordings typically contain hundreds or thousands of jobs of each task, these intervals should be a good estimation of the typical temporal behavior in the reference use-cases. To introduce the execution time data in the model obtained from the source code modeling, a CPU-time consuming statement is inserted in the model (in ART-ML: *execute*) between each model event, with the execution time interval as parameter. This can easily be automated. When executing such a statement in a simulator, a value is picked from the interval and CPU-time is consumed accordingly. This is further described in e.g. [29, 30, 4]. The dynamic

analysis may also extract estimated probabilities for conditions modeled in a probabilistic manner. These probability estimations are based on the number of charateristic task events observed in the recorded execution trace. For instance, consider a task that "sometimes" sends a request $X$ to $TaskB$, where the actual mechanism is not desired to model in detail. If that particular task event is observed in 100 of 200 observed jobs, the corresponding probabilistic selection in the model would be given a 50 % probability of occurance.

### 2.3.3 An Illustrative Example

Next follows an example of the source code modeling process. In three iterations of the process we model a fairly uncomplicated task. The example is fictional but inspired by early experiences of applying the process on industrial embedded code.

In figures 2-6, the boxes represents model-relevant functions found in the source code, the arrows represents potential function calls and the annotations on the arrows are the condition that must be fulfilled for the call to take place. Gray indicates functions or function call that is removed during the conditions modeling. Bold style indicates newly discovered functions and function calls.

Figure 2 shows the result of the initial call-graph search, three call-chains are found from the task entry function to a model event, the target function *sendMsg*. Thus the model-relevant functions of iteration 1, $F_1$ are $\{A, B, C, D, E\}$.

In the conditions modeling, depicted by Figure 3, the variable *active* is removed from the $B - D$ condition as the modeler understands that the variable is always true in the reference use-cases. The $B - E$ call is also removed as it is identified as error handling outside the scope of this model. The remaining three variables (*StateX*, *cmd* and *CounterX*) are added to the list of model variables, i.e. $V_1 = \{StateX, cmd, CounterX\}$. This concludes the first iteration, and the next iteration is started by performing a new call-graph search, where assignments to the newly discovered model variables are considered model events as well.

Figure 4 depicts the result of the call-graph search in the second iteration. Four new model-relevant functions are discovered, in two call-chains; the call-chain $F - H$ may assign *StateX*, depending on the variable *init* and the call-chain $G - I$ may assign *CounterX*, depending on the variable *StateY*. The set of model-relevant functions is now $F_2 = \{A, B, C, D, F, G, H, I\}$.

The result from conditions modeling of $F_2$ is presented in Figure 5. The call-chain $F - H$ is removed since the variable *init* is found to be always false in the reference use-cases. When the call to $H$ is removed, $F$ no longer contains a model event and is therefore removed as well. The other

