Probabilistic Simulation-based Analysis of Complex Real-Time Systems

Anders Wall and Johan Andersson
Department of Computer Engineering
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
{anders.wall,jan98053} @mdh.se

Christer Norström
ABB Robotics
Västerås, Sweden
christer.e.norstrom@se.abb.se

Abstract

Many industrial real-time systems have evolved over a long period of time and were initially so simple that it was possible to predict consequences of adding new functionality by common sense. However, as the system evolves the possibility to predict the consequences of changes becomes more and more difficult unless models and analysis method can be used. Moreover, traditional real-time models, e.g., fixed priority analysis, may be too simple for accurately capturing a complex system's characteristics. For instance, assuming worst-case execution time may not be realistic. Hence, analyses based on these models may give an overly pessimistic result.

In this paper we describe our approach to introducing analyzability into complex real-time control systems. The proposed method is based on analytical models and discrete-event based simulation of the system behavior based on these models. The models describe execution times as statistical distributions which are measured and calculated in the existing system. Simulation will not only enable models with statistical execution times, but also correctness criterion other than meeting deadlines, e.g., non-empty communication queues. The simulation result is analyzed by specifying properties in a probabilistic property language. The result of such an analysis is either of probabilistic nature or boolean depending on how the property is specified. Having accurate system models enable analysis of the impact on the temporal behavior of, e.g., customizing or maintaining the software.

1 Introduction

Large and complex distributed real-time computer systems usually evolve during a long period of time. The evolution includes maintenance and increasing the system's functionality by adding new features. Eventually, if it ever existed, the temporal model of the system will become inconsistent with the current implementation. Thus, the possibility of analyze the effect of adding new features with respect to the temporal behavior will be lost. For small systems this may not be a big problem, but for large and complex systems the consequences of altering the implementation cannot be foreseen. Introducing, or re-introducing, analyzability becomes the task of re-engineering the system and constructing an analytical temporal model of it.

The work presented in this paper is the result from an activity where we tried to re-introduce temporal analyzability in a robot control system at ABB Robotics. In essence, the controller is object-oriented and consists of approximately 2 500 000 LOC divided into 400-500 classes organized in 15 subsystems. The system contains three nodes that are tightly connected: a main node that in essence generates the path to follow, the axis node that controls each axis of the robot, and finally the I/O node that interacts with external sensors and actuators. In this work we have studied a critical part in the main node with respect to control. Details about the case study can be found in [1].

Initially, we tried to apply traditional real-time analyses. However, applying classical real-time models and analyses to large and complex systems, *e.g.*, fixed priority analysis (FPA) [4] [3] [6], often results in a too pessimistic picture of the system due to large variations in execution times and semantic dependencies among tasks. FPA is based on the fact that if a set of tasks, possible periodic with worst case execution times (WCET), and deadlines less than or equal to their periods, is schedulable under worst-case conditions then it will always be schedulable. The result from such an analysis is of a binary nature, *i.e.*, it does not give any numbers on probability of failure, it just tell if the system is guaranteed to work or not. In this work, the result from an FPA would be negative, *i.e.*, assuming worst-case scenarios, the system will not be temporally correct in terms of meeting all its deadlines. FPA assumes a task model where deadlines are assigned to every task. In the robot

controller we have investigated is the temporal correctness defined in terms of other criteria. Some of the tasks can have their deadlines derived from these criteria, but not all tasks can easily be assigned a deadline. An example of another correctness criterion is a message queue that must never be empty.

Furthermore, a task may execute sporadically and with great variations in execution times. To be safe in an FPA, the periodicity of sporadic tasks is modeled as having a frequency equal to the minimum inter-arrival time. Using the worst-case scenario in terms of both execution time (maximum) and periodicity (minimum), is not sufficient as the result would be to pessimistic.

Since traditional temporal models and analysis do not apply to the class of systems we have studied, we have used a simulation-based approach. In this paper we describe our approach to analysis of complex real-time system's temporal behavior. The simulations are based on analytical models of the system made in our modeling language ART-ML (Architecture and Real Time behavior Modeling Language). By using simulations, we can define other correctness criterion than satisfying deadlines as mentioned before. Instead of always assuming worst-case scenarios, we can use execution time distributions. ART-ML also permits the behavior of tasks to be modeled, *i.e.*, on a lower level than the software architecture. This permits a more precise model to be created as semantic relations among tasks can be introduced.

The tool suit, in which the simulator is a part, also includes tools for measuring an existing system implementation, as well as tools for processing measurements and analyzing the results generated by the simulator. The analysis is based on probabilistic properties. Temporal requirement are specified in a query language, the *probabilistic requirement property language*. The result of such a query is the probability of complying with a temporal requirement.

