Experimental Model Synthesis for Timing Analysis of an Industrial Robot

Joel Huselius, Johan Andersson, Hans Hansson, and Sasikumar Punnekkat
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{joel.huselius,johan.x.andersson,hans.hansson,sasikumar.punnekkat}@mdh.se

Abstract

Manual modeling of real-time systems is time consuming and require skilled labor. As an alternative, for legacy systems without valid models, we advocate model synthesis (i.e. automated construction of models). Our previous work has presented a process for model synthesis that inputs logs from the original real-time system, and outputs a behavioral model. The current paper present an evaluation of this process that we performed on a state of practice industrial robot system. With this case study, we establish the feasibility of our model synthesis approach, and also show how to obtain a validated probabilistic models. These model can be used to analyze the temporal properties of real-time systems.

1. Introduction

System development is often not the structured process that we would like to think, where developers start out from models that are kept up-to-date as the system evolves. Many industrial real-time systems have been developed and maintained for many years and today contain millions of lines of code, with fragmentary and often out-dated documentation. Development relies heavily on the experience of skilled engineers and massive amounts of full-system testing.

Traditionally, when developing models of complex systems, in order to achieve the appropriate balance between abstraction and usefulness of the model, modeling is an art rather than an algorithmic process; the wit and cunningness of the model designer is imperative to the accuracy, efficiency, and usefulness of the model.

We can identify at least two problems with this kind of modeling process:

- Due to high learning thresholds for modeling paradigms, expensive time of skilled engineers is required for modeling.
- The continuous evolution of the system and the slow pace of human labor typically invalidates the model before or shortly after it has been completed.

In place of manual modeling, we propose that an automated process should be used to construct valid models for existing legacy systems. Our previous work [7, 8] has presented such a process of *model synthesis* for probabilistic models of real-time systems. In this paper, we evaluate the usefulness of that process by conducting an experiment on an industrial setting.

Model synthesis allow us to obtain probabilistic models of legacy code quickly and efficiently – the models can be used for analysis of the modeled system. The use of probabilistic arguments in the models allows us to obtain a more nuanced and abstract understanding of the system compared to traditional modeling with WCET – for example, execution times can be viewed as distributions instead of intervals, and abstract away from some of the data-state dependencies in the system [11].

1.1. Contribution

In this paper, we show that model synthesis based on logs can deliver models of legacy systems.

Specifically, by performing an experiment on an industrial robot case, we use our process of model synthesis described in [8] to: define the required extent of the logs as input, obtain the logs required, use one set of logs to produce the model, and another set of logs to validate the model.

1.2. Organization

The remainder of the paper is organized as follows: Section 2 describes our process of model synthesis and the tool set that we use to perform the experiment. Section 3 describes the experiment that we have performed. Section 4 presents the results and an analysis of the experiment. Section 5 concludes the paper.

2. Model synthesis

We argue that there is an algorithm that, based on already available information gathered from the implementa-

tion and its environment, is able to compile a valid model of the system without requiring additional inputs from humans. In other words: there is an algorithm that can perform model synthesis. We have produced an example of such an algorithm, and this paper is a proof of concept of using the process successfully in an industrial setting.

For input to the process of model synthesis, there are two potential sources available to us: we can use the source code of the implementation, or logs of the executing implementation.

We have chosen to use the later source of execution logs. This gives us the ability to abstract away from the complexity of the code, and provides us with real performance data. On the other hand, provided that no dynamic constructions such as dynamically linked libraries are used, using the code as input can ensure completeness of the model – if we choose to use only logs from recording as input, completeness of the model can only be guaranteed by performing exhaustive testing. We believe, however, that only using logs has advantages in complexity of the solution, portability, and also allows us to view the average timing behavior as well as the probabilistic behavior of the system.

It would of course be possible to combine the two inputs, thereby harvesting the advantages of both approaches, but that raises a new set of integration issues that we choose to leave for future work.

2.1. Execution recording

Recording is the act of saving information about the execution of a system; it consists of the three sub activities *probing*, *monitoring*, and *logging*.

By inserting *probes* into the system (i.e. *probing*), we can *monitor* the *events* that occur during execution. The output of monitoring can be *logged* to facilitate off-line analysis of the execution. The product of recording is a *log*. 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 reasons, we have used software probes in our implementation. The probes are tailored to extract relevant information from the system, this 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 [6] 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 *data 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.

