Model Synthesis for Real-Time Systems

Joel Huselius and Johan Andersson
Mälardalen Real-Time Research Centre,
Mälardalen University, Västerås, Sweden
{joel.huselius, johan.x.andersson}@mdh.se

Abstract

In this paper, we present a method for model synthesis. Based on observations of running system, a model that can describe the observed behavior is automatically generated. This allows faster and more accurate modeling of existing systems. The models can be used for impact analysis, verification, documentation etc. The method has been implemented; we describe that implementation and present an evaluation of its performance, the conclusion of the evaluation is in favor of the proposed method.

1. Motivation

From observations of an implemented real-time system, we show how to synthesize a probabilistic state-machine model expressed in the ART-ML [15] language. We require only a few simple inputs and a recorded execution trace of the running system – the fact that the inputs are few, simple, and general contributes to the generality, portability and ease of use of our proposed method.

Often, models are created manually from the specification before the implementation in order to prove that an architectural solution can provide desired properties. There are, however, situations where models of existing systems can be interesting:

- When planning to add new functionality to a system, model synthesis can provide an up-to-date model of the existing system which can then be manually altered to reflect the planned alteration – the modified model can then be analyzed to quantify the effects of the alteration. This is often referred to as *impact analysis*.
- Synthesized models provide an abstraction of the real implementation, this abstraction can be used in debugging and documentation of the system as well as being a tool for understanding, i.e. by engineers new to the project.
- COTS-technology allows for insertion of third-party code into the system. However, purchased components and models accompanying these must be verified to comply to the specification – by bi-simulating the synthesized model with the provided model, model synthesis allows verification of accuracy of the accompanying model in relation to the implementation.
- Using computers to synthesize models is expected to be faster and less error prone than manually created models which requires specialists.

1.1. Outline

The remainder of this paper is organized as follows: Section 2 describes two approaches for model synthesis, the modeling language of our choice, and our relation to other work.

Section 3 describes our methodology and its implementation, after which Section 4 presents an evaluation of that implementation. Section 5 presents ideas for future work, and the paper is concluded in Section 6.

2. Background

We can envision implementation of model synthesis through either a *static* or a *dynamic* method:

Static methods work by analyzing the specification or the source code of the implementation offline - the process is similar to translating the implementation to a different language. Dynamic counterparts (e.g. Jensen 1998 [8]) function by watching the running system and estimating a model that can deliver the observed behavior.

Due to the dependence on specification or source code as input, it seems reasonable to assume that static methods are dependent of the language used for the implementation of the run-time system, and may thus encounter difficulties when languages are mixed in the same system. We suspect that fundamental differences between languages (pointers vs. no pointers, object oriented vs. imperative, etc.) may prevent the generality of such methods.

On the other hand, dynamic methods can only model the behavior that has actually been observed, which is likely to be only a subset of the valid system behavior (compare to the completeness problem [7]). Further, the synthesized model is dependent on the observations on the run-time system – if no observations can be made, no model can be created (compare to the observability problem [7]). Finally, dynamic methods depends more heavily than their static counterparts on the interpretations of the observations made and the deductions made from these interpretations.

As a consequence of our background in debugging using record/replay [7], we choose to pursue dynamic methods of model synthesis.

2.1. ART-ML

We have chosen ART-ML (Architecture and Real Time behavior Modelling Language) [15] as modeling language. The major reason for our choice is the probabilistic properties of the language that seems to rhyme well with the fact that observations may not be able to provide all details about the implementation.

ART-ML has been introduced to allow modeling of complex real-time systems. Provided is an FPS (fixed priority scheduling) simulator where ART-ML models can be tested and evaluated. The intention is to facilitate a model of the system that can be altered in order to reflect the intrusion by future changes to the implementation. It is assumed that it is easier to modify the model than to modify the actual implementation. Thus, if the altered model is successfully evaluated with respect to resource availability and temporal requirements specifications, the confidence in the proposed implementation can be increased. We believe that this will lead to less dead-ends and less ad-hoc alterations in order to make the implementation answer to its requirements.