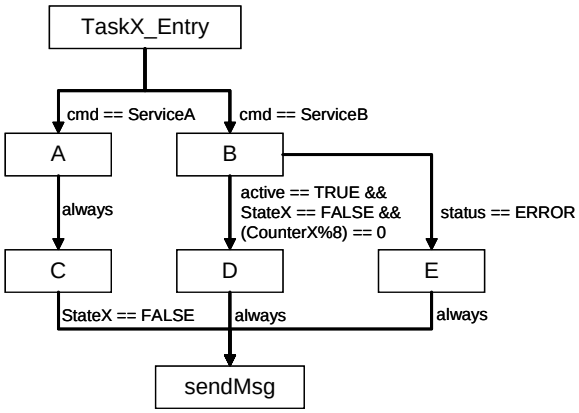


**Figure 2. Call-graph Search, Iteration 1**

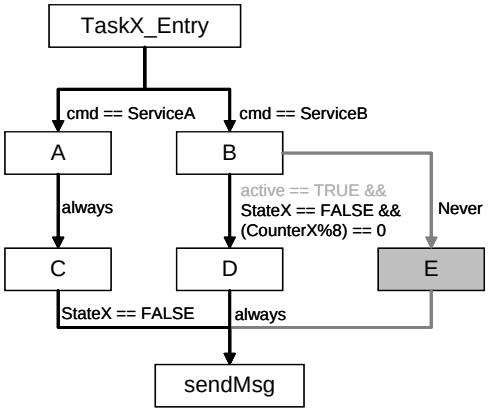


**Figure 3. Conditions Modeling, Iteration 1**

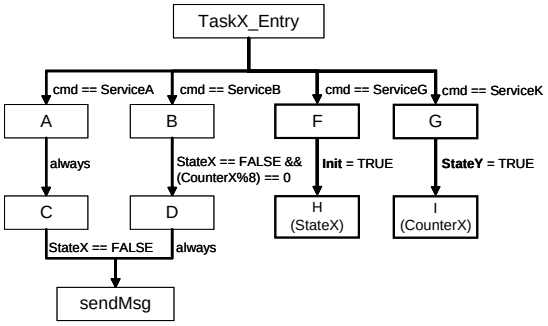


**Figure 4. Call-graph Search, Iteration 2**

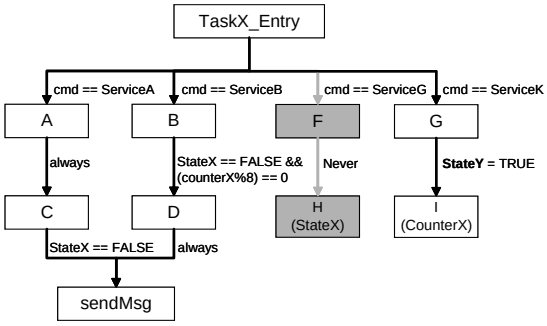


**Figure 5. Conditions Modeling, Iteration 2**

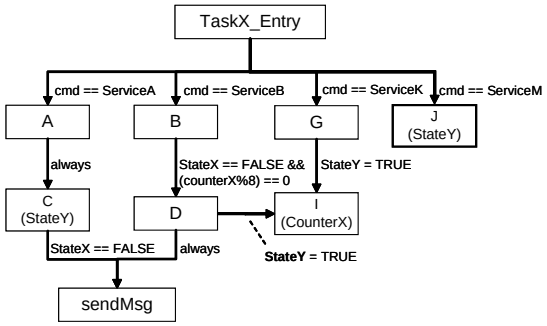


**Figure 6. Call-graph Search, Iteration 3**

new variable, *StateY*, is however added to the model variables, so $V_2 = \{StateX, cmd, CounterX, StateY\}$.

Finally, Figure 6 depicts the result of the third call-graph search. The difference compared to the call-graph search in iteration 2 (Figure 4) is the new model variable *StateY*. The call-graph search finds one new model-relevant function, J, that assigns *StateY*, so $F_3 = \{A, B, C, D, G, I, J\}$, and it is also discovered that the already modeled functions $C$ and $D$ contains new model events. An assignment of *StateY* was found in $C$ and $D$ may call $H$. These were not detected in previous iterations since they were not regarded as model events.

However, no new model variables were detected, so $V_3 = V_2$. Thus the process has reached a fix point and is thereby finished. Note that *StateX* is not assigned in this model. This could be the case if the variable is shared between several tasks. In that case, assignments to *StateX* will be discovered by the call-graph search when modeling other tasks, since the same list of model variables is used for all tasks. The function models are then composed into task models, according to step 6 of the source code modeling process. To obtain a complete simulation model, the timing information is finally inserted, as described in Section 2.3.2.

# 3 Tool Support for Hybrid Model Extraction

We are developing a tool for Hybrid Model Extraction, named MASS (Modeling Assistant). The MASS tool targets large implementations in C, consisting of millions of lines of code. C is commonly used for embedded systems including the systems of our industrial partners. The tool is still a prototype but we aim to release a first version during 2007. Currently, the tool only address the source code modeling process, proposed in Section 2.3.1, but we also intend to implement support for populating models with data from the dynamic analysis, as presented in Section 2.3.2.

## 3.1 The MASS Tool

MASS is implemented in Java and has a graphical user interface containing five main elements: Two code views, one for source code and one for the resulting model, and three lists: the model-relevant functions, the model-relevant conditions of the current function, and the global list of model variables. The code views have syntax highlighting which facilitates reading the code. The MASS tool is depicted in Figure 7.