The introduction of a analyzable model of a system brings a continuous activity of maintaining the model. The model has to be consistent with the current implementation of the system, *i.e.*, the implementation should be a true refinement of the model. Consequently, our method must be an integrated part of a company's development process. In this section we will briefly describe the activities associated with the analytical model. Figure 1 depicts the general activities required in our method. Note that the process described here only concerns the method we are proposing. Important activities such as verification and validation of the implementation are omitted.

The first activity in making an existing system analyzable with respect to its temporal behavior is re-engineering of the system. Typically, the re-engineering activity includes identifying the structure of the system, measuring the system, and populating the model. By comparing the result

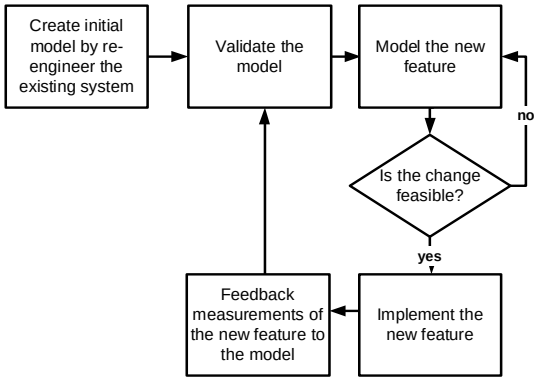

Figure 1. The process of constructing and maintaining an analyzable system.

from analyzing the system using the analytical model with the temporal behavior of the real system confidence in the model can be established.This is exact the same procedure as used in developing models for any kind of systems.

As the system evolves, each new feature should be modeled and the impact of adding it to the existing system should be analyzed. This enables early analysis, *i.e.*, before actually integrating the new feature into the system. Detecting flaws at an early stage is often more cost effective than discovering the problem late in the testing phase of the development process. Note, that such an approach requires a modeling language that support models on different level of abstractions. ART-ML has this property which will be further described in Section 2. Modeling of new features should be part of the company's design phase.

Finally, when the new feature has been implemented and integrated into the system the model of that feature can be refined by feeding back information from the implementation into the model. Hence, a more précised model is implemented. This activity is typically performed in conjunction with the verification phase of a company's development process.

We have studied other simulators such as STRESS and DRTSS. The STRESS environment is a collection of CASE tools for analyzing and simulating behavior of hard real-time safety-critical applications [2] STRESS is primarily intended as a tool for testing various scheduling and resource management algorithms. It can also be used to study the general behavior of applications, since it is a language-based simulator. STRESS has neither support for modeling distributions of execution times or memory allocation nor a requirement language.

Another simulation framework is DRTSS [9], which allows its users to construct discrete-event simulators of complex, multi-paradigm, distributed real-time systems. The

DRTSS framework is quite different from STRESS, although they are closely related. DRTSS has no language where the behavior can be specified. A language that describes the behavior of components is necessary for achieving the goals of our work and excludes DRTSS as a possible solution. In this work there is no tool or language for specifying and analyzing requirements automatically.

In [7], an analytical method for temporal analysis of task models with stochastic execution times is presented. However, sporadic tasks cannot be handled. A solution for this could not easily be found. Without fixed inter-arrival times, *i.e.*, in presence of sporadic tasks, a least common divider of the tasks inter-arrival times can not be found.

The outline of this paper is as follows: Section 2 describes our approach to measure the existing system, build analytical models based on those measurements, using the analytical models for simulating the system's temporal behavior and specifying probabilistic properties that is to be analyzed. Both the modeling language and the probabilistic property language is described. Finally Section 3 concludes the paper and gives indications of future work.

2 The method

When creating an initial model, M_0, of an existing system, S_0, several distinct activities which are depicted in Figure 2, are required. First the structure has to be identified and modeled, *i.e.*, the tasks in the system and synchronization and communication among them. In the next step, we measure the system and populate the structural model with data about the temporal behavior. Moreover, information needed in the validation phase is collected, *e.g.*, response times. When tuning the model we simulate the initial model and compare the results with the validation data collected in the previous step. In this step we may have to introduce more details about the tasks behavior in order to capture the system's behavior accurately. There is a potential risk that we cannot model the system's behavior without introducing too many details. For instance, there are so many implicit relations among the tasks that we can not make a valid model without modeling the complete behavior of the tasks involved. This, however, unveils the complexity of the existing architecture. Consequently, the solution is rather to redesign the complex architecture. Up until this point, the work of making a model is quite straightforward.

