Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings

Anders Wall, Johan Andersson, and Christer Norström

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

Abstract. A common problem with long-lived large industrial software systems such as telecom and industrial automation systems is the increasing complexity and the lack of formal models enabling efficient analyses of critical properties. New features are added or changed during the system life cycle and it becomes harder and harder to predict the impact of maintenance operations such as adding new features or fixing bugs.
We present a framework for introducing analyzability in a late phase of the system's life cycle. The framework is based on the general idea of introducing a probabilistic formal model that is analyzable with respect to the system properties in focus, timing and usage of logical resources. The analyses are based on simulations. Traditional analysis method falls short due to a too limited modelling language or problems to scale up to real industrial systems.
This method can be used for predicting the impact caused by e.g. adding a new feature or other changes to the system. This enables the system developers to identify potential problems with their design at an early stage and thus decreasing the maintenance cost.
The framework primarily targets large industrial real-time systems, but it is applicable on a wide range of software system where complexity is an issue. This paper presents the general ideas of the framework, how to construct, validate, and use this type of models, and how the industry can benefit from this. The paper also present a set of tools developed to support the framework and our experiences from deploying parts of the framework at a company.

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. Such systems, e.g. process control systems, industrial robot control systems, automotive systems and telecommunication systems, have typically been in operation for quite many years and have evolved considerably since its first release and are today maintained by a staff where most of the people were not involved in the initial design of the system. These systems typically lack a formal model enabling analysis of different system properties.

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, wrong competence, 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. Eventually it becomes hard to predict the impact a certain maintenance operation will have on the system's behaviour. This low system understandability forces the developers to rely on extensive testing, which is time consuming, costly and usually misses a lot of bugs. The software systems we are studying have real-time requirements, which mean that it is of vital importance that the system is analyzable with respect to timing related properties, such as response times. However, bugs related to concurrency and timing are hard to find by testing [1], as they are hard to reproduce.

By introducing analyzability with respect to properties of interest, the understandability of the system can be increased. If it is possible to predict the impact of a maintenance operation, it can help system architects making the right design decisions. This leads to a decreased maintenance cost and the life cycle of the system is lengthened.

The work presented in this paper focuses on a model-based approach for increasing the understandability and analysability with respect to timing and utilization of logical resources. A model is constructed, describing the timing and behaviour of the system, based on the source code, documentation and statistical information measured on the system. This model can be used for *impact analysis*, i.e. predicting the impact a change will have on the runtime behaviour of the system. We refer to the general method as the ART Framework. This implementation of the framework is based on the ART-ML modelling language [2]. An ART-ML model is intended to be analysed using simulation. An initial case-study in presented in [3], where we used the modelling language and a simulator is used in order to analyse timing properties of an industrial system. The case-study showed the feasiblity of using the modelling language for this purpuse, but also showed that analysing the simulation output required tool support. We therefore proposed the Probabilistic Property Language (PPL) in [4]. In this paper we present tools supporting PPL.

Existing work related to simulation based analysis of timing behaviour is given in [5, 6]. Analytical methods for probabilistic analysis of timing behaviour is given in [7]. However, none of them fulfils our requirements completely, i.e. a rich and probabilistic modelling language and analyses scalable to large and complex systems.

Our contribution in this paper is a set of tools developed to support the ART Framework, how the tools can be used for impact analysis and regression analysis, and how ART-ML models can be validated. We also present how a company can benefit from using the ART Framework and our experiences from introducing parts of the framework at ABB Robotics in Sweden, one of the world's largest producers of industrial robots and robot control systems.

The paper is organized as follows: In Section 2 the design rationales behind the framework is discussed and in Section 3, we give a brief presentation of the ART Framework, including the modelling language ART-ML and the property specification language PPL. Further, in Section 3.4, we present how the framework can be used, how industry can benefit from it and our experiences from introducing this at ABB Robotics. Section 5 describe the tools we have developed to support this framework. Finally, the paper is concluded in Section 7.

2 Design rationales

There exist many analytical methods for modelling and analysis of a real-time system's temporal behaviour [8, 9]. However, the analytical models and analyses found in conventional scheduling theories are often too simple and therefore a real system cannot always be modelled and analyzed using such methods. The models used in those methods are not expressive enough in order to capture the behaviour of large and complex systems. There is no possibility of specifying dependencies between tasks and the models only allow worst-case execution times to be specified. Moreover, the analyses only cover deadline properties, i.e. whether or not every deadline is violated. In many real systems the temporal requirements are not only expressed in terms of deadlines, for instance there can be requirements on message queues such as starvation properties. Such requirements can not easily be verified with the analytical methods.

