A Framework for Analysis of Timing and Resource Utilization targeting Complex Embedded Systems

Johan Andersson, Anders Wall, and Christer Norström

Department of Computer Science and Electronics, Mälardalen University, Box 883, Västerås, Sweden,
{johan.x.andersson,anders.wall,christer.norstrom}@mdh.se

Abstract. A problem in common of many complex software systems embedded in industrial products is the absence of analyzability as formal models of the system behavior does not exist. When performing maintenance of such systems it is hard to predict how changes will impact specific system properties related to timing and resource utilization and there is therefore a significant risk of running into problems with unexpected side-effects of the changes made, which increases development time required and costs.
In this paper we present the ART Framework, a set of methods and tools that enable behavior impact analysis for existing industrial real-time systems. The ART Framework enables developers of complex software systems to identify problematic side-effects of a proposed design before vast resources have been invested in implementation and testing. This reduces the risk of expensive and time-consuming problems discovered late in a development project and also reduces the risk releasing software containing latent critical errors.

1 Introduction

Large industrial software systems evolve as new features are being added. This is necessary for the companies in order to be competitive. However, this evolution typically causes the software architecture to degrade, leading to increased maintenance costs. Systems of this kind, e.g. industrial robot control systems, automotive systems and process control systems, have typically evolved considerably from their first release as a result from many years of maintenance and are today maintained by a staff where most of the people were not involved in the initial design of the system. Most such systems are not analyzable today as they lack formal models describing their behavior.

The architectural degradation is a result of maintenance operations (e.g. new features and bug fixes) performed in a less than optimal manner due to e.g. time pressure or insufficient documentation. As a result of these maintenance operations, not only the size but also the complexity of the system increases as new dependencies are introduced and architectural guidelines are broken. As a result it becomes harder and harder to predict the impact a certain maintenance operation will have on the system's behavior and the risk increases that an error occur due to an unexpected side-effect of a change. For industrial software systems with real-time requirements this is especially a problem, since side-effects on timing can cause critical errors. Moreover, such errors are often very hard to find when testing the system, as they might only occur very specific, rare situations that are hard to reproduce [1].

By introducing analyzability with respect to properties of interest, e.g. response times, the understandability of the system can be increased and side-effects of suggested designs can be predicted in early phases of development, before a lot of resources have been invested in development and testing. Thereby, the development cost and time required for new features can be reduced, and the risk of releasing software with latent critical errors is reduced.

Introducing analyzability, and consequently introducing the possibility of understanding the impact that changes will have on the system behavior with respect to timing, can be done in two distinct ways: intrusively or non-intrusively. In an intrusive approach the system is re-designed in order to make it analyzable. An example of an intrusive approach is to redesign the system to fulfilling the requirements of the fixed priority scheduling principle. The intrusive approach is, however, associated with a high cost as it might require a considerable effort to re-design the system. It is also associated with a high risk since errors might be introduced that, in worst case, is not captured during testing.

A non-intrusive approach is to construct an analyzable model of the system. Hence, the system is kept intact and unchanged which minimizes the cost and the risks. There is however risks with this approach as well. The model might not be updated as the system evolves, e.g. due to time pressure.

The work presented in this paper focuses on the latter approach. Initially a system model is constructed based on both the structure and the dynamic behavior of the system. The model is thereafter validated with respect to interesting system properties and expected types of changes. Given that a model has been constructed and validated, it can be used for Impact Analysis, i.e. predicting the impact a change will have on the runtime behavior of the system.

The work originates in a case study at ABB Robotics in Sweden, where a statistical model describing the temporal behavior of a large industrial real-time system was constructed [2]. The work resulted in the development of the ART Framework, a probabilistic modelling and analysis framework, including a modelling language and analysis tools. The system studied at ABB is based on VxWorks, a commonly used Real-Time OS [3].

Apart from Impact Analysis, there is another use of the tools presented in this paper, Regression Analysis. Instead of analyzing a model, after each major change of the system, measurements are made of the current implementation and analyzed with respect to important system properties. The results from the analysis are compared with results from previous versions of the system. Unexpected differences can point out potential problems and undesired behavior.

The contribution of this paper is a presentation of the ART Framework, including methods for Impact Analysis, Regression Analysis, Model Validation as well as languages and tools support.

The paper is organized as follows: In the next section we present related work, Section 3 presents the general approach, the ART Framework. Section 4 is an overview of the two languages included in this framework, the modelling language ART-ML and the property language PPL. Section 5 presents a method for validation of simulation models. In Section 6 we present a set of tools developed to support this approach. Section 7 describes an industrial case study where the feasibility of this approach has been evaluated and finally, in Section 8, we conclude the paper and give hints on future work.

2 Related Work

Dynamic Analysis is the area of analyzing data generated by a program at execution time and includes e.g. performance analysis, error localization and runtime system monitoring. There are quite a lot of work within the area that relates to this, since we base the ART Framework on the concept of analyzing observations of the system, i.e. recorded data. Parts of the work within the Dynamic Analysis community also deals with reconstructing architectural descriptions of systems, based on observations.

For instance, a system called DiscoTech is presented in [4]. Based on run time observations an architectural view of the system is constructed. If the general design pattern used in the system is known, mappings can be made that transforms low level system events into high level architectural operations and from that construct an architectural description of the system. The system presented is designed for Java based systems. The type of operations that are monitored are typically object creation, method invocation and instance variable assignments. Automated, or mechanical, generation of models based on observations of the system is very related to the construction of the structural model described in Section 3.1. We intend to investigate automated generation/validation of ART-ML models in future work.

Another work related to the construction of ART-ML models is [5]. They present a process for reconstructing software architectures, Symphony. The process incorporates the state of the practice, is problem-driven and uses a rich set of architectural views. It provides guidance for performing reconstruction. Symphony consists of two stages. The first stage is to create a reconstruction strategy, selecting what views to reconstruct. The second stage is the execution of the strategy, i.e. to perform the reconstruction of the selected earlier views.

An approach for deterministic replay is presented in [6]. A common problem when debugging real-time systems is to be able to reproduce an error in order to find the source of this error, i.e. the bug. Due to behaviors such as task-switches and interrupts, finding the bug by studying the code alone is very hard. In their approach, they instrument a real-time system with software probes, collecting various data describing the state of the system. After an error has been observed, the data can be stored and used to replay the execution using a debugger. This is very related to the ART Framework, since they use a similar technique for recording, i.e. software probes, records similar things and has the same overall purpose, to make it easier to develop complex dependable systems reliability. There are clear differences as well. Compared to our work, they record much more details of the execution of the system, but for a much shorter time. They use this very detailed data to exactly reproduce an execution, in order to find bugs, while the data recorded in the ART Framework is used to build probabilistic models, enabling Impact Analysis.

There is a lot of work within the area of formal methods. Model Checking is a technique for verifying different properties of models. For real-time systems, a commonly used tool is Uppaal [7, 8]. Uppaal is an integrated tool environment for modeling, simulation and verification of real-time systems. Uppaal is suitable for systems which can be modeled as a collection of nondeterministic processes with finite control structure and real-valued clocks that are communicating through channels or shared variables. Some major areas where this is applied include real-time controllers and communication protocols where especially those in which timing aspects are critical. A general problem with model checking is the state-space explosion. The general idea behind Model Checking is to search all states of the model for a certain condition. However, if the model contains a lot of parallel processes and clocks, the number of states easily becomes overwhelming and thus too large to search. Compared to the simulation approach in this work, Model Checking gives a lot higher confidence, since all states of the model is explored. However, the state space explosion problem limits the complexity of the models that can be analysed, so in many situations Model Checking is not an option and Model Checking has the same problems with model validity as the simulation based approach in this work.