To validate the usefulness of the model we have to perform a sensitivity analysis. The sensitivity analysis should be based on foreseen potential changes in the particular system. In the system we have studied the following typical changes were identified:

- change existing behavior of a task which results in changes in the execution time distribution
- add a task to the system
- change the priority of an existing task

By introducing the changes in the model as well as in the system and comparing their behavior, we can increase the confidence in the created model. Any divergence between the behavior of the simulated model and the system indicates that more details must be introduced in the model. For instance, a change of the execution time in a task may result in a time-out for another task that waits for a semaphore. This could indicate that the semaphore behavior has to be introduced in the model as well.

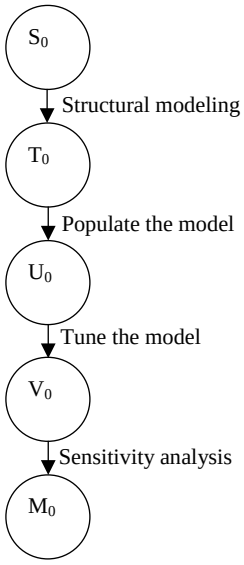

Figure 2. The process of constructing a model.

Moreover, the accuracy of the model is dependent on the quality of the measured data. The measuring of the data should affect the system as little as possible. Too big probe effect on the system will result in an erroneous model and might cause wrong decisions regarding future developments.

A suitable notation is necessary for creating a system model. The language has to support both the architecture, *i.e.*, nodes, tasks, semaphores, message queue, and the behavior of the tasks in different levels of abstractions. It should be possible to compare the behavior of the created model with the target system in an easy way in order to iteratively improve the model to satisfactory level, illustrated in figure 3.

Our approach to analysis of the temporal behavior is simulation since our notation not only describes the architecture of the target system, but also the behavior of the included

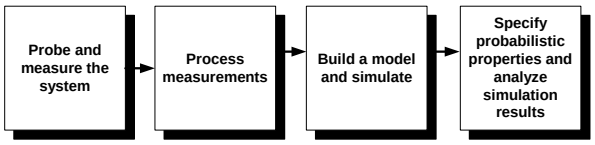

Figure 3. The work flow of making an analytical model

tasks. Simulation allows execution times to be expressed as distributions. We analyze the output from the simulator by defining properties of interest. An example of such a property is the probability of missing a deadline requirement on a task. Moreover, the simulation approach allow us to define non-temporal related properties, *e.g.*, non-empty message queues.

2.1 Measuring and processing data

Measuring data in a software system requires the introduction of software probes if no hardware probes are used [8]. The data of interest is resource utilization, *e.g.*, task execution times, memory usage or sizes of messages queues. We used software probes in order to log task switches and message queues. The measured data is stored in static allocated memory at runtime, in binary format. All formatting of the output is done off-line, writing to a file at runtime is too time consuming. This minimizes the probe effect, *i.e.*, the part of the execution time that is caused by the probe.

The output from the system is a text-file containing task switches, time stamps, and the number of messages in different queues. The size of the output can be very big, several hundred kilobytes per monitored second of execution. To manually analyze that data for developing a model would be too time-consuming. We have therefore developed a tool that extracts data from a log and computes the statistical distribution of each task's execution time. In table 1 the result of processing data from a task is shown.

In order to calculate the statistical distribution for a set of execution times for a task, we divide all execution times into *instance equivalence classes* (IEC). Formally we define an IEC as:

Definition 1. *An instance equivalence class IEC is a subset of execution time instances of a task E, $IEC \subset E$, defined by its upper bound $\max(IEC) \in E$ and its lower bound $\min(IEC) \in E$ and a threshold that specifies the interval between $\max(IEC)$ and $\min(IEC)$.* □

A task instance's execution time is a member of the IEC I_n iff it is larger or equal to $min(I_n)$ but less or equal than $max(I_n)$. In the model are all instances in a IEC represented as the average execution time of the IEC which have the probability of occurrence equal to the number of instances in the IEC divided by the total number of measured instances for a task. For example, consider the first entry in table 1 which express that, with the probability of 61.5 %, is the execution time for the task 360.097 time units. Consequently, the execution time of tasks in our method is represented as a set of pairs consisting of the average execution time of an IEC and its probability of occurrence.