On the other extreme we find model-checking methods with rich modelling languages such as timed automata [10, 11]. Timed automata allow modelling of temporal behaviour as well as functional behaviour. By using synchronization channels we can model dependencies between tasks in the system. However, model-checking does not scale properly to larger systems due to the state-space explosion which makes such an approach useless in a realistic setting.

Simulation is better from that point of view. Using simulation, rich modelling languages can be used to construct very realistic models, using e.g. realistic distributions of execution times. A disadvantage of the simulation approach is that we can not be confident in finding the worst possible temporal behaviour through simulation, since the state-space is only partially explored. Therefore, designers of hard real-time systems and safety-critical systems should not rely on simulation for analysis of critical properties if safe analysis methods are possible, such as model checking or scheduability analysis. However, when analysing a complex, existing system, that has not been designed for analysability, simulation is often the only alternative.

The design rationale behind the modelling language ART-ML is to provide a rich, probabilistic modelling language suitable for simulation. It should be possible to describe the behaviour of tasks, how the tasks synchronize and communicate and describe explicit execution times using probability distributions. We could have used an existing notation, for instance timed automata, and extended it with execution time distributions and probabilities. However, we wanted a modelling language similar to the implementation, in order to facilitate the understanding of the model. We believe that software developers that are not used to formal methods are more likely to use this kind of models.

When designing the ART Framework we basically had one major trade-off to consider: being able to predict something at the cost of precision. We have chosen to use simulations as a tool for analysis. Even though a simulation might miss some situations, it may still point out potential problems and thus guiding the developers making the right decisions, while analytical methods are often not even possible to use in practice.

3 The ART Framework

The general idea in the ART Framework is the use of a model for analyzing the impact on timing and utilization of logical resources caused by maintenance operations, e.g. changing an existing feature or adding a new feature. It is also possible to use the tools in this framework to analyse measurements of the system directly, without using a model.

The core of the ART Framework is a general process (Figure 1) describing how model is constructed and used. Since the process is general it to be instantiated using any appropriate modelling and analysis method. The process is intended to be integrated in the life-cycle process at software development organizations.

We will start with a brief overview of the process and describe its 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. Populate the structural model with data measured on a running system. This data is typically probabilities of different behaviours 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 behaviour sufficiently, the first two steps are repeated in order to construct a better model and the new model is validated. This process should be repeated until a valid model is achieved. Model validation is discussed in Section 4.
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 behaviour, such as deadline misses or starvation on critical message queues.

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

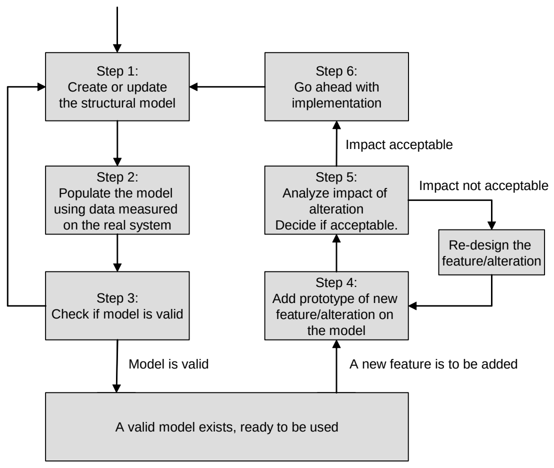

Step 1:
Create or update the structural model
Step 6:
Go ahead with implementation
Impact acceptable
Step 2:
Populate the model using data measured on the real system
Step 5:
Analyze impact of alteration
Decide if acceptable.
Impact not acceptable
Re-design the feature/alteration
Step 3:
Check if model is valid
Step 4:
Add prototype of new feature/alteration on the model
Model is valid
A new feature is to be added
A valid model exists, ready to be used

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

real system by profiling the system in order to update the estimated execution times, steps 1-3 in Figure 1.