A tool-suite called STRESS is presented in [9]. The STRESS environment is a collection of tools for analyzing and simulating the behavior of hard real-time safety-critical applications. STRESS contains a modeling language where the behavior of the tasks in the system can be modeled. It is also possible to define algorithms for resource sharing and task scheduling. STRESS is in some ways similar to the ART Framework, but there are a lot of differences too, as STRESS is primarily intended as a tool for testing scheduling and resource management algorithms. It does not allow probabilistic modeling like the ART Framework.

Another simulation framework called DRTSS is presented in [10]. DTRSS is a high level simulation framework that allows its users to construct discrete-event simulators of complex, multi-paradigm, distributed real-time systems. The DRTSS framework contains a set of algorithms and protocols from which one can pick the appropriate ones and build a simulator. New algorithms and protocols can be added to the original set. It has support for searching for extremes in the timing behavior of the simulated system. DRTSS is a part of the PERTS tool-suite, which was developed at the University of Illinois at Urbana-Champaign. The PERTS tool-suite has been commercialized by Tri-Pacific Software Inc. [11].

Analytical methods for dealing with probabilistic temporal attributes have been proposed in the literature. In [12], 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.

Another analytical approach to probabilistic analysis is presented in [13]. Here they assume execution times and deadlines that both vary over time in an unpredictable manner, while their arrival times are fixed. Basically, the task model consists of a set of scenarios where every scenario is associated with a probability. For instance, a task may arrive with a certain execution time and deadline with a specified probability. Tasks execute probabilistically depending on several factors, e.g. the scheduling algorithm. The paper proposes solutions for Earliest Deadline First (EDF), and Least Laxity First (LLF). Even though the computational complexity of this solution has not yet been established, it seems, intuitively, that it is quite large.

3 Concepts of the ART Framework

The purpose of the ART Framework is to increase the maintainability, reliability, and understandability of complex real-time software systems by introducing analyzability. The core of the framework is the process describing how the model is constructed and used. Since this process is general it can be instantiated using any appropriate modeling and analysis method. The process is intended to be integrated in the life-cycle process at software development organizations. The general idea in the ART Framework is the use of a model for impact analysis of timing and utilization of logical resources caused by maintenance operations, e.g. changing an existing feature or adding a new feature. Models are constructed through reverse engineering of an existing system's implementation by identifying the architectural structure and by profiling of the runtime system, an example being execution time distributions for features or tasks. We will start with a brief overview of the process (Figure 1) and describe its individual steps in more details later on in this paper. The process consists of five steps:

1. Construct (or update) a structural model of the system, based on system documentation and the source code.
2. Populated the structural model with data measured on a running system. This data is typically probabilities of different behaviors and execution times.
3. Validate the constructed model by comparing predictions made using the model with observations of the real system. If the model does not capture the system's behavior sufficiently, the first two steps are repeated in order to construct a better model and the new model is validiated. This process should be repeated until a valid model is achieved.
4. Use the model for prototyping a change to the system, for instance if a new feature is to be added, the model is used for prototyping the change.

5. Analyse the updated model in order to identify any negative effects on the overall system behavior, such as deadline misses or starvation on critical message queues.

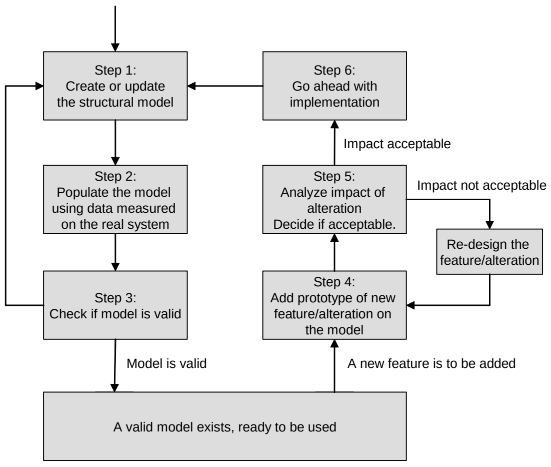


Fig. 1. The process of introducing and using a model for impact analysis

If the impact of the change on the system is unacceptable, the change should be re-designed. On the other hand, if the results from the analysis are satisfactory the change can be implemented in the real system. The final step when changing the system is to update the model so that it reflects the implementation in the real system by profiling the system in order to update the estimated execution times, steps 1-3 in Figure 1.

3.1 Constructing the structural model

To construct a structural model of a system is to document the architecture and behavior of the system in a notation suitable for analysis, at an appropriate level of abstraction. The resulting model describes what tasks there are, their attributes such as scheduling priority and their behavior with respect to timing and interaction with other tasks using e.g. message queues and semaphores. To construct this model requires not only studies of the system documentation and code, but also to involve system experts throughout the complete process. This is important in order to select what parts of the system to focus the modeling effort on, since it is likely that some parts of the system are more critical than others and thus more interresting to model. Other parts of the system can be described in a less detailed manner, in the extreme case with no behavioural description, only describing the execution time distribution.

Iterative reviewing of the model is necessary in order to avoid misunderstandings, due to e.g. different backgrounds and views of the system. If the system is large, this step can be tedious, several man months is realistic if the model engineer is unfamiliar to the system, according to our experiences. An experienced system architect can probably construct this structural model faster, but since such experts often are very busy, it is likely that the model developer is a less experienced engineer. In order to simplify the construction of the model, reverse-engineering tools such as Rigi [14, 15] or Understand for C++ [16] can be used. These tools parse the code and visualize the relations between classes and files.

3.2 Profiling and populating the model

In step 2 of the process described in Figure 1, the system is profiled in order to populate the model with execution times distributions and probabilities. The runtime behavior of the system is recorded with respect to task timing, i.e. when tasks start and finish, how the task preempt each other, their execution times and the usage of logical resources such as the number of messages in message queues. This requires the introduction of software probes, unless hardware probes are used [17]. The problem with the latter approach is that it requires special-purpose hardware. The output from the profiling is a stream of time-stamped events, where each event represents the execution of a probe. Typically, the execution of a probe corresponds to a task-switch or an operation on a logical resource (e.g. message queue).

The measured data can be graphically visualized in order to increase the understandability of the system. In Figure 2, two such graphs are depicted, showing the temporal behavior of two tasks in a real system we have studied. These tasks are complex and time-critical, communicating with many parts of the system. Each graph shows the execution time and response time of task instances during a specific time interval. A task instance, a job, is one execution of a task. Each instance results in two dots in the graph corresponding to the task, the execution time (stars) and response time (squares), for the corresponding instance. In these graphs, we can observe a discrepancy around time 400. The execution- and response-times of these instances of the tasks are very high, in comparison to the other instances in the graphs. This could be a coincidence, but most likely this is due to dependencies with other tasks, e.g. communication or a response to a global state change in the system. The temporal behaviors of these tasks are thus not independent. By introducing such dependencies in the model of the system, the model will become more accurate with respect to the implemented system. In this case, the cause could be that both tasks reacts to a global state change, caused by a command from a human operator or a message from another system. To update the model with this dependency, the states and the various reactions on state transitions must be modeled as well, either in detail or in a probabilistic manner.