Definition 2. *The execution time for task t, t.exe, is a set of pairs, $\langle iec, p \rangle$ where iec is the average execution time of an IEC and p is its probability of occurance.* □

An algorithm was developed to automatically identify the boundaries $\min(I)$ and $\max(I)$ for all IECs given a set of execution times for a task and a threshold. The algorithm is recursive. Initially all instances are sorted by their execution time using the quicksort algorithm. The sorted list constitutes the initial IEC, I_0 for the task. Next, the largest difference in execution time between two adjacent instances in the sorted list is located. If the largest difference is larger than a specified *threshold*, the list I_0 is split into two new IECs and recursive calls are conducted with each of the two new IECs. Consequently, the threshold specifies mathematically how big variations there can be between two consecutive execution times belonging to the same IEC. From the system modeling point of view the threshold has two purposes. First, it can be used to filter small variations in execution times due to cache memories or branch prediction units, *i.e.*, independent from the control-flow. Moreover, threshold can also specify the level of abstraction with which the temporal behavior is modeled. A large threshold results in a more coarse-grained distribution, *i.e.*, less number of IECs for a task.

Below the equation for finding distinct IECs, given a set of sorted execution times and a threshold, is displayed.

$$\begin{aligned} &\forall \langle x_i, x_{i+1} \rangle \exists \langle x_j, x_{j+1} \rangle \in I_0 : \\ &\quad abs(x_j - x_{j+1}) > abs(x_i - x_{i+1}) \wedge \\ &\quad abs(x_j - x_{j+1}) \geq threshold \wedge i \neq j \end{aligned}$$

As a result from applying the equation above for the first time on a sorted set of execution time instances we may get two new potential IEC, I_k and I_{k+1} where $\min(I_k) = \min(I_0)$, $\max(I_k) = x_j$, and $\min(I_{k+1}) = x_{j+1}, \max(I_{k+1}) = \max(I_0)$. If no gap is found greater than the threshold, the final IEC is already found and the recursion is stopped. When the recursion is stopped, the largest and the smallest execution time in the list is considered to define the boundaries of an IEC.

This approach worked well with the characteristics of our data. However, the distance between min and max in a IEC could be quite big if no gap greater than the threshold

Min time	Max time	Average time	n	n/N
287.265	420.876	360.097	131	61.5%
577.448	604.320	590.884	2	9.4%
4176.659			1	4.7%
4797.058	5024.122	4911.885	12	5.6%
5177.941	6829.881	5829.924	65	30.5%
11962.947			1	0.47%
12814.769			1	0.47%

Table 1. An example of statistical distribution of a task. $N = \sum n$, were n is the number of instances in an IEC.

is found in the sorted list of execution times. Theoretically, all measured execution times may end up in the same IEC. We have three possible solutions for such a scenario:

- Reduce the threshold and try again
- Do not create any IECs (threshold = 0), use the entire set of instances and assign each of them the probability of $\frac{1}{no.of instances}$. This solution results in a very detailed model
- Model such a task as a linear distribution with a max- and a min execution time and uniformly assign probabilities in between them.

The measured data can also be graphically visualized in a chronological order, see Figure 4. Studying such a graph may reveal executional dependencies among tasks. Introducing those dependencies will make the model more accurate with respect to the implemented system as they reduce pessimism.

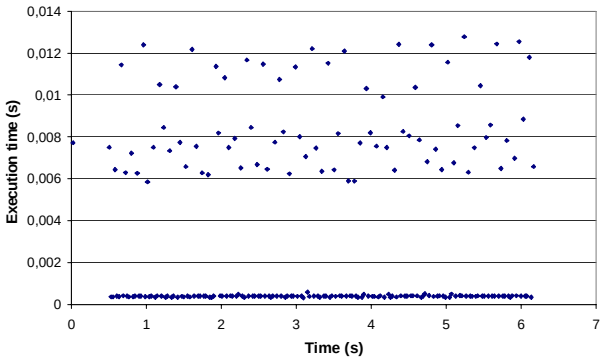

Figure 4. An example of measured execution times

2.2 The ART-ML language

The notation developed, ART-ML, is composed of two parts, the *architecture model*, and the *behavior model*. The architecture model describe the temporal attributes of tasks, *e.g.*, period times, deadlines, priorities. The architecture model also describes what resources there are in the system.

The behavior model describes the behavior of the tasks in the architecture model. Thus the behavior is encapsulated by the architecture model. The behavioral modeling language is an imperative, Turing-complete language close to Basic and C in its syntax.

```
mainbox TASK_C_MAILBOX 4;
mainbox TASK_C_MAILBOX 6;

const msgcode_ref_request 1001;
const msgcode_ack 1002;

task APERIODIC_TASK_C
  trigger mailbox TASK_C_MAILBOX
  priority 2

behavior{
  variable incoming;
  incomming = 0;

  recv(incoming, TASK_C_MAILBOX)
    timeout 100;
  if (incoming == msgcode_ref_request){
    recv(incoming, TASK_C_MAILBOX)
      timeout 10000;
    execute((60,6200),(40,6750));
    send(TASK_B_MAILBOX, msgcode_ack);
  }else{
    chance(80){
      execute((63,400),(37,470));
    }else{
      execute((100,1000));
    }
  }
}
```

Two constructs make ART-ML unique compared to other modeling languages that has been studied: the *execute-statement* and the *chance-statement*.