As the ART-ML model provides a very high-level view of the system, the logic for selecting different behaviors might not be available in the model. ART-ML solves this by providing a probability model where runtime selections can be resolved by chance. Relieving the model from much of the implementation details moves the focus from low-grained functional issues to architectural issues and temporal behavior.

2.1.1. Example In Figure 1, we provide a small example of a task modeled in ART-ML. Among the set of reserved words in ART-ML, the following subset is important to the contents of this paper (for a more detailed description, we refer to [15]):

`if, else`: works intuitively, as in e.g. C or Java. Variables can be defined and modified to provide input for the selection.

```
task AAA
 trigger period 1300
 priority 254
 deadline 1300
behaviour{
 chance(60){
  execute((20,100),(30,130),(50,200));
 }
 else{
  execute((50,200),(50,210));
  snd(MBOX0,0);
  chance(50){
   snd(MBOX0,1);
  }
 }
}
```

Figure 1. ART-ML example.

`chance, else`: a selection that takes a probability as input and makes a selection based on that value during run-time.

`execute`: taking a series of tuples (probability, execution time) as input, the execute-statement can represent computation with varying execution times.

`snd, recv`: provides means to perform inter process communication.

2.2. Related work

Jensen [8] presents dynamic model synthesis of real-time systems in UPPAAL [3]. However, UPPAAL has no notion of probability, and is therefore not capable of presenting such a nuanced model as can be achieved using ART-ML. Also, the models presented, and the environment where they are used are simple and put high predictability requirements on the implementation (the use of loops and selections is restricted, and the environment only supports preemptions at designated points). More recently, Grinchtein et al. [6] proposed learning for dynamic synthesis of UPPAAL models.

Sifakis et al. [13] propose a static method that uses tagging of the real-time software with time constraints to facilitate model synthesis. It seems that quantifying the time constraints is a difficult sub-problem that requires much care and planning in order for the method to yield a useful result.

Bastos and Sanches [2] propose a static method that, based on UML (Unified Modeling Language) models, produces SDL (Specification and Description Language) models. The method is inherently object oriented, and does require some additional input from humans.

Yan et al. [17] present a dynamic method for synthesizing high-level architecture models from non-real-time system implemented in Java. Their models describe only the architectural structure of the system (without behavioral

descriptions), which has less information content than the ART-ML models described here.

Moe and Carr [10] present a dynamic method that use the CORBA notion of *interceptors* to introduce probes transparently to the application. Recordings are later used to analyze RPC call-patterns that present a high-level view of the interactions in a distributed non-real-time system. No actual model is produced from the recordings, but the tool Spotfire.net is used to visualize these interactions - the tool is similar to the Tracealyzer used in the validation part of this paper. A real system is used in an experiment to verify the method.

Briand et al. [4] propose a dynamic method to synthesize UML sequence diagrams. Even though sequence diagrams can be helpful in the effort to understand the system and verify that a known functionality is performing as intended, it is uncertain if the overhead is justified. For example, the representation, while able to describe the act of making a selection, can only describe the one path of the selection that was actually performed (e.g. only one path of the selection can be modeled in a given sequence diagram). Further, the model cannot represent state, and can therefore not be used in simulations etc. In order to justify the overhead, it should be shown that the amount of recording performed is sufficient to produce also other types of UML models of the system.

Richner and Ducasse [12] present a hybrid solution that use both statically and dynamically obtained data to synthesize models of object oriented non-real-time systems implemented in Smalltalk. This will provide for a more comprehensive picture than a strictly dynamic or static approach. However, their modeling language of choice cannot describe the nature of choices made in the execution of the system (e.g. variables and values that determine selections), which limits the applicability of the model.