2.2. The ART-ML modeling language

We have chosen to realize models in the modeling language ART-ML [11], which was defined as part of a previous research work in our center. The language allows probabilistic modeling of the behavior and the architecture of real-time systems. The probabilism is used as an alternative way to express selections, execution times, and to fire jobs of tasks in the system. Thus, when data state information is scarce, a probabilistic expression can replace a normal if-then-statement. Using the notion of probabilism, we can also express execution time as probabilistic distributions, which can be used to abstract the model from the variable state of the implementation – if the state determines how long a computation takes, we can optionally abstract away the state, and use an execution time distribution instead of modeling the exact behavior.

These means of abstraction are potentially valuable in system maintenance [11] (e.g. for understanding the complexity of the system and for understanding what effects future changes will have to the system). They can also be used to analyze real-time systems that are not constructed according to real-time theory.

2.3. Our process for model synthesis

Based on the two vital parts *model generation* [7] and *automated model validation* [8], we have constructed a process for model synthesis [8] as described by Figure 1.

The basic operation of the model generation is to, based on a given *log* from a recording, produce models from each executing task. In short, a *trace* for each individual task is filtered from the collected logs of the system, the trace is then folded into jobs of the task. Then, the smallest *tree* is constructed based on the trace, and that tree is then transformed into a model of the task. To provide model generation, the system must allow probing of system and task level events (e.g. context switches, ipc-operations, and selected variable assignments). We refer to [7] for details on the generation.

Automated model validation is performed on the above mentioned tree, and a collection of the above mentioned traces. These are transformed into communicating timed automata [2], one for the tree (see Figure 2), and one for each trace (see Figure 3). A third general auxiliary automata is used to pass parameters concerning variable identifiers, values etc. between the two automata (see Figure 4). Reachability analysis on the last label of each trace then proves the inclusion of the trace in the tree – which is evidence of a valid model. We refer to [8] for details of the model validation.

As outlined in Figure 1, to support model generation and model validation, we need to provide a number of function-

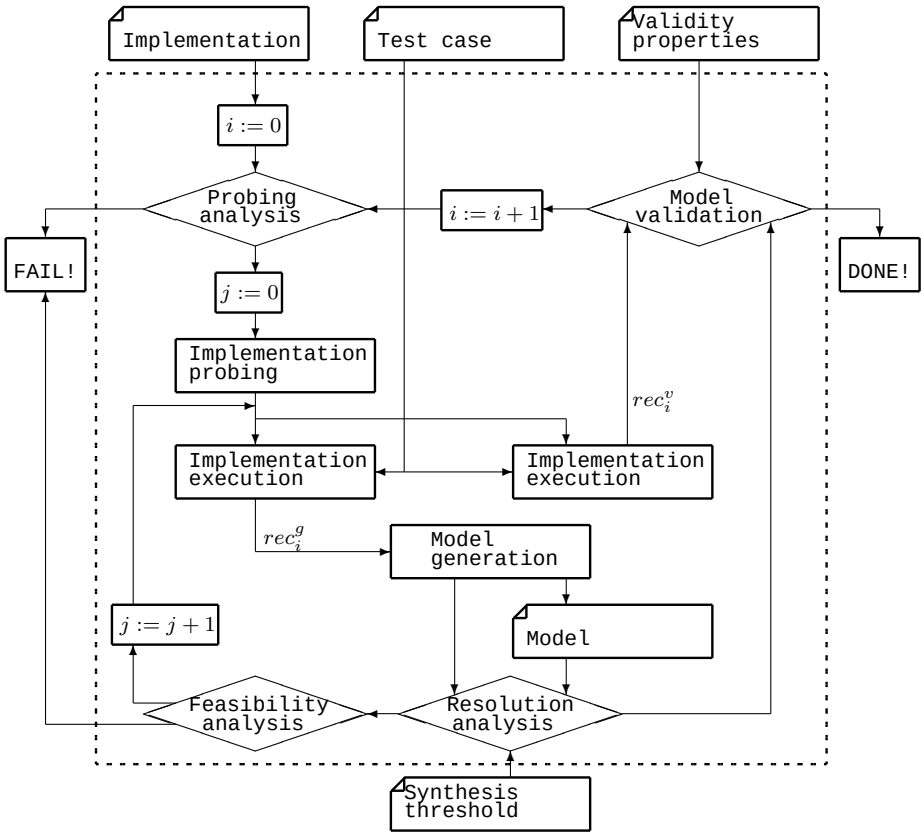

Figure 1. The process of model synthesis.