3.1 Constructing the model

To construct a structural model of a system is to document the architecture and behaviour 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 behaviour 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 modelling effort on, since it is likely that some parts of the system are more critical than others and thus more interesting to model. Other parts of the system can be described in a less detailed manner.

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 [12, 13] or Understand for C++ [14] can be used. These tools parse the code and visualize the relations between classes and files.

In step 2 of the process described in Figure 1, measurements are made in order to populate the model with execution times distributions and probabilities. The runtime behaviour of the system is recorded with respect to task timing, i.e. when tasks start and finish, how the tasks pre-empt 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.

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).

Since the type of systems we consider are quite large and are changed often, we assume that the probes can remain in the system. This way we avoid the probe effect, since the probes becomes a part of the system, and also facilitates future measurements. The added probes will have a small effect on the system performance, less than one per cent, as the amount of probing necessary for our purposes is quite reasonable. The operating system used, VxWorks, allows user code to executed on every context switch, so by adding a single probe it is possible to record all context-switches and also the resulting scheduling status of the tasks. To record IPC communication requires four probes per task of intererst. This way we can record what message codes that are sent, i.e. what commands, but not the arguments. Recording the usage of a logical resource such as a message queue requires two probes per logical resource of interest. To register operations on semaphores could be done by adding three probes in the OS isolation layer.

Before the model is used to make predictions, it should be validated, i.e. compared with the real system in order make sure that the model described the system behavior correctlty and in an apporpriate level of abstraction. In Section 4 we present a method for this.

3.2 The Modelling Language ART-ML

The ART-ML language describes a system a set of tasks and mechanisms for synchronization, where each task has an attributes part and a behaviour part.

The behaviour part of a task is described in an imperative language, C extended with ART-ML primitives, and describes the temporal behaviour and, to some extent, the functional behaviour. The attributes are the scheduling priority and how the task is activated, one-shot, periodically or sporadically. The behaviour is described in C, extended with ART-ML primitives and routines, e.g. for sending and receiving messages to message queues, semaphore operations. The execution time for a block of code is modelled with a special statement for consuming time, *execute*. The execute-statement is used for modelling sections of code from the real system by their execution time only. The execution time is specified as a discrete probability distribution.

An example of a task in ART-ML is:

```
TASK SENSOR
   TASK_TYPE: PERIODIC
   PERIOD: 2000
   PRIORITY: 1
BEHAVIOR:
   execute ((30, 200), (30, 250), (40, 300));
   sendMessage(CTRLDATAQ, MSG_A, NO_WAIT);
   chance(20){
      execute ((60, 200), (40, 230));
      sendMessage(CTRLCMDQ, MSG_B, FOREVER);
   }
END
```

The *chance statement* in ART-ML provides a probabilistic selections, i.e. a non-deterministic selection with probability. Chance express the probability of a particular selection. A chance statement can be used for mimicking behaviours observed in measurements of the real system, where the exact cause is not included in the model.

A message queue is a FIFO buffer storing messages. The message queue declaration contains the name and the size of the message buffer. A message is sent to a message queue using the sendMessage and a message is read from a message queue using the recvMessage.

An ART-ML semaphore provides mutual exclusion between tasks and conforms to the concept of the classic binary semaphores proposed by Djikstra. A semaphore is locked using the sem_wait and released using sem_post routine.

3.3 Analysis of system properties using PPL

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. For this purpose we have developed a property language called Probabilistic Property Language (PPL). This language was first proposed in [4]. We have recently implemented a tool for evaluating PPL queries, described in Section 5.3.

The analyses of system properties in our implementation of the ART Framework are based on trace of the dynamic behaviour, either from a real system implementation or from a simulation based on a ART-ML model. Based on the recording we evaluate the system properties specified in PPL.

System properties related to deadlines is a requirement on response times, either related to a particular task or features involving several tasks, i.e. *end-to-end response time*. A deadline property can be formulated as hard, 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. In PPL, this property is formulated as follows:

```
P(TaskX.response < 1200) >= 0.9
```

Another property of interest could be the separation in time between instances of a task. The following PPL query checks if two consecutive instances of a task can be separated in time with less than 1000 time units.

```
P(TaskX(i+1).start - TaskX(i).end >= 1000) = 1
```

A PPL query may contain an unbounded variable. If replacing a constant value with an unbounded variable, the PPL tool calculates for what values the query result is true. The following PPL query finds the tightest deadline D that TaskX meets with a probability of at least 0.9.

```
P(TaskX.response < D) = 0.9
```

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 message queue is always non-empty. In PPL, this is formulated as follows:

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

In this query above, it is assumed that the number of messages in the critical message queue is monitored using probe 21. The asterisk specifies that the condition should hold at all times. If a task name is specified instead, it means that the condition should hold when instances of that task starts.