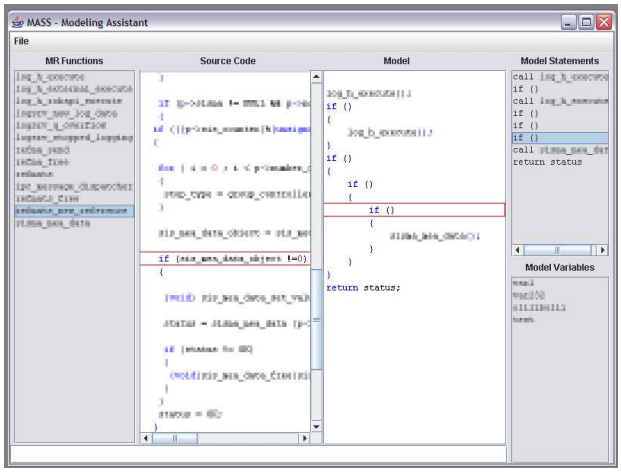


**Figure 7. The MASS tool**

The tool takes as input a list of references to model-relevant functions from a call-graph search (see Section 3.2). The model-relevant functions are parsed using a C parser, generated using the parser generator ANTLR [5]. For each model-relevant function the parsing generates an abstract syntax tree, which is filtered with respect to the model events. This result is model skeletons for each model-relevant function of the task. A model skeleton contains only the model events and selections or loops that encapsulate model events. The selections and loops of the

model skeleton contain no information regarding the conditions, these are instead manually modeled after the parsing is complete. The user can select each condition from the list and enter a suitable condition, based on the source code view that is visible in parallel. When selecting a condition from the conditions list, the tool focuses on the corresponding location in the source code and model code. Thereby, the modeler does not have to search the source code for the statements corresponding to the model-relevant conditions. When editing a condition, the modeler can add any variables used in the condition modeling to the list of model variables. The list of model variables can be written to a file, for use in the call-graph search.

### 3.2 Call-graph Searching

The MASS tool depends on an external tool for performing the call-graph search, which takes as input an entry function, set of target functions and a set of model variables, and returns a set of model-relevant functions, as presented in Section 2.3.1. We have used a commercial tool for this part, *Understand for C++* from Scientific Toolworks [25], which we have customized to our needs. Understand has a graphical user interface that provides code analysis, navigation, advanced searching and various visualizations of code, such as call-graphs. We evaluated two similar tools before selecting Understand, *Imagix 4D*, from Imagix [17] and *CodeSurfer*, from GrammaTech [10]. There were several reasons for selecting Understand for this task. It coped much better with large amounts of code than the other tools (we analyzed some 700.000 lines of C code without problems), it was relatively cheap for this type of tool (a license is 495 USD), and had a nice well-documented API for making custom extensions using Perl-scripts. A 30-day evaluation licence is available free of charge.

## 4 Related work

There are a number of methods available for model generation. They can be divided based on input; some use code, some use recording, some use manual input (e.g. documentation), and some use a mix of all or some of the above. Of course, due to the information content of the inputs, the completed model is limited by its inputs. Also, due to the different levels of flexibility in the available inputs, the portability may be an issue (a method that requires code in C++ as input can probably not model a system implemented in an assembly language). A method proposed by Holzmann and Smith [14] derives verification models from ANSI C code and some well-defined documentation. The result is a Promela model can then be analyzed using SPIN, a widely used tool for model checking. However, SPIN and Promela doesn't have a concept of time, so properties such as response time can not be analyzed. Sifakis et al. [24] propose a method that uses tagging of the real time software with time constraints to facilitate automated modeling based on the code as input. Yan et al. [34] present DiscoTect, a tool that can obtain high-level architecture models from recordings of system implemented in Java. Moe and Carr [22] present automated modeling from recordings of RPC calls in CORBA-based legacy systems. The tool has reportedly helped discover and identify a number of bugs in an operation and maintenance system for cellular networks by Ericsson. Program slicing [32], is a technique for extracting a subprogram containing only the statements of a program that affects the specified variable(s). An overview of program slicing techniques can be found in [27]. The source code modeling process proposed in this paper reminds of program slicing, in the sense that a subprogram is extracted, a model, but is different in the sense that instead of extracting a subprogram based on variables, the source code modeling process generates a subprogram (the model) based on a set of relevant functions. Even though variables are used during the source code modeling process they are an output rather than an input.