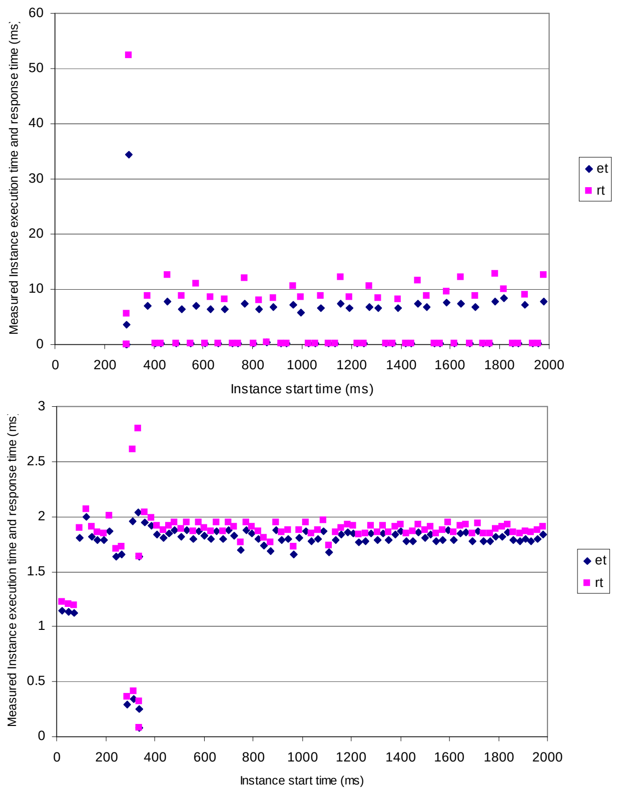


Fig. 2. Visualization of observed execution and response times of two tasks

3.3 Analysis of system properties

The analysis method decides what properties that can be analyzed and also affects the confidence assessment of the result. The main focus of the ART Framework is to support analysis of probabilistic system properties related to timing and usage of logical resources.

A deadline property is a requirement on a response time, either on a particular task or features involving several tasks, i.e. end-to-end response time. A deadline property can be formulated as hard, absolute requirements, or as a soft deadline. An example of a formulation of a soft deadline is that at least 90% of the response times of the instances of a task are less than a specified deadline and that no more than two consecutive instances exceeds that deadline. The PPL language, described in Section 4.2, can be used to formulate such properties. Other system properties related to timing can be e.g. if task X is preemting task Y, and in that case, how often?

Resource usage properties are those addressing limited logical resources of a system such as fixed size message buffers and dynamic memory allocation. When analyzing such properties, the typical concern is to avoid "running out" of the critical resource. An example is the invariant that a data buffer must never be empty. It could be a system requirement that the buffer always contains data, in order to avoid blocking the reading task.

The analysis of system properties is done by recording a trace of the dynamic behavior, either from a real system or from a simulation, and based on the recording the properties can be evaluated, e.g. probabilistic properties such as the soft deadline property described above can be evaluated by counting the number of instances of a task for which the property holds and comparing it to the total number of instances. Comparing to model checking, instead of an exhaustive search of all possible scenarios, the output from a running system or a simulator is analyzed. This gives a realistic picture of the system behavior, but the analysis result is not necessarily safe, as this does not explore all possible situations. However, we believe that this is a suitable solution for large and complex systems, as industrial software systems often are. Even though a simulation might miss some situations, it may still point out potential problems and thus guiding the developers making the right decisions. Formal methods such as model checking often has problems to manage the complexity of such systems, the state space becomes too large to search. Traditional methods for response time analysis are not suitable either [2], as they are too pessimistic and make assumptions that often are violated by the real world system.

Due to the size of the recorded trace, tool support is necessary and therefore we have developed the PPL analysis tool, which evaluates properties formulated in the PPL language. The language is described in Section 4.2. The ART Framework contains graphical front-end tools, making PPL relatively user friendly. This is described in Section 6.

3.4 Validating the Model

The third step in the process is to validate the model, i.e. to determine whether or not the constructed model is a correct description of the system with respect to the system properties that the model is intended to describe. This is typically done by comparing observations of the system's behaviour with the predictions derived from the model.

This is however not trivial, since a direct comparison of the traces is not feasible. The comparison must be made on a higher level of abstraction, by comparing system properties. The properties are evaluated as described in Section 3.3, with respect to both the predictions based on the model and measurements of the real runtime system.

In order to facilitate future usage of the model, it should be easy to keep the model and the system consistent as the system evolves. The effort of adjusting the model to reflect the impact of a maintenance operation should not be similar to constructing the initial model, the necessary change in the model should be intuitive and similar to the change in the system. Therefore, it is necessary to verify that the model is robust with respect to typical, foreseen, changes of the system. Model validation and robustness is further discussed in Section 5, where we present a methodology for establishing the validity of a model.

3.5 Using the Model for Impact Analysis

Given that a model has been constructed and validated, it can be used for predicting the impact an maintenance operation will have on the runtime behavior of the system. The change is prototyped in the model and simulations of the updated model are made, generating execution traces. These are analyzed (as described in Section 3.3) in order to evaluate important system properties. This analysis can, in a very early phase, indicate if there are potential problems associated with the change that are related to timing and usage of logical resources. If this is the cases the designers should change their design in order to consume fewer resources. Since the change is not implemented yet, this means in practice to impose a resource budget on the implementer.

If the impact of the change is acceptable, and is implemented, the model should be updated in order to reflect the implementation. This corresponds to steps 1 to 3 in the process in Figure 1, i.e. updating the model structure, profiling and validation.

3.6 Regression Analysis and Trend Identification

Two other uses of the measured data are regression analysis and trend identification. The regression analysis is to analyze the current release of the system with respect to certain invariants. This is very close to regression testing, but instead of testing the functional behavior, timing and resource usage is analyzed.

The use of trend identification is to compare different releases of the system with respect to the system properties of interest to study how the evolution of the system affects them. There might be trends that will cause problems in future releases, e.g. execution times are increasing for each release as more features are added. If such a trend is allowed to continue, eventually overload situations will occur. If this is observed early, the appropriate measures can be taken before actual problem occurs. If a model has been developed, the Impact Analysis, described in Section 3.5, can be used in order to predict how an extrapolation of a trend will affect the system.

In order to use this in a development organization, measurements of new releases needs to be made. This would typically be made during the system testing. The only change would be that after each test case, an execution trace is stored. This trace is analyzed and compared with earlier releases, using a highly automated tool. Based on a set of rules, typically defined by system experts, the tool decides if there are alarming differences and in that case instructs the tester to notify a system expert. A tool supporting this is presented in Section 6.3.

4 ART-ML and PPL Languages

In this section we describe our approach of the modeling and analysis. First we will present the ART-ML modelling language, the notation which we use to construct probabilistic models. We will also present the Probabilistic Property Language, PPL, which is used to formulate the system properties that we wish to analyze.

4.1 The Modeling Language ART-ML

The ART-ML language describes a system as a set of tasks. Each ART-ML task consits of two parts, the attributes and the behavior. The attributes describe static properties of the tasks, such as name, scheduling priority and optional periodicity. The behavior part of a task is described in an imperative language, C extended with ART-ML primitives, and describes the temporal and to some extent the functional behavior.

```
TASK SENSOR
    TASK_TYPE: PERIODIC
    PERIOD: 2000 us
    PRIORITY: 1
BEHAVIOR:
    execute((0.40, 1000),(0.54, 1300), (0.06, 1400));
    sendMessage(CTRLDATAQ, MSG_A, NO_WAIT);
    chance(0.19){
            execute ((0.60, 200), (0.40, 230));
            sendMessage(CTRLCMDQ, MSG_B, FOREVER);
    }
END
```

Fig. 3. A Typical ART-ML task

Models written in ART-ML are translated into C using a translator and then compiled and linked together with the ART-ML C-library. The resulting executable is a synthesis of the ART-ML model, i.e. a specialized simulator program for that model only. When this simulator is executed, it produces an output in the form of a trace.