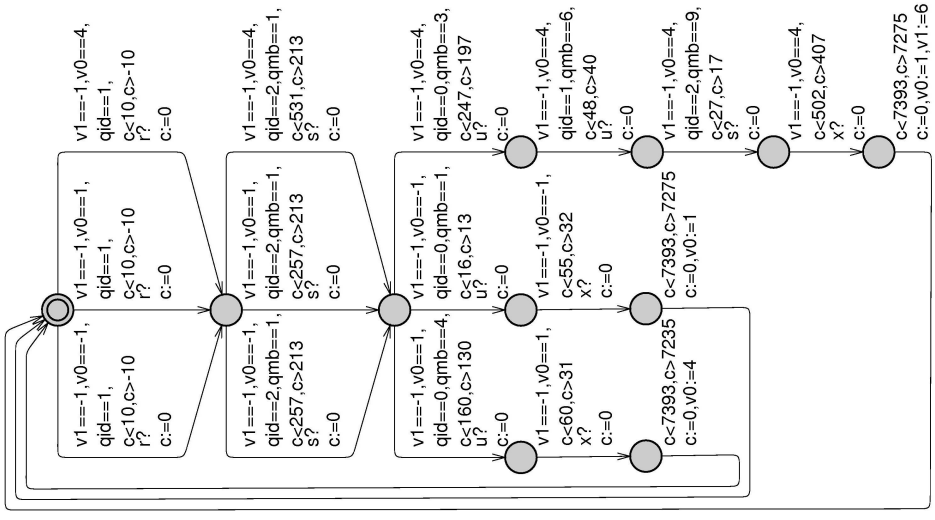

Figure 2. A timed automaton of a model, here without data-state.

alities:

Probing analysis suggests what data sources should be probed in the implementation.

Implementation probing implements the probes accord-

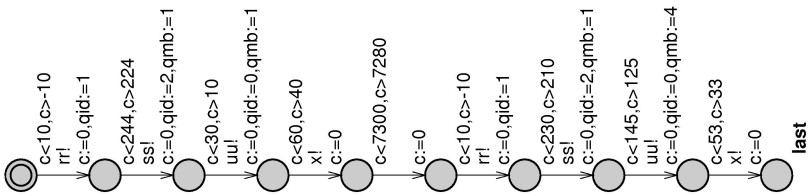

Figure 3. A timed automaton for a (short) trace that is accepted by the timed automaton of the model.

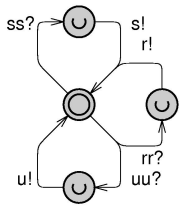

Figure 4. An auxiliary automaton for parameter passing between the two automata.

ing to the probing analysis.

Implementation execution produces logs by recording the executing system.

Resolution analysis analyses the coverage of the log to ensure that different behaviors observed are statistically sound – this is used to evaluate the length of the logs given the complexity of the implementation.

Feasibility analysis can abort the model synthesis if it seems unlikely to succeed (i.e. resolution analysis has failed too many times with the same probe setup).

In this context, it is the role of the probing analysis to suggest what variable assignments that need to be probed. The more accurate the analysis, the faster the model synthesis can terminate – if the analysis is poor, several iterations must be made before a passable suggestion is reached.

The process takes a set of tree inputs as follows:

Implementation is the subject of the model synthesis.

Test case is the context of the model synthesis – it is with respect to the test case used that the model must be viewed.

Validity properties is used to specify the leeway for automated model validity – the validity properties dictate how far from the reality the model is allowed to be. In fact, this is a measure of the allowed abstraction in the model.