In [9], Marburger and Westfechtel reported on set of reengineering tools, developed in cooperation with Ericsson Eurolab Deutchland, including support for both structural analysis and behavioral analysis. The behavioral analysis includes state machine extraction from PLEX source code (an asynchronous real-time language). Traces recorded from a system emulator can be used to animate the state machines in order to illustrate the system behavior.

Another interesting work is that by Systä and Koskimies [14] where state diagrams are synthesized from traces. The source code of the system in focus is instrumented in order to generate a trace. The trace is then fed into the SCED tool, which generates a (minimal) state diagram corresponding to the observed behavior. The work does however not address real-time systems, no timing information is recorded.

3. Implementation

In this section, we explain our dynamic method for model synthesis, and provide details of our implementation of the same.

3.1. System model

For the remainder of the paper, we will assume the following system model:

The system is a real-time system of *tasks* that each execute *jobs* (a.k.a. instances) with approximate frequency. Between two jobs, the task is in idle. Jobs can communicate only via a queue-based protocol for inter process communication (ipc) using system calls.

Each task can have at most one job at a given time, and each job is triggered by an event of known type; these are either: the elapsing of a relative timer, i.e. a periodical task, or the arrival of a new message on a known IPC queue. Triggering of a new job for a given task is performed on one and only one designated event (i.e., a task cannot listen to more than one queue simultaneously or have more than one timer simultaneously).

Further, we assume that the operating system provides an interface to execute code at both context-switches and system calls. This functionality is available in commercial operating systems such as VxWorks.

We assume that a set of properties are known for each task that should be modeled: the priority of the task, the identification of the task, and the method of triggering a new job of the task. Of these, the triggering property deserves some detail as it is used to identify the end of each job: The triggering method can either be *periodic* (with a time frequency), or *event driven* (with a queue to which new messages activates a new job). If the triggering method is periodic, the logs of the recording are analyzed to determine the frequency based on the task-state at preemptions. In the case of event driven triggering, the end of each job can be backtracked from each completed receive.

3.2. Recording

We abide by the following terminology: By inserting *probes* into the system, we can *monitor* the *events* that occur during execution. The output of monitoring can be *logged* to facilitate offline analysis of the execution. Monitoring and logging are grouped into the activity *recording*. Our method for model synthesis is dependent on recording to produce inputs to the modeling process.

Generally, probes can be implemented in hardware, software or some hybrid – for portability and simplicity we have used software probes in our implementation. The probes are tailored to extract some information from the system, this

Event	Parameters
Context switch (A for B)	time, state of A, id of A, id of B.
Send to ipc-queue	time, queue id.
Read from ipc-queue	time, queue id, timeout.
State assignment	time, value, variable name.

Table 1. Events and their parameters.

extraction comes at the price of perturbation to the original system. Probes that are highly perturbing (i.e. software implementations) should remain resident in the system (see [7] for further elaboration on this and other issues related to recording).

Our method requires generic probes to hook on to *context switches* and *system calls*. There is also an option to use *state-probes* that record the values of selected variables to represent the state in the system. For each event, a set of parameters are logged as described by Table 1.

The drawback with the optional state-probes is that the use normally requires access to the implementation code and the possibility to modify it – the mandatory probes on context-switches and system calls can be added in an operating system abstraction layer. Modifying the source, however, requires a white-box rather than a black-box view of the implementation.

That situation may be possible to work around if the application uses a data-base such as that described in [11]. In such a system, the data-base can be accessed by an observing probe transparently from the system and without treating the system as a white-box.

3.2.1. From recorded job-sequences to task-trees For the moment, let us assume that we use only the mandatory probes on context-switches and system calls, we will discuss the use of the optional state-probes later in the paper.

We make recordings of the system to model and analyze these on a task-basis (using recordings of context switches to differentiate between tasks). For each job of a task, we get sequences of actions such as: `execute 20 time unit (tu)` → `receive from queue 1` → `execute 10 tu` → `end job`. Each such sequence is referred to as a *job-sequence*.