The execute statement describe the partial execution time of the code in the target system, *i.e.*, the execution time for a complete task or part of a task. The execution time for a task is represented by a statistical distribution. A probability distribution is implemented as a list of pairs that corresponds to the calculated IECs described in Section 2.1. Every pair has a probability of occurrence and an execution time. When a task performs an “execute” it supplies a probability distribution as parameter. An execution time is picked according to the distribution and the task is put into “executing state”.

When a task has been allowed to execute for that amount of time, the next statement, if any, in that task's behavior description is executed. In the example below, the execute statement will execute 10 time units with the probability of 19 % and 56 time units with the probability of 81 %:

```
execute((19,10), (81, 56));
```

The chance statement implements a stochastic selection. Stochastic selection is a variant of an IF-statement, but instead of comparing an expression with zero, the expression is compared with a random number in the interval [1-100]. If the value of the expression is less than the random number, the next statement is executed. If not, the else-statement is executed if there is one. Stochastic selection is used for mimic tasks behavior observed as a black box. For instance, we can observe that a task sends a message to a particular queue with a certain probability by just logging the queue. This can be model with stochastic selection such that we send a message with the observed probability. For instance, it is possible to specify that there is a 19 % chance of sending a message:

```
chance(19)
   send(mbox1, msg)
```

The language has also support for message passing through the primitives *send* and *recv*. Both can be associated with timeouts. Moreover, binary semaphores can be specified in ART-ML through *semtake* and *semgive*. Semtake can be used in combination with a timeout as well.

2.3 Modeling on different levels of abstraction

When creating a model of the tasks in the target system, a level of abstraction has to be chosen. That level defines the accuracy of the model. The lower the abstraction level, the more detailed and accurate model. There is no point in using the lowest possible level of abstraction, *i.e.*, a perfect description. In that case, the actual code could be used instead. Using an extremely high level of abstraction results in a model that is not very accurate and therefore of limited use. The best result is something in between these two extremes.

In the ART-ML language, very detailed models of task can be made, theoretically perfect ones. By describing blocks of code only by their execution time, *i.e.*, an execute-statement in the model), the abstraction level is raised to a higher level. The more code that is described by an execute-statement, the higher level of abstraction. The highest abstraction-level possible is if all code of the task is described using a single execute statement.

It is possible to use any level of abstraction when describing a task using the ART-ML language. It is therefore possible to describe different tasks at different levels of abstraction. This property of the language allows the model to be improved (in terms of level of detail) task by task.

The execution time distributions used also have different levels of abstraction. The measured data from the target system is somewhat filtered when creating the distributions. The recorded instances are grouped into equivalence classes. This causes data to be lost. The level of abstraction is in this case the number of intervals used to describe the execution time of the task. This level of abstraction impacts the accuracy of the model.

If there are multiple tasks in the system that is of no interest and do not affect the behavior of other tasks, they can be modeled as a single task at maximum abstraction level, *i.e.*, only by a single execution-time probability distribution. This reduces the complexity of the model without affecting the accuracy of the result regarding the tasks of interest. However, it is required that all tasks in a group has the same or adjacent priorities. Moreover, tasks can only be grouped in such a way that no other modeled task, *i.e.*, not part of the group, has a priority within the range of a group. For instance, consider a composed task consisting of two task, *a* with high priority, and *c* having low priority. Moreover, consider task *b* which is also part of the system and runs at mid priority. Task *a* should be able to preempt task *b*, but not task *c* should not. Thus, the composed task has to run on different priorities in order to reflect the control flow of the implemented system. We refer to such a group of tasks as a *composed task*.

Formally we can express the rules of grouping tasks into composed tasks, *i.e.*, assigning execution time distribution, period time and priority, in a way that preserves the utilization of the CPU which the tasks in the group contributes. First the set of tasks to compose, *C*, have to be normalized with respect to the period times. The composed task will run with the shortest period time among the participating tasks. Consequently, the period time of the composed task *c* is:

$$c.T = \min_{t \in C}(t.T)$$

Normalizing the tasks in such a way that the CPU utilization is preserved requires re-calculating the execution times for all IECs described in Section 2.1, for all tasks in *C*.

$$\forall t \in C \forall i \in t.exe : \frac{c.T}{t.T} i.iec$$

The resulting execution time distribution for the composed task is obtained by calculating the cartesian product, *V*, of all *t.exe* where $t \in C$, *i.e.*, $t_1.exe \times t_2.exe \times ... \times t_n.exe$. Every n-pair which is part of the cartesian product corresponds to an executional scenario. For instance, $\langle x_1, x_2, ..., x_n \rangle$ corresponds to the scenario where task 1

executes for $x_1.iec$ time units, task 2 executes $x_2.iec$ time units, and so on.