Apart from tasks there are two other elements in the ART-ML language: Message Queues and Semaphores. They can be accessed from the behavior part of tasks through ART-ML functions, such as *sendMessage*. Next follows the basic elements of an ART-ML model explained in detail.

Task A task consists of three parts, a name, a set of attributes and a behavioral description. The attributes are the scheduling priority and how the task is activated, one-shot, periodically (with or without offset) or sporadically. The behavior is described in C, extended with ART-ML primitives and routines.

Within the task behavior it is possible to call ART-ML routines that correspond to typical OS services, such as sending and receiving messages to message queues and semaphore operations. There is also a special statement for consuming execution time, execute. The execute-statement is used for modeling sections of code from the real system by their execution time only.

The execution time of the section is described using a discrete probability distribution, i.e. a list of possible execution times, where each execution time is associated with a certain probability of occurrence, see Figure 3. This allows probabilities to be used to describe variations in the execution time. Depending on the selected level of abstraction when constructing the model, an execute-statement can represent a whole task or a smaller section of code.

When a task performs an execute it supplies a probability distribution as parameter. An execution time is chosen according to the distribution and the task is put into executing state for that amount of time. During this time, the task can be preempted by other tasks in the system. The task can not be preempted unless executing a kernel routine such as execute, send, sem_take, delay etc. The execution time and probabilities used in an execute statement is assumed to be from measurements (i.e. profiling as dicussed in Section 3.2) of the real system.

```
A()                  // C = 1000 us
if (cond1)           // Ptrue = 0.60
{
    B()              // C = 300 us
    if (cond2)       // Ptrue = 0.10
    {
        C()          // C = 100 us
    }
}
```

Fig. 4. The code corresponding to the first execute-statement in Figure 3

As mentioned, an execute-statement corresponds to a section of code in the real system. Figure 4 depicts the code that is modelled by the first execute-statement in Figure 3. There are three different functions/blocks of code, (A, B and C), for the sake of simplicity having constant execution times (C_A, C_B and C_C). They can be executed in three different ways, A, A-B or A-B-C depending on the conditions cond1 and cond2. These conditions are not known in the model, due to the necessary level of abstraction, but statistical data from observations of the system can be used to derive the different executions times and calculate the probabilities of the different cases. This execution time distribution is used to form the execute statement, as depicted in Figure 3 and Figure 4. This allows the model to accurately describe the execution time of the tasks in the system, without making it unnecessary complex.

Another ART-ML specific statement is the chance statement, non-deterministic selection with probability. It is a variant of the classic IF-statement, but instead of checking if the value of an expression non-zero, as in C, the argument expression is compared with a random number, linearly distributed in the interval [1-100]. The probabilistic selection is evaluated as True if the value of the expression is less than the random number. In that case, just as in C, the next statement/block is executed. If the value is equal or larger than the random number, any else-statement/block is executed. The chance-statement is related to the execute-statement, but instead of probabilistic selection of execution times, chance allows probabilistic selection of different behaviours. A chance statement can be used for mimicking behaviors observed in measurements of the real system, where the exact cause is not included in the model, e.g. exteral stimuli. For instance, in the measurements of the system we can observe that when a certain task executes, it sometimes sends a message to a particular message queue. This can be modeled using a chance statement and a statistical probability derived from the measurements.

Message queue An ART-ML message queue is a fixed size FIFO buffer, storing messages sent by tasks. A message contains only an integer. That should be sufficient since ART-ML message queues are only intended to model communication events and not to tranfer large amounts of data. Other tasks can read the messages from the message queue, in a FIFO manner. A message is sent to a message queue using the ART-ML library routine *sendMessage* and reading a message is done by calling *recvMessage*.

To use a message queue in a model, it must be declared. The syntax for declaring a message queue is as follows:

```
MESSAGEQUEUE name size;
```

Semaphore An ART-ML semaphore provides mutual exclusion between tasks and conforms to the concept of the well known binary semaphores proposed by Dijkstra in the 1960's. A semaphore is declared and identified with its name. A semaphore is locked using the sem_wait library routine (corresponding to djikstra's P, or wait) and released using sem_post routine (corresponding to djikstra's V, signal). The syntax for declaring a semaphore is:

```
SEMAPHORE name;
```

4.2 The Probabilistic Property Language

The purpose of the Probabilistic Property Language, PPL, is to allow formulation of queries on properties related to the temporal behavior a system, such as response times and usage of logical resources. PPL allows formulation of probabilistic properties, e.g. soft deadlines such as "at least 99 % of task X should be completed within 1000 time units".

Compared to temporal logics with probability and time, e.g. TPCTL [18], there are obvious similarities. It is possible to use a temporal logic instead of PPL, but PPL is specially designed for expressing probabilistic properties of the temporal behavior of tasks which makes PPL queries more intuitive for software developers without previous experience of formal methods.

In this section the implemented version of PPL is presented using a set of examples.

The PPL query
A typical use of PPL is to check a deadline property of a task. Example 1 presents a PPL query that checks if all instances i of task A meets a deadline of 1000 time units with a probability of 1. A task instance is a particular execution of the task. A task instance is represented in PPL using it's task identity, start time, finishing time and execution time.

Example 1:

```
P(A(i), A(i).response < 1000) = 1
```

The first parameter to the P operator is a quantifier specifying that the condition in the second argument should be checked for all instances i of the task A. This is different from the original version of PPL [], where the P-function did not accept any quantifier argument. It was discovered during the implementation of the PPL analysis tool that the original definition of PPL had ambiguous semantics when multiple tasks are referred in a query. The quantifier parameter is necessary in those cases to solve the ambiguity. The second parameter is the condition to check. In the example the condition specifies that the response time of the task A should be below 1000. The P operator returns the ratio of the instances in the execution trace for which the condition holds. If the P operator has a return value of 1, it means that the condition holds for all observed instances of the task, i.e. a probability of 1.

PPL allows checking probabilistic properties such as a soft deadline. For instance, a soft deadline requirement could be that at least 90% of the task instances should meet the deadline. An example of a soft deadline is presented in Example 2.

Example 2:

```
P(A(i), A(i).response < 1000) > 0.9
```

PPL has a data model allowing a query to refer to task instances in the execution trace using a combination of task name, instance index and property name, on the form "*task(i).property*". The data model provides five properties for each task instances in the execution trace, which can be used in PPL queries. These are specified in Table 1.

Property	**Description**
start	The time when the instance started
end	The time when the instance finished
exec	The execution time of the instance
response	The response time of the instance
probeN	The value of probe N when the instance started

Table 1. The properties of a task instance in PPL

The "probeN" property of a task instance corresponds to the value of the generic probe "probeN" at the time the task instance is started. A generic probe may monitor any quantifiable property, but typically generic probes are used to monitor logical resources of different kinds, such as the current utilization of a buffer. Further, PPL contains a set of operators and function that allow conditions to be formulated on the data model. These operators are described in Table 2.