Please note that each action is represented by a *type* (e.g. send, receive etc.) and a set of type-specific *properties* (e.g. queue identifier, execution time etc.). The properties of each action are the values of the parameters logged with that action as described by Table 1.

In job-sequences, we let execution-actions be associated with the next up-coming event, but no further effort is made to distinguish between different execution-actions. Remaining actions are distinguished by their type and some property of that type:

Thus, when comparing a `receive from queue 1`-action with a `receive from queue 2`-action, we come to the conclusion that they are not equal, while a `execute 20 tu`-action is equal to a `execute 30 tu`-action provided that they are followed by identical actions.

We then proceed to construct the simplest tree of actions that can represent the set of job-sequences. This is labeled the *task-tree*. In the task-tree, leafs are actions describing execution and system calls (execute, send, or receive, etc.), and properties of these leafs describe the detail of the particular action (e.g. time, queue, etc.).

Note that, as the execute-statements are associated with the up-coming event, two job-sequences: `execute 20 tu` → `receive from queue 1` → `end` and `execute 20 tu` → `receive from queue 2` → `end` will lead to a tree where the two branches have no common leaf.

Similarly, the job-sequences: `execute 20 tu` → `receive from queue 1` → `end` and `execute 30 tu` → `receive from queue 1` → `end` will lead to a tree with a single path, but the execute-leaf of that tree will hold information about one occurrence of `20 tu`, and one occurrence of `30 tu`.

3.2.2. From extracted task-trees to ART-ML code From the task-tree, we can then construct the ART-ML model. Figure 2 describes this process graphically.

The property of the execute-action is the time spent to execute. As data is collected and many versions of the same tree-leaf are discovered, these will be represented as a sequence of execution times. When the code is generated, the quartiles of this sequence are used to describe the interval. The quartiles are: the minimum value, the 25^{th} percentile, the median, the 75^{th} percentile, and the maximum value. Probabilities for these are used to describe the distribution over the sequence.

If the leafs of one action in the tree are more than one, this will be represented by a chance-selection when the model is constructed from the task-tree.

3.2.3. An example To clarify, we will briefly describe the example displayed in Figure 2.

First, the trace collected from the running implementation is collected and analyzed. Known are the name of all tasks in the taskset and their individual priorities (this information can also be collected from the running system). Also known, for every task, is the method of triggering new job of the task (periodic or on event from a known queue).

The trace is analyzed to separate the different jobs (the job-sequences) of individual tasks. These are shown in the left-most part of Figure 2, for example, the first job-

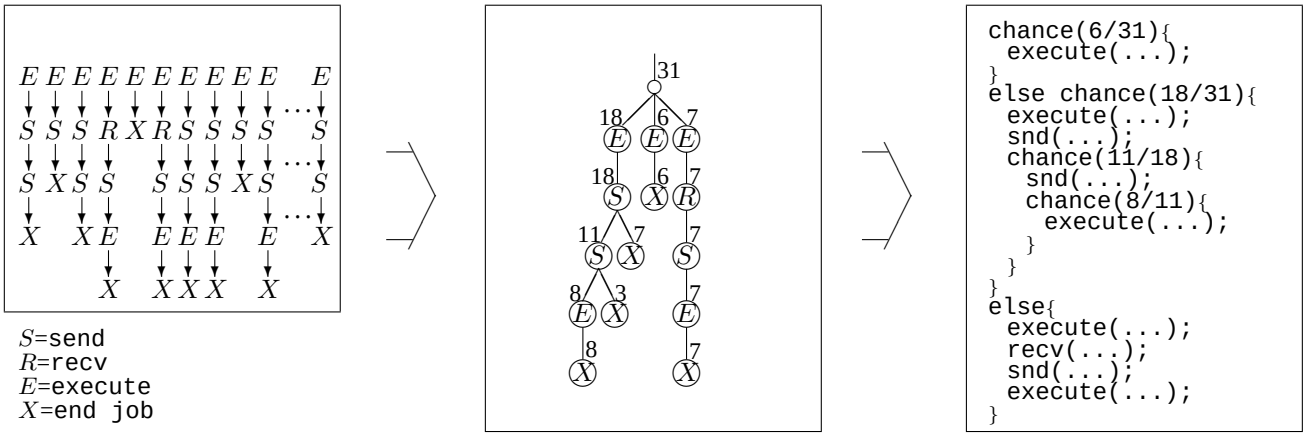

Figure 2. Modeling with chance-statements assuming system-level recording only.