As all the inputs are supplied, the first step of model synthesis, the probing analysis, can be initiated (see Figure 1). The first setup is to use only the required system level probes that cover system calls and context switches. This set up is then implemented in the following step (implementation probing), after which the first logs can be retrieved in implementation execution. Thereafter, Model generation will produce the first version of the model – this is then analyzed by the resolution analysis to ensure that the length of the executions are sufficient. In the Model validation, the relation between the model and the system is considered with respect to the validity properties that where supplied as input. If the quality of the model is inadequate, the probing analysis will find a suggestion on what data state to complement the probing with. There after the process will continue.

2.4. Tools used in the experiment

The tool for model generation has previously been described in [7]. Generally, it takes no more than seconds to consume a log that represent a minute of testing on the system.

The model validation use a set of recursive functions for automata production [8]. As the trace grows long, these will consume large quantities of memory – this experiment allowed up to 10 Mb of working space for the functions. In figures 2, 3, and 4, we show examples of these automatically constructed automata.

Our automated model validation require a timed automata model checker [8]. Unfortunately, the automata for

the set of traces generally have too many physical labels (> 5000) to allow the use of a standard general purpose model checker such as UPPAAL [4]. However, because the trace automaton will always be serial (there is a maximum of one transition to any label), the specific model checking problem is quite simple. Thus, we have been able to device a simple custom model checker to solve our specific problem. The execution time of our model checker is relative to the number of transitions in the tree automata multiplied by the number of transitions in the trace automata (for $40*5000$ transitions, the execution is less than ten seconds).

To analyze the properties of logs during the experiment, we made use of a set of tools previously described in [3]: The Tracealyzer provides a graphical view of the log with task interleavings and probed data state. The Property Evaluation Tool (PET) evaluates Probabilistic Property Language (PPL) queries on the logs. Queries can for example have the form "How large is the maximum response time of task i", or "How large percentage of the jobs of task i have an execution time lower than k?". These tools allow us to verify and debug our methods.

2.5. Implementation discrepancies

Our implementation of the process for model synthesis includes most of the functionality required. Probing analysis and implementation probing are, however, currently not implemented. This will not effect the final model produced, it may however be that manual implementation probing has another error frequency, and that manual probing analysis use a different pattern when configuring probing configurations. Though it will not affect the produced model, any of these issues may lead to that the experiment takes longer time to perform than necessary.

2.6. Related work

Previous work has presented a variety of methods for model generation from both code [5, 9] and recording [10, 12]. To our knowledge, ours is the only one that also provides an automated validation of the generated model – which is essential for successful model synthesis.

Moe and Carr [10] present an industrially sound method for model generation. However, the model produced in their method is not operational, but rather a visualization of occurred events. The method is reported to have helped discover and identify a number of bugs in an operation and maintenance system for cellular networks by Ericsson.

DiscoTect of Yan et al. [12] can construct architectural models from object oriented Java implementations. These models can show the possible interactions between tasks and resources such as files, semaphores, and abstract data objects. The approach is inherently non-real-time – monitoring of the real implementation is performed by using a debugger to query the implementation during run-time to extract information about occurred events.

3. The experiment

The work presented in this paper serves to show two things:

Firstly, that model synthesis is a feasible approach to obtain a functional model of a system. This is shown by construction – we execute the system with different probe settings, generate models from the obtained logs, and evaluate the models through automated model validation.

Secondly, validating the model with our automated method puts the focus on the validity parameter used in the study – a non-fitting validity parameter can potentially accept a model that has a dissimilar temporal behavior from that of the system. To examine the effect of the validity property, we performed the same validations with a set of different validity properties – the intention was to estimate the robustness of the validity process.

The goal of the experiment was to produce a validated model of one of the most complex tasks in the system. The absence of automated probing analysis and automated implementation probing made it infeasible for us to perform a larger study.

3.1. The industrial robot system

In our work, we study the ABB Robotics state of practice industrial robot IRC5. The object-oriented code base of the robot consists of > 2500 KLOC of C-code divided into $400 - 500$ classes and runs on the VxWorks 5.5 operating system on an industrial PC hardware with Intel Pentium 3 processors. We study the main computer (MC), which runs more than 60 tasks with preemptive fixed priority scheduling. Many of the tasks are event triggered, in which case they typically execute one of several services requested by the triggering task.