## 5 Conclusions and Future work

This paper have proposed a process for modeling complex industrial real time systems, for the purpose of enabling simulation based impact analysis of timing and resources usage. The process relies on two methods for extracting models of individual tasks, Model Synthesis and Hybrid Model Extraction. Model Synthesis have been proposed and evaluated in previous work [15], while the modeling process and the method of Hybrid Model Extraction are novel contributions of this paper, together with a tool for Hybrid Model Extraction, named MASS.

The single most important part of future work is an industrial evaluation of the modeling process, especially the method of Hybrid Model Extraction. We intend to apply our modeling process and methods on the systems of our industrial partners (ABB and Bombardier Transportation), during 2007. We plan to extract models from the systems using the proposed modeling process and use these to make predictions regarding real changes that are to be implemented, e.g. new features. The experiences from using Hybrid Model Extraction will be very valuable for our future research in simulation model extraction. We intend to investigate the accuracy of the impact analysis by comparing the predicted impact with the actual impact, which is obtained from measurements of the running system when the change have been implemented. Important parts of future work are also methods for model validation and conditions modeling with a higher degree of automation.

## References


[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] J. Andersson. Modeling the temporal behavior of complex embedded systems - a reverse engineering approach. Licentiate thesis, Mälardalen University, http://www.mrtc.mdh.se, June 2005.

[3] J. Andersson, A. Wall, and C. Norström. Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings. In *Proceedings of the First International Symposium on Leveraging Applications of Formal Methods (ISoLA '04)*, 2004.

[4] J. Andersson, A. Wall, and C. Norström. A framework for analysis of timing and resource utilization targeting complex embedded systems. In *ARTES - A Network for Real-Time research and graduate Education in Sweden, Editor: Hans Hansson*, pages 297–329. Uppsala University, 2006.

[5] ANTLR website, http://www.antlr.org.

[6] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[7] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, , and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems Journal*, 8(2/3):173–198, 1995.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[9] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[10] GrammaTech website, http://www.grammatech.com.

[11] A. David and W. Yi. Modelling and analysis of a commercial field bus protocol. In *Proc. of 12th Euromicro Conference on Real-Time Systems*, pages 165–172. IEEE Computer Society Press, 2000.

[12] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 66, Washington, DC, USA, 1995. IEEE Computer Society.

[13] ENEA website, http://www.enea.com.

[14] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *Transactions on Software Engineering*, 28(4):364–377, April 2002.

[15] J. Huselius, J. Andersson, H. Hansson, and S. Punnekkat. Automatic generation and validation of models of legacy software. In *Proceedings of the 12:th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, August 2006.

[16] J. G. Huselius and J. Andersson. Model synthesis for real-time systems. In *Proceedings of the 9:th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 52–60, Manchester, UK, March 2005.

[17] Imagix website, http://www.imagix.com/ .

[18] Kronos website, http://www-verimag.imag.fr/temporise/kronos.

[19] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS '90)*, pages 201–212, December 1990.

[20] P. K. P. M. Joseph. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[21] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[22] J. Moe and D. Carr. Using execution trace data to improve distributed systems. *Software - Practice and Experience*, 32(9), July 2002.

[23] Rapita systems website, http://www.rapitasystems.com.

[24] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111, January 2003.

[25] Scientific Toolworks, http://www.scitools.com.

[26] K. Tindell. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Technical Report YCS189, Dept. of Computer Science, University of York, United Kingdom, 1992.

[27] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[28] Uppaal website, http://www.uppaal.com.

[29] A. Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. PhD thesis, Mälardalen University, Sweden, 2003.

[30] A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*, 2003.

[31] A. Wall, J. Andersson, and C. Norström. Probabilistic Simulation-based Analysis of Complex Real-time Systems. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing*, 2003.

[32] M. Weiser. Program slicing. In *In the Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[33] Wind River website, http://www.windriver.com.

[34] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 2004 International Conference on Software Engineering*, May 2004.