sequence reads `execute` → `send` → `send` → `end` (for simplicity, properties of the action-types, such as queue identifiers, are ignored).

For each task, all job-sequences are then joined into a task-tree that describes observed actions and their properties. This is shown in the middle of Figure 2.

Please note the way the three execute-actions at the very root of the task-tree are separated, this is because of that the second action of the sequences observed have differing types: 18 had send-actions, 7 had receive-actions, and 6 sequences end directly after the execute action. It is assumed likely that the distribution of the execute-actions may differ significantly based on the path about to be taken, which is why they are separated.

Finally, the tree is traversed in preorder and the code corresponding to each visited action is output. If there are reasons for selection, `chance`-statements are inserted appropriately; probabilities are calculated based on the observed frequencies. This is described in the right-most part of Figure 2.

3.3. Adding a notion of state to the model

The modeling described so far does not respect the state, or the semantics, of the system. By adding the optional probes that record values of variables at selected points, the model synthesis is made more accurate to the implementation. The drawback, and the reason to present this optional recording separately, is that the application is no longer a black-box to the recording. Thus, the source of the implementation must be available and possible to modify. Obtaining the source code of the implementation may present a problem, but the gain is such that we will describe the procedure:

In the job-sequences, similar to entries for system calls, we add actions that describe altering of the state of the system. There is no limit set to the amount of variables that can be used to represent this state. Different update-actions are separated by both the variable updated and the value assigned to that variable (as described by Table 1).

When compiling the task-tree with update-actions present in the job-sequences, the compilation must respect the current state of the task when events are compared. As a leaf is updated, the current state is therefore added to a set of valid states for that leaf. Also the number of occurrences for different states is recorded.

Thus, we note that adding state information will not lead to a task-tree with more leafs, but the leafs will be larger. The reason for why we do not use state-information when comparing actions for the compilation of the task-tree will become evident in the next section where this design-choice saves considerable effort.

3.3.1. Counting states The representation of state in the ART-ML model can either be in the form of a general assignment based on unknown premises (e.g. reading of a sensor) or as an operation on the previous state (e.g. increase of a counter). To handle both these, we need ways to express them in the task-tree.

Concerning the general assignment, this is already expressed with the update-action. Subsets of the update-actions may however be exchanged for the optional state operation as described above – leading to smaller trees, which in turn will lead to smaller models that are easier to understand. To implement this, we need to apply analysis to the task-tree:

The procedure is to make a premise and to search for a counter example on the recording available. For example, assuming a variable v, the premise can be that the update-action is an increment-operation on the form $v_n = v_0 + (n * C)\ modulo\ K$ where v_0 is the initial value of v. (The values of v_0, C, and K are measured on the update-

```
Poor modeling:                  Good modeling:
if(va==1)                       if(va==1 || va==2)
  f();                            f();
else if(va==2)
  f();
```

Figure 3. Poor and good modeling practice.

actions that are to be compared). If the premise is not found to be faulty, we can assume that the set of update actions that are covered by the premise can be seen as equal actions.

The design choice described in the previous section will result in that all the update-actions that may be concatenated will have the same root in the task-tree. Thus, the search for a counter example to the premise is limited to the leafs of one root, and if no counter example can be found, all that remains to perform is to register the finding in the update-action and to merge the trees that have the examined update-actions as their root.