$$c.exe = \{\langle iec, p\rangle | \forall v \in V : iec = \sum_{\forall j \in v} j.iec \wedge p = \prod_{\forall j \in v} j.p\}$$

The final c.exe is obtained by merging pairs in c.exe that have equal iecs (cmp. the generation of IECs described in Section 2.1). For the set of pairs, $\{\langle iec, p_1\rangle, ..., \langle iec, p_n\rangle\} \subseteq c.exe$, of all pairs having the same execution time, the merged pair remaining in c.exe is $\langle iec, \sum_{i=1}^{n} p_i\rangle$, where $\sum_{i=1}^{n} p_i$ is the probability that task *c*, executes iec time units.

Finally, the priority of the composed task *c*, *c.p*, is assigned the maximum priority of the tasks participating in the composition.

$$c.p = \max_{forallt \in C}(t.p)$$

As an example consider the composition of two tasks: *a* and *b*. Task *a* executes with the distribution *a.exe*={(1,0.75),(2,0.25)}, and *a.T*=10. Task *b* executes with the distribution *b.exe*={(2,0.5),(3,0.5)} and *a.T*=5. Normalizing the execution of task *a*, *i.e.*, *a.exe*={($1\frac{5}{10}$,0.75), ($2\frac{5}{10}$,0.25)} gives the cartesian product, *V*, equal to ((0.5,0.75),(2,0.5)),((0.5,0.75),(3,0.5)),((1,0.25),(2,0.5)),((1,0.25),(3,0.5))}. The cartesian product *V* results in a execution time distribution for the composed task, *c.exe* equal to {(2.5,0.375),(3.5,0.375),(3,0.125),(4,0.125)}, *c.T* = 5.

The assignment of temporal attributes to composed tasks described above is a coarse approximation of the system behavior. Ideally, all tasks are modeled individually. However, in order to limit the modeling effort, and to prune the state space, such approximations can be practical. The result from the case study presented in [1][10], indicates that the use of composed tasks is quite adequate. The result of applying the proposed rules may lead to situations where execution times are longer than the period time. This corresponds to a possible system overload in the implementation.

2.4 Simulating the system behavior

The simulation-based approach used in this work allows correctness criterion other than meeting deadlines. An example of other correctness criterion could be the non-emptiness of a certain message-queue. The system studied in this work had this criterion. If a certain message-queue got empty, it was considered a system failure.

Simulation also allows us to specify arbitrary system cycles. FPA assumes cycles equal to the Least Common Multiple of the period times in the task set (LCM). However, there exists systems such as the robot controller investigated as part of this work, where the cycle times are determined by other criterion. For instance, in the robot case, the system cycle is determined by the robot application, *i.e.*, the cycle time of the repetitive task of robot which it is programmed to do.

When designing the simulator, two different approaches were identified. The most intuitive was to let the simulator parse the model and execute it statement by statement. The other approach was to create a compiler that translated the high level ART-ML model into simple instructions and construct the simulator as a virtual machine that executes the instructions. A test was made to compare the performance of the two approaches based on two prototypes. The virtual machine solution performed significantly better which is crucial for an analysis tool.

The simulator engine is based on three parts, the *instruction decoder*, the *scheduler* and the *event-processing*. The instruction decoder executes the instructions generated by the compiler, *i.e.*, it is the virtual machine. Some of the instructions generate events when executed, *e.g.*, execute, send, semtake. The simulator engine acts upon the generated event, *e.g.*, takesem is only possible if the semaphore is free which only the simulator knows. An event contains a time stamp, type of event, and an id of the source task. The time stamp specifies when the event is to be fired. Consequently, new decisions about what task to execute are taken upon an event. The scheduler decides what task that is to execute according to the fixed priority strategy.

The "execute" kernel-call, the consumption of time, is what drives the simulation forwards.

First, an execution time is selected according to the distribution that is passed as an argument. The current time is increased with that amount of time, or until an event interferes with the execution. If an event occurs during the execution of a task, the execution is suspended, the event is taken care of and the scheduler makes a new decision. The next time the preempted task is allowed to execute, it will restart the execution of the execute-instruction, remembering how much time it has left for execution.

Since an "execute" kernel call is necessary for pushing the simulation forwards, there must always be a task that is ready to execute and contains such a statement. Due to this it is mandatory to have an idle-task in the simulation that consumes time if no other task is ready.