Relational operators	=, <, <=, >=, >	value op value -> bool
Logical connectives	and, or, not	bool op bool -> bool
Arithmetic operators	+, -, *, /, abs	value op value -> value
Statistical functions	max, min, avg, median	op(list) -> value
Index operator	X(i)	op(list, index) -> instance
Following operator	X(following(Y(i)))	op(list, list, index)-> instance

Table 2. The operators of PPL

PPL queries using the instance operator
The index operator is used to differentiate instances of the same task. One property that can be checked using the index operator is temporal separation, i.e. a property that specifies the minimum distance in time between two consecutive instances of a task. This is demonstrated by Example 3.

Example 3:

```
P(A(i), A(i+1).start - A(i).end >= 1000) = 1}
```

Another use of the instance operator is demonstrated in Example 4, which specifies that two consecutive instances must not violate the deadline of 1000 time units.

Example 4:

```
P(A(i), A(i).response > 1000 and
        A(i+1).response > 1000) = 0
```

Expressing a requirement that e.g. 5 consecutive instances must not miss their deadline would result in a very large expression if the presented from the previous example is used. To simplify such queries it is possible to specify intervals rather then single integers in the index operator. Example 5 specifies that there must never be 5 consecutive task instances that violate the deadline of 1000 time units.

Example 5:

```
P(A(i), A(i +[1..4]).response > 1000) = 0
```

Queries using functions and unbounded variables
In order to relate adjacent instances of different tasks, the *following* function can be used. Example 6 shows a query checking if there are any situations where an instance of A and the following instance of B have execution times above 1100 time units and 1700 time units respectively.

Example 6:

```
P(A(i), A(i).exec > 1100 and
        B(following(A(i))).exec > 1700) > 0
```

Moreover, PPL queries may contain an unbounded variable. For instance, by specifying the probability as an unbounded variable, the result of the query is the minimum/maximum value for which the condition holds. A query using an unbounded variable to evaluate the probability of meeting a deadline of 2000 time units is presented in Example 7.

Example 7:

```
P(A(i), A(i).response < 2000) = X
```

It is also possible to use unbounded variables inside the second parameter of the P-operator. A query evaluating the shortest deadline D that is met with a probability of at least 0.9 is presented in Example 8.

Example 8:

```
P(A(i), A(i).response < D) >= 0.9
```

Statistical functions
As presented in Table 2, there is also a set of statistical functions that may be used to extract simple statistical measures of the different tasks. The statistical functions can be used as stand-alone queries as in Example 9.

Example 9:

```
avg(A.response)

median(A.exec)

max(A.exec)
```

The statistical functions can also be used instead of constant values inside the second parameter of the P-operator, as in Example 10.

Example 10:

```
P(A(i), A(i).resp > avg(A.resp)*2 ) = X
```

The above described PPL query returns the probability (X) of task A having a response time above a certain limit, which is specified as "two times the average response time".

Queries on logical resources
PPL also allows queries on data from generic probes. A generic probe may monitor any quantifiable property of the system, but are typically used to monitor logical resources, such as the usage of a data buffer. In the current implementation, the generic probes are identified using a number. If the number of messages in a certain message queue is monitored using generic probe number 21, it is possible to formulate a PPL query checking that the message queue is never empty when taskX is activated as presented in Example 11.

Example 11:

```
P(taskX(i), taskX(i).probe21 > 0) = 1
```

It is also possible to specify conditions on a probe that are independent of what tasks that are to be executed, by replacing the name of the task with a wildcard character. This is demonstrated by Example 12. For such queries the probabilities are calculated differently, by summing the lengths of the time intervals where the condition holds and divide that with the length of the recording. The resulting value is thus the fraction of the total time in the recording where the condition holds. This is an approximation of the probability that the condition holds at an arbitrary point in time.

Example 12:

```
P(*, *.probe21 > 0) = 1
```

Tool Support
The PPL language is supported by two tools available within the framework. The Tracealyzer tool contains a PPL terminal, where it is possible to formulate and run queries with respect to an execution trace. This is the preferred tool for experimenting with PPL. The Property Evaluation Tool (PET) is a dedicated front- end application for PPL, allowing designed to run a batch of queries on two different execution traces and present the results side-by-side. The tools are presented in Section 6.

5 Validation of models

Validating a model is basically the activity of comparing the predictions from the model with observations of the real system. However, a direct comparison between traces from a simulation and traces from the real system is not feasible since the model is a probabilistic abstraction. Instead, we compare the model and the system based on a set of properties, comparison properties. The method presented in Section 3.3 is used in order to evaluate these comparison properties, with respect to both the predictions based on the model and measurements of the real runtime system. If the predicted values match the observed, we considered the model observable equivalent to the real system. A typical comparison property can be the average response time of a task. It is affected by many factors and characterizes the temporal behavior of the system. Selecting the correct comparison properties is important in order to get a valid comparison. Moreover, as many system properties as practically possible should be included in the set of comparison properties in order to get high confidence in the comparison. The selected system properties should not only be relevant, but also be of different types in order to compare a variety of aspects of a model. Other types of comparison properties could be related to e.g. the number of messages in message queues (min, max, average) or pattern in the task scheduling (interarrival times, precedence, preemption).

Even if the model gives accurate predictions, there is another issue to consider, the model robustness. If the model is not robust, the model might become invalid as the system evolves, even if the corresponding updates are made on the model. Typically, a too abstract model tends to be non-robust, since it might not model dependencies between tasks that allow the impact of a change to propagate. Hence, it may require adding more details to the model in order to keep it valid and consistent with the implementation. If a model is robust, it implies that the relevant behaviors and semantic relations are indeed captured by the model at an appropriate level of abstraction.

5.1 Observable Property Equivalence

We propose a measure of model validity named Observable Property Equivalence, where the model is compared to the real system with respect to a set of system properties of interest. However, since the relation of observable property equivalence is not transitive, this is not a true equivalence relation, only a measure of similarity.

A model and a corresponding system are observable property equivalent if they are equivalent with respect to a set of comparison properties, i.e. statistical measures of the observed temporal behavior. However, as discussed earlier, since the model is an abstraction of the system, it is necessary to allow a certain amount of tolerance in comparison.

In Definition 1 we formalize the observation of a system, x, that is either the real implemented system, executing on the real hardware, or a model of a system executed in a simulator. The resulting recording is a list of time-stamped events related to tasks-switches and operations on logical resources. The environment e specifies the configuration of the system and any external stimuli that effects the system behavior during the observation.

Definition 1 *R = Rec(x, e, d)*
The function Rec returns a recording, R, of the execution of x, in the environment e, with the duration d time units. R is a list of events, where each event contains a time-stamp, an event type and generic data, where the semantics are specific for each event type. □

Definition 2 presents the function Eval, which evaluates a system property p with respect to the recording R.

Definition 2 *v = Eval(p, R)*
The function Eval evaluates the property p with respect to the recording R. The result, v, is a decimal value. If the property p is a boolean expression, v is either 1 (true) or 0 (false). □

Since a certain amount of tolerance is often necessary in the comparison, we introduce a function which expressing the tolerance allowed for a specific comparison property,

Definition 3 *t = Tol(p)*
The function Tol returns the maximum allowed difference between two evaluations of the same property p on two different recordings. The return value, t, is a decimal value. If the property p is of boolean type, the function returns a tolerance of 0. □

Definition 4 presents the definition of observable property equivalence. If evaluations of all comparison properties with respect to the model results in values sufficiently close to the values from the real system recording, the model and the system are observable property equivalent.