3.3.2. State-change and path explosion An interesting conflict arises from the strive for "better" models:

We should strive for smaller models, which includes using good modeling practice as seen in Figure 3. If state-variables are not respected when comparing actions during compilation of the task-tree, this can be efficiently ensured.

However, this will result in that several disjunctive executions share paths in the task-tree, and care must be taken when generating the model-code. Contrary to the intention, the condensed representation may lead to a situation where, after an update-action, the number of possible paths seems larger than intended.

Imagine a set of job-sequences describing a system that can experience a state change through an update-action on a single variable. Say that of the sequences $\{A,B\}$, sequence A has a prefix Ax followed by an update-action transferring the state from a to b, after which the job-continues. Now, if B has the same prefix as A, and the same update-action, but a different continuation, we must take care. The internal state added to actions following the state transition of A must, for the duration of A, remain to be a.

Failure to implement this *lazy state transition*, would lead to that the model describes the behavior of B for both states. Further, it would be more complicated where there more than two jobs in the same style.

3.4. Limitations and expected problems

There are a number of issues with the current version of the model synthesis, some are possible to amend, and some are inherent of the approach.

Currently, we only support two system calls: send and receive over inter process communication queues. This is indeed a limiting factor, but we expect no problems in extending our tool to support other system calls such as semaphore operations etc.

As the approach chosen is working with observations of occurred behavior as only input, we can not ensure that the model generated describes the implementation in every respect (compare to the completeness problem [7]). This could possibly be amended by combining the tool with a static model synthesis, or by using a limited amount of manual modeling.

Further, the probabilistic nature of ART-ML may lead to that worst case execution times are over or under estimated, and that best case execution times are under estimated. Imagine a trace through the model of a task that passes two execute-statements in the same job of the task. In the real implementation, it may be that executing for example a low time-count in the first execute-statement will lead to that the second statement must execute a high time-count. This implicit knowledge is not necessarily incorporated in the ART-ML model, which is why the spectra of modeled execution times may cover a larger interval than possible in the run-time system. This can of course be amended by incorporating the optional state-information into the recording effort.

4. Model Validation

In order to validate this method of model synthesis, we have compared observations of a system with predictions from a model, synthesized from that system using the method presented in this paper. The validation was done using the ART-ML simulator [15] and a recently developed set of tools called the ART Framework [1]. This framework contains two tools, the Property Evaluation Tool, an analysis tool for PPL queries [16], and the Tracealyzer, a graphical execution trace browser.

The system that was modeled was a small Real-Time System previously developed for similar experimental purposes. The system pretends to control an electric motor based on sensor readings. However, the system does not do any real calculations and does not interface any sensor or motor electronics, but the tasks in this system have realistic execution times distributions and communicate using a commonly used mechanism, IPC message queues. The temporal behavior of this system have many similarities with industrial control systems we have studied, for instance the robot control system described in [15].

The system was implemented on a platform commonly used for real time and control applications, VxWorks from WindRiver, a real time operating system which uses preemptable fixed priority scheduling. The system basically

Task	Priority (%)	Frequency (%)
Sensor	High	High
Drive	Medium	Medium
Ctrl	Low	Low

Table 2. Task properties.

Task	Predicted (%)	Observed (%)
Sensor	15.15	15.04
Drive	5.03	5.03
Ctrl	38.30	37.55

Table 3. CPU utilization, predicted and observed.

consisted of three tasks (see Table 2); `Sensor`, `Ctrl` and `Drive`. The task `Sensor` executes periodically, with a high frequency and the highest priority. The task `Ctrl` has low priority and is event triggered. The `Ctrl` task has rather long execution time and is therefore pre-empted several times. The `Drive` task has medium priority and executes periodically, but with a lower frequency than `Sensor`.

The model was executed in the ART-ML simulator in order to produce an execution trace. The resulting execution trace was analyzed with respect to a set of system properties acting as a point of view for the validation. The properties were formulated as PPL queries and then analyzed using the Property Evaluation Tool, with respect to both the execution trace from the simulation and the execution trace measured on the real system.