3.4 Uses of the ART Framework

The ART Framework was initially developed for impact analysis, to predict the impact on timing and resource usage caused by a change. However, we discovered another use of the ART Framework, regression analysis, to analyse the current implementation and compare with previous versions. In this section these uses are described.

Impact Analysis If a model has been constructed as described in Section 3.1 it can be used for predicting the impact an maintenance operation will have on the runtime behaviour 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 an 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 case, 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, specifying for instance a maximum allowed execution time.

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 step one trough three in the process shown in Figure 1, i.e. updating the model structure, profiling and validation.

The main problem with impact analysis is how to validate models. This is however not a problem unique for our approach, any formal analysis based on a model has this problem, if the model has been re-engineered from an existing system. In this paper we have presented a method for model validation (see Section 4), and in future work we intend to investigate other complementary methods.

Regression Analysis Another use of the framework is regression analysis, i.e. to compare properties of the current release of the system with respect to certain invariants and with previous versions of the system. This is very close to regression testing, but instead of testing the functional behaviour, timing and resource usage are analyzed. It is also possible to compare the analysis result with earlier versions of the system. In this way, it is possible to study how the evolution of the system has affected the properties of interest. It might be possible to identify trends in system properties that could cause problems in future releases. If a model has been developed, the impact analysis can be used in order to predict how an extrapolation of a trend will affect the system.

In order to use regression analysis in a development organization, there is an initial effort of specifying the properties of interest, formulate them as PPL queries, define comparison rules and instrument the system with the appropriate software probes. The setup of the system should be specified in a document, in order to allow measurements to be reproduced. It is possible that different properties require different system setups, in that case multiple measurements of the system is necessary.

After this initial work, performing a regression analysis is straightforward and can be performed as one of many test-cases by a system tester, without deeper system understanding or programming knowledge. Measurements are made according to the documents initially produced. This results in execution traces, which are analyzed and compared with earlier releases, using a highly automated tool. Based on the comparison rules, the tool decides if there are alarming differences and informs the user of the outcome. A tool supporting this is presented in Section 5.3.

We are working on this approach in tight cooperation with ABB Robotics where we intend to introduce regression analysis. We have already integrated our recording functionality in their robot control system, which allows them to use our tools. The overall reaction among key persons at the company has been very positive. Before the method can be fully utilized at ABB Robotics, the relevant properties and the system setups used for the measurements must be specified.

4 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 is not feasible, since the model is a probabilistic abstraction of the system. 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 of the comparison properties match the observations from the real system, the model is *observable property 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 behaviour 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 (inter-arrival times, precedence, pre-emption).

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 behaviours and semantic relations are indeed captured by the model at an appropriate level of abstraction.

4.1 Model Robustness

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 behaviour predicted by the model in the same way as they affect the observed behaviour 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 2. 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.