Definition 4 *Given that P is the set of comparison properties, M is a model of the system S, E_M is the environment model of M and E_S is the environment of the system S, iff*

$$\forall p \in P : Abs(Eval(p, Rec(M, E_M, d)) - Eval(p, Rec(S, E_S, d))) \leq Tol(p)$$

then $S \equiv M$, i.e. S and M are observable property equivalent with respect to P, in the specific environment. □

Obviously, this relation of similarity relies heavily on the comparison properties and tolerances used. It is important to select a suitable set of comparison properties in order to compare as much as possible of the behavior of the model with the corresponding system.

5.2 Model Robustness

In this section we propose a method for ensuring the robustness of an ART-ML model. We refer to this activity as sensitivity analysis. To exemplify the importance of model robustness, imagine a system containing a binary semaphore protecting a shared resource. A timeout occurs if a task has been waiting on the semaphore for a certain predefined time. If the timeout occur, the execution time of the task is increased due to the error handling necessary. However, in all previous versions of the system, this timeout has never occurred. If the timeout is left out when constructing the model of the system the model still seems accurate since the timeout never occurs. However, as a result from changing the system, e.g. increasing the execution time of another task, the timeout will in some cases occur. Since the timeout was not included in the model, the system's behavior will diverge from the behavior predicted based on the model.

Our approach to sensitivity analysis is influenced by system identification. System identification is a technique used in the domain of control theory [19]. By measuring and observing the input-output relationship between signals in the process a model can be determined in terms of a transfer function. Validating models based on the system identification approach is somewhat related to testing. Typically, output signals predicted using the model is compared with the output signals of the physical process. Hence, the model is regarded as correct if the analysis and the physical process generate approximately the same output, if fed with the same input.

Testing the model with different input signals and comparing the prediction with the signals produced by the actual system is fine if the process is continuous in its nature. It is fair to assume that we can interpolate the behavior in between the tested signals. However, computer software is not continuous; they are discontinuous systems meaning that the behavior may change dramatically as a result of small changes in the system. A model of a software system can thus quickly become invalid when the system evolves, if the model is not robust with respect to typical changes. By analyzing the impact on the system caused by different changes, it is possible to determine if the model is sensitive to such changes, i.e. less robust.

The robustness of a model can be analyzed using a sensitivity analysis. The basic idea is to test different probable alterations and verify that they affect the behavior predicted by the model in the same way as they affect the observed behavior of the system. Performing a sensitivity analysis is typically done after major changes of the model, in the validation step of the process. The process of performing sensitivity analysis is depicted in Figure 5. First a set of change scenarios has to be elicitated. The change scenarios should be representative for the probable changes that the system may undergo. Typical examples of change scenarios are to change the execution times of a task, to introduce new types of messages on already existing communication channels or change the rate sending messages. The change scenario elicitation requires, just as developing scenarios for architectural analysis, experienced engineers that can perform educated guesses about relevant and probable changes.

The next step is to construct a set of system variants $S = (S_1, ..., S_n)$ and a set of corresponding models $M = (M_1, ..., M_n)$. The system variants in S are versions of the original system, S_0, where n different changes have been made corresponding to the n different change scenarios. The model variants in M are

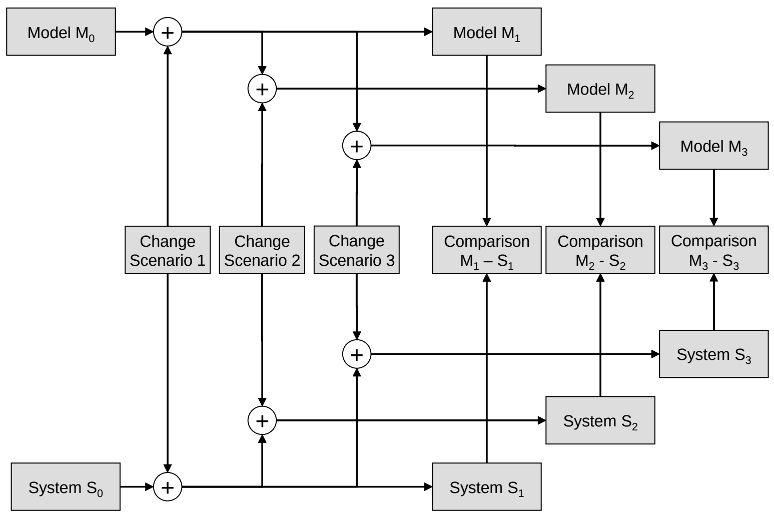


Fig. 5. The Sensitivity Analysis

constructed in a similar way, by introducing the corresponding changes in the initial model M_0.

Note that these changes only needs to reflect the impact on the temporal behavior and resource usage caused by the change scenarios, they do not have to be complete functional implementations. For instance, if the new feature is some sort of service offered by a server task to a client task, the implementation necessary would be to introduce two new messages on the communication channel, a request and a reply, and some minor changes in the tasks. When the request message is received by the server, it should consume a realistic amount of time (e.g. by executing an empty for-loop) and then send a reply message. The client should send requests at the appropriate times and waits for the reply. These changes are therefore easy to implement.

Each model variant is then compared with its corresponding system variant by investigating if they are equivalent as described in 5.1. If all variants are equivalent, including the original model and system, we say that the model is robust.

6 The Tools

This section presents three tools within the ART Framework, supporting the process described in Section 3.

– An ART-ML simulator, used to produce execution traces based on an ART-ML model.

– The Tracealyzer, a tool for visualizing the contents of an execution trace and also allow PPL queries to be executed on the data.
– The Property Evaluation Tool. A tool for analyzing and comparing execution traces, using a predefined set of PPL queries.

The Tracalyzer and Property Evaluation Tool are available for download at

`http://www.idt.mdh.se/~jxn01`

, they can be used freely for academic and other non-commercial purposes, but they are not open source. The only platform supported (so far) is Microsoft Windows.

6.1 The ART-ML Simulator

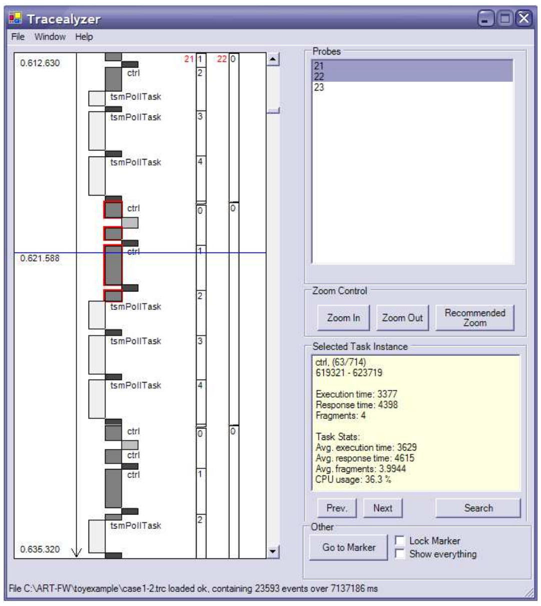
The ART-ML Simulator is basically a C library and an ART-ML to C translator application. Given an ART-ML model the translator outputs an ANSI C representation which can be compiled and linked together with the ART-ML library in order to produce an executable file containing a compiled version the ART-ML model together with configuration and logging functionality. It is possible to specify what seed (an integer number) to use for the random number generator, which makes it possible to reproduce simulations of non-deterministic models. This ART-ML simulator is approximately 5 times faster compared to first ART-ML simulator, which interpreted the model. The output of the simulator is a binary file readable by the Tracealyzer and the Property Evaluation Tool. If desired, the Tracealyzer translate the binary file to text format.