2.5 The probabilistic property language

The impact of altering a component, or adding components due to new features can be analyzed based on the simulation results. Basically, we compare the result from simulating the extended system with simulations performed without the extension. The differences constitute the impact. For real-time systems there exists an overarching

criteria somewhat parallel with the impact analysis. The utilization of available resources must not exceed the upper limit and the temporal requirements must not be violated. Moreover, particular component may have temporal requirements associated with their execution that must be conformed to. Typically examples are deadlines and jitter, *i.e.*, variations in periodicity. The temporal behavior can also affect other requirements. In the case study performed at ABB robotics the correctness of the system was partly dependent on the non-emptiness of a particular message queue. The temporal behavior of components in the system had influence on this requirement.

The result of an impact analysis is in the form of the probability of violating a requirement due to the modeled change of the system. If the system is in the class of hard real-time systems, *i.e.*, all temporal requirements must always be fulfilled. Thus, the probability of complying with a requirement must be 1.

Even if all temporal requirements are fulfilled after changing the system, there still is an impact. For instance, the response times of a component may increase or decrease. The decrease and increase in response times corresponds to the differences in response time distribution obtained by simulating before and after changes in the analytical model.

The requirements are specified in the simulation approach with a *probabilistic requirement property language*, (PPL). PPL can specify probabilistic properties on tasks that control the execution of components and on message queues over which components communicate. Given the number of times a requirement property has been violated the probability of violating it can be calculated.

For every requirement property there must be a property theory which is used for evaluating the simulation. As the property theory for simulation is based on observations from simulating the system, the property gets proportionately simple compared to the correspondence in the analytical approach [5]. For instance, checking the deadline property of a task is done by comparing every observed response time, *i.e.*, the response time distribution, with the required deadline. If the response time is greater than the deadline, the requirement is violated. Given a response time distribution we can calculate the probability of violating the deadline. As an example, consider the response time distribution displayed in Figure 5. The probability of violating a deadline requirement of 24 ms is equal to 0.1.

The requirement property language supports the definition of properties as well as theories for calculating the property, *i.e.*, defining the property theory in terms of the variables available after a simulation with a probabilistic property language. The probabilistic property language specifies properties based on the knowledge generated by the simulator, and includes relation operators, $=, >, \geq, <$

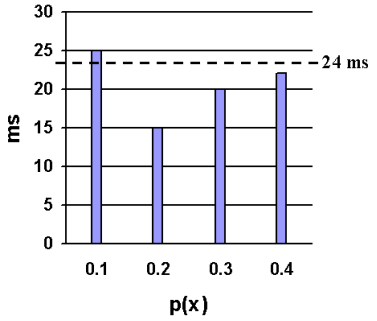

Figure 5. The response time distribution of a task

$, \leq, \neq$, the logical operators $\wedge, \vee$, the binary operators +, -, the functions *max* and *min*, and an *instance operator*. The instance operator binds a task to instances of execution and enables specifying properties where the relative mutual relation among instances of tasks is of importance.

The output generated by the simulator determines the properties available for every task and message queue. Currently the simulator generates the following temporal data about tasks and message queues in a system:

- Size of message queues at task switches, $q.size$
- Time when a task starts an execution, $\tau.start$
- Time when the execution of a task was interrupted
- Time when the execution of a task was restarts after an interrupt
- Time when a task has finished its execution, $\tau.end$

The response time for a task, $\tau.response$, is not generated as such but can be calculated as $\tau(i).end - \tau(i).start$.

Properties are specified as probabilistic statements. Specifying an invariant property, *i.e.*, a property that should always be true, corresponds to a probability equal to 1. A property that verifies that all instances i of task τ, $\tau(i)$, always meet a deadline of 10 time units is:

$$P(\tau(i).response < 10) = 1 \; (hard\; deadline)$$

If it is not critical that every instance of a task meet its deadline, we say that the deadline is firm. In our probabilistic property language we can express a firm deadline as:

$$P(\tau(i).response < 12) \geq 0.75 \; (firm\; deadline)$$

The instance operator is used to distinguish different instances of the same task from one another, or to specify

properties over the same instance number for different tasks. Separation is a property that specifies the minimum distance in time between two consecutive instances of a task.

$$P(\tau(i+1).start - \tau(i).end \geq 10) = 1 \ (separation)$$

A precedence relation specifies the order in which two tasks should execute.