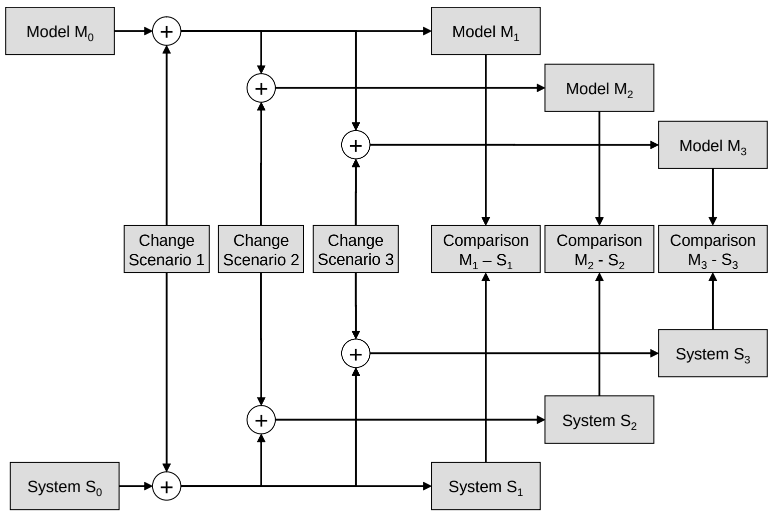


Fig. 2. The Sensitivity Analysis

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 constructed in a similar way, by introducing the corresponding changes in the initial model M_0. Note that these changes only need to reflect the impact on the temporal behaviour and resource usage caused by the change scenarios, they do not have to be complete functional implementations. Each model variant is then compared with its corresponding system variant by investigating if they are equivalent as described in Section 3.3. If all variants are obserable property equivalent, including the original model and system, we say that the model is robust.

5 Tools in the ART Framework

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

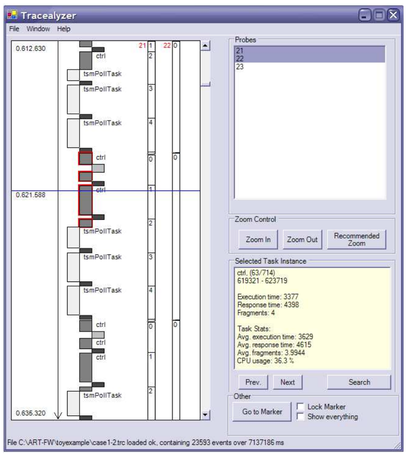
– An ART-ML simulator, used to produce execution traces based on an ART-ML model.
– The Tracealyzer, a tool for visualizing the contents of execution traces and also allows PPL queries to be evaluated with respect to execution trace.
– The Property Evaluation Tool, PET, a tool for analysis and comparison of execution traces.

5.1 The ART-ML Simulator

The ART-ML Simulator has a graphical front-end with an integrated model editor, making it easy to use. This is not a simulator in the traditional sense, i.e. a general simulator application reading models as input. Instead, when the user clicks on the simulate-button, it translates the ART-ML model into ANSI C, compiles it using a standard C-compiler and links it with an ART-ML library. This results in an executable file, containing a synthesis of the ART-ML model. The synthesized model is executed for the specified duration, which produces an execution trace.

5.2 The Tracealyzer tool

The Tracealyzer has two main features, visualization of an execution trace and a PPL terminal, a front-end for the PPL analysis tool. The execution trace is presented graphically. Tasks and generic probes are presented in parallel, allowing correlation between the task scheduling and the task behaviour. It is possible to navigate in the trace by using the mouse and also to zoom in and out and to search for task instances or probe observations with different characteristics. If the user clicks on a task instance, information about it is presented, such as the execution time and response time of the instance and the average execution and response times for the task. If more task statistics are desired, it is possible to generate a report, containing a lot of information about all tasks.



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. 3. The Tracealyzer Tool

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 Tracealyzer. 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.

5.3 Property Evaluation Tool

The Property Evaluation Tool, PET, is a tool for analysing and comparing execution traces with respect to different system properties, formulated in PPL (described in Section 3.3). The operation of PET is rather simple. The user selects a file containing a predefined set of system properties, formulated as PPL queries. The file can also contain a comparison rule for each property, specifying what results that is acceptable and what is not. The user then only has to select the execution trace(s) to analyse.

If desired, two execution traces can be specified, but it is also possible to analyse a single trace. Specifying a second execution trace allows the tool to do an automatic comparison of the results using the comparison rules. When the user clicks on the analyse-button the properties are evaluated with respect to the trace(s) and the results are presented. If two traces are specified, the properties with comparison rules are compared automatically. Any properties where the rule has been broken are pointed out.

The application has three uses: impact analysis, regression analysis and model validation. In the impact analysis case, execution traces from simulation of two 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.

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.

In the regression analysis case, no data from simulation of models are used. Instead, execution traces measured from different versions of the system are analyzed and compared, in order to identify trends and alarming differences, which might be a result of undesired behavior in the system.

6 Benefits for Industry

If impact analysis can be performed when designing a new feature or other vast changes in the system, bad design decisions can be avoided. The designer of the feature can try alternative designs on a model and predict their impact on the system. This is likely to decrease maintenance costs since problems with timing and resource usage can be identified before implementation. Consequently, the time for identifying errors related to timing in late testing phases is reduced which decreases the cost for maintenance. This also leads to better system reliability.