6.2 The Tracealyzer tool

The Tracealyser has two main features, visualization of an execution trace and a PPL terminal, a front-end for the PPL analysis tool. Figure 8 depicts the user interface of the Tracealyzer. The left part contains a window presenting a section of the execution trace. It is possible to navigate in the trace by using the mouse or the scrollbar. It is also possible to zoom in and out.

The leftmost part of the trace shows how the tasks execute, over time. The shaded boxes correspond to uninterrupted execution of a task. The point in time between two boxes corresponds to a task-switch. It is possible to select a task instance, by clicking on it. This will display information about the selected instance in the textbox named "Selected Task Instance" in the right part of the window. A selected task instance is marked with a red frame. This information presented includes the name of the task, the execution time and response time of the instance and the average execution and respose times for the task. If more statistics about the different tasks is desired, it is possible to generate a report, containing a lot of information about all tasks.

The upper right part of the window contains a lists, labeled *probes*, showing a list of the probes that have been found in the trace. Selecting one of the probes



```
21
22
23
```

```
ctrl, (63/714)
619321 - 623719

Execution time: 3377
Response time: 4398
Fragments: 4

Task Stats:
Avg. execution time: 3629
Avg. response time: 4615
Avg. fragments: 3.9944
CPU usage: 36.3 %
```

```
File C:\ART-FW\toyexample\case1-2.trc loaded ok, containing 23593 events over 7137186 ms
```

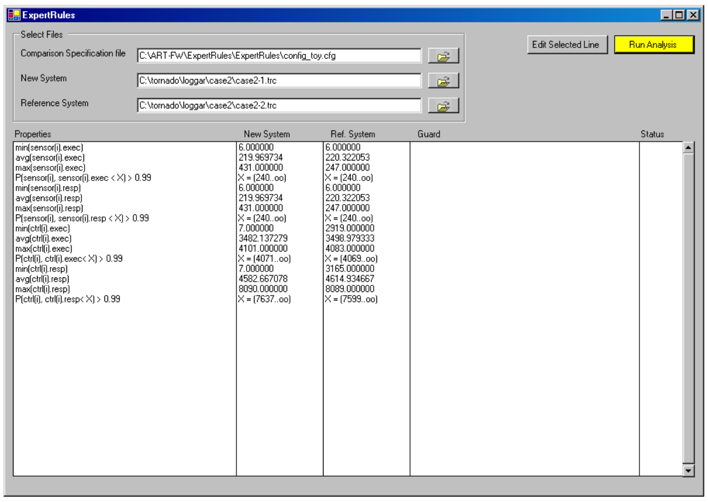
Fig. 6. The Tracealyzer Tool

in the probe list will display the value of the probe over time, next to the tasks. Since a probe can monitor various things, the meaning of a certain probes is defined by the developer that puts the probes in the system. The probes shown in the screenshot of the Tracealyzer monitors the number of messages in different message queues. It is also possible to save a list of the task instances to a text file. This way, the data can be imported into e.g. Excel and visualized in other ways than the ones provided by the Tracealyser. For instance, Figure 9 were created using this feature together with a common office application.

Apart from visualizing the data in an execution trace, the Tracealyzer also contains a PPL terminal. It is basically a front-end for the PPL analysis engine. The terminal contains two fields, one input where PPL queries can be typed and one output where the result is presented.

6.3 Property Evaluation Tool

The Property Evaluation Tool (PET) is a tool for comparing execution traces with respect to different system properties. These properties are formulated as PPL queries. The application has three uses: Impact Analysis, Regression Analysis and Model Validation. In the Impact Analysis case, execution traces from simulation of two slightly different models are compared. One of the models is considered "valid" and used as reference. The other model contains a prototype of a new feature or other changes. By comparing these traces, the impact of the new feature can be analyzed. Impact Analysis is discussed in Section 3.



Comparison Specification file: C:\ART-FW\ExpertRules\ExpertRules\config_toy.cfg

New System: C:\tornado\loggar\case2\case2-1.trc

Reference System: C:\tornado\loggar\case2\case2-2.trc

Properties	New System	Ref. System	Guard	Status
min(sensor(i).exec)	6.000000	6.000000		
avg(sensor(i).exec)	219.969734	220.322053		
max(sensor(i).exec)	431.000000	247.000000		
P(sensor(i), sensor(i).exec < X) > 0.99	X = (240..oo)	X = (240..oo)		
min(sensor(i).resp)	6.000000	6.000000		
avg(sensor(i).resp)	219.969734	220.322053		
max(sensor(i).resp)	431.000000	247.000000		
P(sensor(i), sensor(i).resp < X) > 0.99	X = (240..oo)	X = (240..oo)		
min(ctrl(i).exec)	7.000000	2919.000000		
avg(ctrl(i).exec)	3482.137279	3498.979333		
max(ctrl(i).exec)	4101.000000	4083.000000		
P(ctrl(i), ctrl(i).exec< X) > 0.99	X = (4071..oo)	X = (4069..oo)		
min(ctrl(i).resp)	7.000000	3165.000000		
avg(ctrl(i).resp)	4582.667078	4614.934667		
max(ctrl(i).resp)	8090.000000	8089.000000		
P(ctrl(i), ctrl(i).resp< X) > 0.99	X = (7637..oo)	X = (7599..oo)		

Fig. 7. The Property Evaluation Tool

In the Regression Analysis case, no data from simulation of models are used. Instead, two execution traces measured from two versions of the real systems are analyzed and compared, in order to identify trends and alarming differences, which might be a result of undesired behavior in the system. Regression Analysis is discussed in Section 3.6.

The application uses a set of rules, typically specified by a system expert, in order to judge what differences in system properties between system versions (execution traces) that are alarming and which one that can safely be ignored.

When used for model validation, a trace from simulation is compared with a trace measured from the real system. This way, it is possible to gain confidence in the model validity. This is discussed in Section 5.

The PET application is a front-end to the PPL analysis engine. The user of PET selects a configuration file, containing a predefined set of system properties, formulated as PPL queries. These properties are the point-of-view for the comparison. Then, the user selects two execution traces and starts the analysis and comparison process. The properties are evaluated with respect to the two traces and the results are presented. Finally, a property can be associated with a guard. A guard is a condition on the result from a property. Guards are used to formulate the rules on how much a property is allowed to before the user is notified.

7 An Industrial Case Study

We have applied the framework on an industrial control system. The system we have investigated is a robot control system, developed by ABB Robotics. It was initially designed in the beginning of the nineties. In essence, the controller is object-oriented and consists of approximately 2 500 000 LOC divided on 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, which controls each axis of the robot, and finally the I/O node, which interacts with external sensors and actuators. In this work we have studied a critical part with respect to control in the main node. The controller runs under the preemptive multitasking real-time operating system VxWorks from the company Wind River [3].

Maintaining such a complex system requires careful analyses to be carried prior to adding new functions or redesigning parts of the system not to introduce unnecessary complexity and thereby increasing both the development and maintenance cost.

7.1 The model

We have modeled some critical tasks for the concrete robot system in the main computer (see Figure 8). The axis computer periodically sends requests to the main computer and expects a reply in the form of motor references within a certain time. There are three tasks in the main computer that are responsible for generating these motor references: A, B, and C. The tasks B and C have high priority, are periodic, and runs frequently. A executes mostly in the beginning of each robot movement and has lower priority. The final processing of the motor references is performed by task C. Task C sends the references to the axis node. Moreover, task C is dependent on data produced by task B. If the queue between them becomes empty, task C cannot deliver any references to the axis node. This state is considered as a critical system state and the robot halts. Task A sends data to task B when a movement of the robot is requested. If the queue between task A and task B gets empty, the robot movement stops. In this state, task B sends default references to task C. The complete case study is presented in [20]. All comments have been removed and variable names have been changed for business secrecy reasons. The model is not complete with respect to all components in the system.