The final step in the validation was to compare the results from each analyzed property. First we compared the CPU utilization of the tasks. For this, we used the Tracealyzer tool which has a feature for presenting the tasks CPU utilization.

As you can see in Table 3, the model predicted the CPU utilization of the tasks with high accuracy. However, the CPU utilization is only a weak indication of the validity of the model, since it is only represents the average values on execution time and inter-arrival time of the tasks.

In order to further investigate the model validity, the distributions of the execution times and response times were compared using the Property Evaluation Tool. Five types of properties were used for this:

- The maximum observed execution/response time of an instance of the task (`maximum`).
- The average execution/response time of a task (`average`).
- The highest execution/response time that at least 10 % of the instances exceed (`highest 10`).
- The highest execution/response time that at least 25 % of the instances exceed (`highest 25`).
- The lowest execution/response time that at least 10 % of the instances are below (`lowest 10`).

These properties were used both for execution times and response times, for the three tasks `Sensor`, `Ctrl` and `Drive`. When analyzing these 30 properties using the Property Evaluation Tool, we got the values presented in Table 4 and Table 5. The third column in the tables, accuracy, is the accuracy index, the quotient between the predicted and the observed values, times 100. For the execution time properties, 13 of the 15 predictions have an accuracy index between 96.7 and 103.7, where 100 is a perfect match. However, when it comes to the response time properties, the predictions are not as good. The following discrepancies were observed:

- In the predictions of the response times for `Drive`, the accuracy index is around 50, i.e. the predicted values are only 50 % of the measured ones.
- The accuracy index of the response times for `Ctrl` was between 79.17 and 96.75, i.e. significantly lower in the predictions than in the measurements.
- In the measurements of the real system, the highest observed response time of `Sensor` was 944 us. The corresponding value from the simulation was only 367 us.

	Execution Times		
	Pred. (us)	Obs. (us)	Accuracy
Task ctrl			
maximum	4398	4366	100.7
average	3811	3756	101.5
highest 10	4372	4215	103.7
highest 25	4359	4038	108.0
lower 10	3197	3306	96.7
Task drive			
maximum	580	579	100.2
average	509	505	100.8
highest 10	562	550	102.2
highest 25	551	535	103.0
lower 10	447	459	97.4
Task sensor			
maximum	367	906	40.5
average	303	302	100.3
highest 10	332	329	100.9
highest 25	320	310	103.2
lower 10	277	282	98.2

Table 4. Execution time distribution, predicted and observed.

Using the Tracealyzer tool, we could quickly identify the causes of these three discrepancies. The first two turned out to be caused by a sleep-operation in the task `Sensor` in

	Response Times		
	Pred. (us)	Obs. (us)	Accuracy
Task ctrl			
maximum	5161	5986	86.2
average	4488	4952	90.6
highest 10	5123	5415	94.6
highest 25	5066	5236	96.8
lower 10	3561	4498	79.2
Task drive			
maximum	580	1126	51.5
average	509	1052	48.4
highest 10	562	1099	51.1
highest 25	551	1082	50.9
lower 10	447	1006	44.4
Task sensor			
maximum	367	944	38.9
average	303	302	100.3
highest 10	332	329	100.9
highest 25	320	310	103.2
lower 10	277	282	98.2

Table 5. Response time distribution, predicted and observed.

the real system. The model synthesis does not yet recognize that system call, so synthesized model did not include any sleep-operation.

Because of this, the response times of the `Drive` task were much higher in the real system than in the prediction from the model. The purpose of the sleep was to make the task execute with an offset. This however causes the `Drive` task to be split in two fragments, and our analysis tool calculated the response-time to include both fragments and the sleep period in between. The sleep-operation also caused the difference in response times for the `Ctrl` task. The offset caused by the sleep makes `Drive` pre-empt `Ctrl`. This does not occur in the simulation.