Regression analysis and trend identification can point out undesired behaviour in the system that reflects in the system properties of interest, for instance execution times. It can also be used for performance analysis, by pointing out bottlenecks in the system. Information about trends in system properties can be used to plan ahead for hardware upgrades in the product which also is an important maintenance activity. If a trend in a property has been identified, impact analysis can be used to predict how the system will behave if a trend continues, for instance if a certain execution time keeps increasing as the system evolves.

The graphical visualization of execution traces, provided by the Tracealyzer tool is, according to our experiences, an effective way of increasing the understandability of the system. When the tool was introduced to developers and architects at ABB Robotics, showing them execution traces from the latest version of their system, we got immediate reactions on details and suspicious behaviours in the execution trace. We provided them with a new view of the run-time behaviour, increasing the understandability and facilitating debugging activities.

The results we have gotten so far from using the ART Framework at ABB Robotics indicates that maintenance costs can be reduced, as it enables impact analysis, regression analysis and significantly increases the system understandability. Even though the deployment of the framework is in an initial phase it has already pinpointed anomalies in the timing behaviour that were not previously known. Based on discussions with system architects, we believe that by deploying regression analysis we can reduce maintenance costs at ABB Robotics. As mentioned, we are working on introducing regression analysis in the company and later, when this has been used for a while, we plan to investigate the actual impact on maintenance costs.

We believe that introducing impact analysis could further reduce maintenance costs, as it helps system designers taking the right design decisions. Further research is however necessary in order to validate this approach.

7 Conclusions

In this paper we have briefly presented the ART Framework; the general ideas, the languages ART-ML and PPL. We have presented the three tools developed for this framework and an approach for validating ART-ML models. We have presented how the framework can be used for impact analysis, regression analysis and how the industry can benefit from these uses of the ART Framework. We have also presented our experiences from deploying parts of the framework in a development organisation, which strengthen our hypothesis that maintenance costs can be cut by introducing the methods proposed in the ART Framework.

We believe that this approach is very useful for its purpose, analysis of properties related to timing and resource utilization, targeting complex real- time systems. However, there is work remaining before we can validate this approach.

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 behaviour or a hybrid approach. This is part of our future work.

Further we intend to perform two case studies on the two uses of the ART Framework. In the first case study, we plan to further investigate the benefits and problems associated with deployment of regression analysis. We also intend to do a continuation of the case study on impact analysis, presented in [2], using a more advanced model and the tools presented in this paper. Later on, when this framework has been in use for some time, we plan to investigate how the maintenace cost at ABB Robotics have changed, by analysing the company's fault report database.

References

1. Schutz, W.: On the testability of distributed real-time systems. In: Proceedings of the 10th Tenth Symposium on Reliable Distributed Systems, (IEEE) 52–61
2. Wall, A., Andersson, J., Neander, J., Norström, C., Lembke, M.: Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In: Proceedings International Conferance on Real-Time Computing Systems and Applications. (2003)
3. Wall, A.: Architectural Modeling and Analysis of Complex Real-Time Systems. PhD thesis, Mälardalen University (2003)
4. Wall, A., Andersson, J., Norström, C.: Probabilistic Simulation-based Analysis of Complex Real-Time Systems. In: Proceedings 6th IEEE International Symposium on Object-oriented Real-time distributed Computing. (2003)
5. Audsly, N.C., Burns, A., Richardson, M.F., Wellings, A.J.: Stress: A simulator for hard real-time systems. Software-Practice and Experience **24** (1994) 543–564
6. Storch, M., Liu, J.S.: 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)
7. Manolache, S., Eles, P., Peng, Z.: 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. Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., , Wellings, A.J.: Fixed priority pre-emptive scheduling: An historical perspective. Real-Time Systems Journal **8** (1995) 173–198
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in hard-real-time environment. Journal of the Association for Computing Machinery **20** (1973) 46–61
10. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994)
11. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Springer International Journal of Software Tools for Technology Transfer **1** (1997)
12. Muller, H., Klashinsky, K.: Rigi: a system for programming-in-the-large. In: Proceedings of the 10th International Conference on Software Engineering. (1988)
13. Rigi Group: (Rigi Group Home Page) http://www.rigi.csc.uvic.ca/index.html.
14. Scientific Toolworks: (Scientific Toolworks Home Page) http://www.scitools.com/.