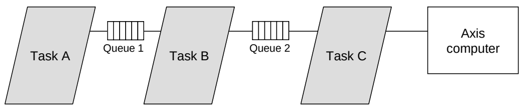


Fig. 8. The task structure of the critical control part of the system

7.2 The results

The model we made is quite an abstraction of the existing system. There were approximately 60 tasks in the system which was reduced to six in the model. This level of abstraction was selected since there were three tasks of particular interest which was modeled in detail. The rest of the tasks were modeled with respect to CPU utilization only, no behavior was described. The axis computer was modeled as a task with zero execution time. The 2.500 KLOC in the existing implementation was reduced to 200 LOC in the model.

A more detailed model would not only represent a more accurate view of the system, it will also prune the state-space which the simulator has to consider. For instance, by introducing some dependency between tasks, the size of the statespace is reduced. This allows the simulator to explore a larger fraction of the possible behaviours of the system during a given the amount of simulation time and thus improving the confidence in the simulation result.

Despite the course-grained model, the result when comparing response times produced by the simulator and the response times measured on the system is quite good. In Figure 9, the response times from the simulation and the real system are plotted. The resemblance is obvious. However, at the time of this case study, no tools that would allow a more formal comparison were available, such as PPL and the supporting tools presented in Section 6.

7.3 Validation results

The results from the case study indicates that we have made one valid model out of many which may be valid for the system in its current state. However, we can not assume the model to be completely correct. In order to validate the model and establish confidence in the model we must develop a set of change scenarios as described in Section 5. Our initial list of scenarios was:

- add/remove tasks to the system,
- add/remove functional behavior in an existing task,
- change the behavior of existing functionality, i.e. changing execution times,
- change the priority of existing tasks,
- change message queue sizes,
- add shared resources,
- change the period time of a task, and
- change the triggering condition of a task.

After interviewing engineers at ABB Robotics, the list was reduced to only include the most likely changes: add/remove tasks to the system, add/remove functional behavior in an existing task, change the behavior of existing functionality, and change the priority of existing tasks.

We developed four different cases out of the scenarios:

- Case 0: No change at all,

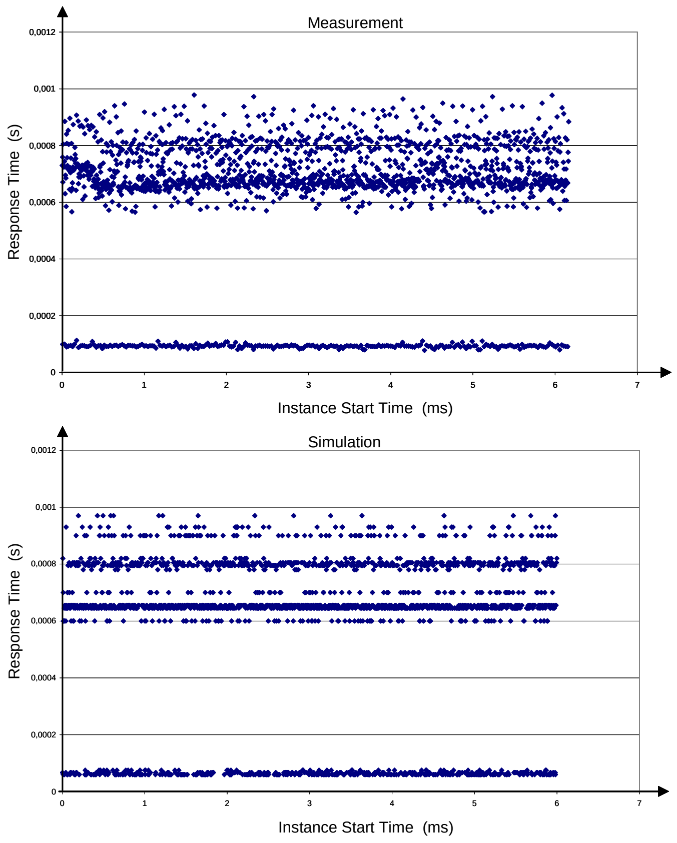


Fig. 9. Response time distribution of a task, simulation and measurement on real system.

- Case 1: Add a new task called dummy with a short, varying execution time and low priority,
- Case 2: Raise the priority of the dummy task drastically, and
- Case 3: Increase the period time for the dummy task and extend its execution time

We model changes in a task's functional behavior by changing its execution time.

In general, by observing the results we see that the model indeed capture the temporal behavior of the system quite well. The simulations follow the measured system over the change scenarios. However, there are small differences in execution times between the model and the system. Consequently, we need to tune the execution time distributions in the model as it is too coarse grained.

Moreover, we had to model yet another composed task since the priority of the dummy task is within the range of the low priority composed task.

The complete result from the validation is provided in appendix E in [21].

8 Conclusions and Future work

In this paper we have presented the ART Framework; the general ideas, the languages ART-ML and PPL, the three tools within the framework and an approach for validating ART-ML models. We have presented a process for use the ART Framework for impact analysis and we have also presented how the framework can be used for regression analysis of timing properties. We beleive that this approach is very useful for analysing properties of complex real-time systems, related to timing and resource utilization.

The next step in this work is to perform an industrial case study evaluating the benefit of performing regression analysis, as described in 3.6, and a continuation of the case study presented in 7, on modeling and analysis, using a more advanced model and the (new) tools presented in 6.

One problem with the approach described in this paper is the error-prone work of constructing the model. Instead of manually constructing the whole structural model, tools could be developed that mechanically generate at least parts of it, based on either a static analysis of the code, dynamic analysis of the runtime behavior or a hybrid approach. This is also part of our future work.

References

1. W. Schutz. On the testability of distributed real-time systems. In *Proceedings of the 10th Tenth Symposium on Reliable Distributed Systems*, pages 52–61. IEEE.
2. A. Wall and 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.
3. Wind River Website. http://www.windriver.com.
4. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discotech: A system for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.
5. A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In *Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA)*. IEEE Computer Society, 2004.
6. H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time systems using Time Machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.
7. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

8. Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Mller, Paul Pettersson, and Wang Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.
9. N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. STRESS: A Simulator for Hard Real-Time Systems. *Software-Practive and Experience*, 24(6):534,564, 1994.
10. M.F. Storch and J.W.−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.
11. Tripac: RAPID Sim High-Performance Simulation of Real-Time Systems. http://www.tripac.com.
12. 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.
13. A. Leulseged and N. Nissanke. Probabilistic Analysis of Multi-processor Scheduling of Tasks with Uncertain Parameters. In *Proceedings of the 9th Conferance on Real-Time and Embedded Computing Systems and Applications*, pages 317–336, 2003.
14. H.A. Muller and K. Klashinsky. Rigi: a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, 1988.
15. Rigi Group Home Page. http://www.rigi.csc.uvic.ca/index.html.
16. Scientific Toolworks, Inc: Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and FORTRAN. http://www.scitools.com/.
17. Mohammed El 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.
18. H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
19. R. Johansson. *System Modeling Identification*. Prentice-Hall, 1993. ISBN 0-13-482308-7.
20. J. Andersson and J. Neander. Timing Analysis of a Robot Controller, 2002.
21. A. Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. PhD thesis, Mälardalen University, 2003. ISBN 91-88834-05-0.