Regarding the very long maximum response time of `Sensor` that was observed on the real system, it turned out that the first instance of `Sensor`, in the real system, had an unusually high execution time of 906 us and was also pre-empted by another task, which executed for 38 us. The sum of these execution times equals the observed response time, 944 us.

This behavior did not occur in the simulation. There are two possible reasons; either did the model not include this behavior, or the behavior was indeed in the model, but was never executed as it a low probability of occurrence. It turned out that the model did not include this very high execution time. This since the model synthesis tool deliberately filters out the first instance of tasks when it constructs the model.

The motivation for this is that the first instance of a task tends to have a rather different behavior, as in this case, and it only occurs once in a recording. To model a behavior that only occurs in the first instance of a task would require analyzing several measurements in order to get a sufficient amount of data on execution times and probabilities.

5. Future work

When recording using intrusive probes such as we do here, the implementation is normally eligible to the probe effect [5]. However, we are in the situation that, provided our probes do not invoke irregular behavior, the model could very well be the same with probes and without. If our probes do indeed invoke irregular behavior, this should be possible to sense with a moderate testing effort. Thus, it should be possible to remove the probes from the system without penalty once the modeling is completed. We will investigate this in our future work.

Further, we plan to use model synthesis on a larger scale system than examined here. Ideally, we would attempt to generate models of industrial state-of-practice systems. This requires the capability to model more system calls than we do today.

6. Conclusions

The method presented in this paper allows for model synthesis of real-time systems. Motivation for the work has been presented and includes facilitating impact analysis and COTS-verification.

For the validation of the method, we have developed tools to examine recorded traces originating from both the run-time system and the synthesized model. The use of these tools have been shown here as the model synthesis was evaluated. During the successful validation, some weaknesses of the model synthesis have been experienced and examined. The causes of these weaknesses have been identified and are expected to be amendable.

References

[1] J. Andersson, A. Wall, and C. Norström. Decreasing maintenance costs by introducing formal analysis of real-time behavior in industrial settings. In *Proceedings of the* 1^{st} *International Symposium on Leveraging Applications of Formal Methods*, October 2004. Accepted for publication.

[2] S. J. S. Bastos and M. L. D. Sanchez. Modelling real-time systems from object oriented methods. In *Real-Time Embedded System Workshop*. IEEE, December 2001.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Proceedings of the* 4^{th} *DIMACS Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[4] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of uml sequence diagrams. In *Proceedings of*

the 10^{th} *Working Conference on Reverse Engineering*, pages 57–66, November 2003.

[5] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[6] O. Grinchtein, B. Jonsson, and M. Leucker. Inference of timed transition systems. In *Proceedings of the* 6^{th} *International Workshop on Verification of Infinite-State Systems*, September 2004. To apear in the Electronic Notes in Theoretical Computer Science series.

[7] J. Huselius. Preparing for replay. Licentiate Thesis, Mälardalen University, Sweden, November 2003. ISSN 1651-9256, ISBN 91-88834-15-8.

[8] P. K. Jensen. *Reliable Real-Time Applications. And How to Use Tests to Model and Understand*. PhD thesis, Aalborg University, 2001.

[9] A. Marburger and B. Westfechtel. Tools for understanding the behavior of telecommunication systems. In *Proceedings of the* 25^{th} *International Conference on Software Engineering*, pages 430–441, 2003.

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

[11] D. Nyström, A. Tesanovic, M. Nolin, C. Norström, and J. Hansson. COMET: A component-based real-time database for automotive systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEE, May 2004.

[12] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the International Conference on Software Maintenance*, pages 13–22, August 1999.

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

[14] T. Systä and K. Koskimies. Extracting state diagrams from legacy systems. In *Proceedings of ECOOP'97*, 1997. LNCS 1357.

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

[16] A. Wall, J. Andersson, and C. Norström. Probabilistic simulation-based analysis of complex real-time systems. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing*, 2003.

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