$$P(\tau(i).end <= \sigma(i).start) = 1 \ (precedence)$$

The probabilistic statements may contain an unbounded variable. For instance, the probability may be unbounded which gives as result the probability of the statement being true. A property that specifies the probability of meeting a deadline equal to 10 time units is:

$$P(\tau(i).response < 10) = X$$

We can also leave variables in the predicate unbounded. This could, for instance, be used for feeding back temporal constraints to control engineers, *e.g.*, the feedback loop delay. The probabilistic property that answers with what deadline will be met with a probability of 0.9 is:

$$P(\tau(i).response < d) \geq 0.9$$

Specifying firm deadlines only in terms of the probability of missing them is not sufficient since the distribution of deadline misses can be nasty. For instance, we can miss many consecutive deadlines and still fulfill the temporal requirement since sufficiently many deadlines are met in between bursts of deadline misses. In the probabilistic property below, we specify that two consecutive instances of task τ must not both miss their deadline.

$$P(\tau(i).response > 10 \wedge \tau(i+1).response > 10) = 0$$

Correctness criterion for real-time systems may not only be specified in terms of explicit temporal requirements. As discussed earlier in this paper, the correctness of a system may be defined in terms of non-empty message queues. Such a invariant requirement expressed in our probabilistic property language would be:

$$P(queue.size > 0) = 1 \ (non - emptiness)$$

Calculating the properties of a system is done offline from a simulation point of view, *i.e.*, it is done when the simulation has produced its output. Thus, it will not influence the simulation performance. Moreover, the output generated by the simulator is in an equivalent format to the data measured on the real system. This makes it possible to apply the probabilistic properties on the implementation as part of the verification. Consequently, confidence that the implementation is a refinement of the model can be established.

3 Conclusions

System complexity can be handled informally in early phases of large software system's life time. However, as the system evolves due to maintenance and the addition of new feature, the harder it gets to predict the temporal behavior. Even though a formal model of the temporal domain was initially constructed, it may become obsolete if it is not updated to reflect the changes in the implementation.

The method proposed in this paper is intended for the introduction, or re-introduction, of analyzability into complex distributed real-time systems with respect to temporal behavior. A suitable modeling language, ART-ML, was developed, as well as tools for measuring execution times and the length of message queues in the existing system. Moreover, a tool for processing the measured data was developed. The data processing tool approximates the execution time distributions for the investigated tasks.

A discrete-event based simulator was used when analyzing the temporal behavior of systems described in ART-ML. The simulation approach was chosen since no existing analytical method for analyzing the temporal behavior of a real-time system can express execution times as probabilistic distributions. Furthermore, the simulation approach enables us to define correctness criterion other than meeting deadlines, *e.g.*, non-empty message queues in the system.

The probabilistic requirement property language is used for specifying the properties of a system that is to be analyzed. The analysis of a query specified in the probabilistic requirement property language gives as a result the probability of complying with the specified property. The analysis explores the output generated by the simulator.

The method has been successfully applied in a case study of a robot controller at ABB Robotics where a model was constructed and the temporal behavior was simulated. Even though the model was rather abstract in terms of both functional dependencies and temporal behavior, the results were very promising. Based on this result we claim that our method can be applied on a large class of systems.

ART-ML is still a prototype, thus many improvements of the method and the language are possible. Currently we are expanding ART-ML to also support the modeling and analysis of multi-processor systems. Moreover, we are implementing constructions in ART-ML to describe complete product lines, *i.e.*, a set of related products that share software architecture and software components. If such constructions exist, the impact of altering the behavior of a soft-

ware component can be analyzed for all products that use it.

References

[1] J. Andersson and J. Neander. Timing Analysis of a Robot Controller, 2002. http://www.mrtc.mdh.se.

[2] N. Audsley, A. Burns, M. Richardson, and A. Wellings. STRESS: A Simulator for Hard Real-Time Systems. *Software-Practive and Experience*, 24(6):534,564, 1994.

[3] 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.

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems: PredictableScheduling Algorithms and Applications*. ISBN 0-7923-9994-3. Kluwer Academic Publisher, 1997.

[5] S. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau. Packaging Predictable Assembly. In *Proceedings of the 1st IFIP/ACM Working Conferance on Component Deployment*, 2002.

[6] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[7] S. Manolache, P. Eles, and Z. Peng. Memory and Time-efficient Schedulability Analysis of Task Sets with Stochastic Execution Time. In *Proceedings of the 13nd Euromicro Conference on Real-Time Systems*. Department of Computer and Information Science, Linköping University, Sweden, 2001.

[8] M. E. Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *8th International Conference on Real-Time Computing Systems and Applications*. IEEE, March 2002.

[9] M. Storch and J.-S. Liu. DRTSS: a simulation framework for complex real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Dept. of Comput. Sci., Illinois Univ., Urbana, IL, USA, 1996.

[10] A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *Proc. International Conference on Real-Time Embedded Computing Systems and Applications*, 2003.