The system critical motion control subsystem, consisting of tasks *A*, *B*, and *C*, of the MC is responsible for generating motor references and brake signals to a DSP that in turn controls the physical robot. The DSP issues requests to the MC with a fixed rate > 200 Hz, it is critical that the MC can reply to each request within a given time.

Task *A*, with the lowest priority of the three, calculates the motion control commands on a high level of abstraction and submit results to task *B*. Task *B* communicates with *C* and *A*, to reduce the abstraction of the motion control commands from *A*. Task *C*, with the highest priority of the three, is responsible for maintaining the communication with the DSP.

In the experiment, we have focused on tasks *A* and *B*, whose implementation consists of > 250 KLOC C-code.

3.2. Experiment setup

We have used an experimental setup for the IRC5 where the physical moving parts of the robot are replaced with an emulator that communicates in real-time with the robot control system. This allows for a safe and realistic setup that closely resembles execution on the real platform. Using the experimental setup allows us to lower the safety and space requirements compared to a fully operational robot.

Due to the time factor of manually probing the system, as noted above, we decided only to perform model synthesis for a single task. For this task, we tried a series of validity parameters so that we could analyze the effect of the parameter.

The test cases evaluated where:

1. A simple, slow, moving pattern, where the robot is allowed to move in the simplest way. This pattern will result in low calculation intensity for the recorded task.
2. A complex, fast, moving pattern, where the robot moves short distances and halts its movement on designated coordinates. This pattern will result in high calculation intensity for the recorded task.

These will be referred to as *test case 1* and *test case 2*.

3.3. Making recordings for generation and validation

Probing the system manually proved to be an exhausting and error prone task – the model generation has specific rules concerning the probing it requires [7]. For example, each ipc-receive operation should log the time of making the call, the time of finishing the call, the queue number, and the timeout parameter of the call. Also, each variable assignment of each variable logged should be probed. We made good use of the Tracealyzer and PET tools to search the logs for errors.

After that a probing setup had been correctly implemented, we programmed the robot to make a series of maneuvers according to the test cases and started recording executions for generation and validation.

3.4. Using the recordings to generate and validate models

The model synthesis tool has a simple command-line interface which expects two files and the validity property as inputs. One file contains the subset of log to use for generating the model, and the other file with a (implicitly disjunct) subset of logs to use for the model validation.

We started model synthesis for each test case with a validity property of 0, and increased the property until all of the validation logs where accepted.

4. Experiment analysis

This section is dedicated to presenting and analyzing the data from the performed experiment presented above.

4.1. Experiment data

The results of the experiment are accounted in tables 1 and 2, where: *tc* is the test case, *log time* is the average time span of the logs, *log size* is the average size of the logs, *# variables* is the number of variables probed, *# g-logs* is the number of logs used in model generation, *g-time* is the time required for model generation, *# v-logs* is the number of logs used in model validation, *v-time* is the time required for model validation, *Vp-factor* is the validity property divided by the measured WCET for the task, *edges* is the number of edges in the validation automaton, *res.*, used in the resolution analysis, is the average of observations for unique events, and *Result* is the outcome of the synthesis. The result is given as a fraction of successes in model validation – if g-logs is 5, and model validation succeeded with 3 of these, the result is $3/5$.

Please note that, the function of the validity property (see Section 2.3) is such that increasing its value will always lead to a better or equal result. Thus, as the validity property is increased, the fraction of successful validations will increase or remain the same - but never decrease. If we have achieved full model validation ($5/5$ in the example above), we know the validity property required for the set of validation logs.

4.2. Data analysis

We can see from analyzing the complete models that the model in test case 1 was simpler than that of test case 2. This is reflected in Table 1, where we read that the number of edges used in the validation automaton for test case 2 of task B is twice of that needed for test case 1. Also, as follows from Table 2, the measured WCET of both tasks is higher in test case 2.

For both tasks, the ease with which the generated models passed through model validation differed with the test case that we examined. As test case 2 is more complex than test case 1, it seems likely to assume that the complexity of the test case has impact on the validity property required to pass the model. This fits with our understanding of the validity property as a filter to ease the stringency of the validation. Compared to the total measured WCET of the task however,

tc	log time [s]	log size [kB]	# variables [B]	# g-logs	g-time [s]	# v-logs	v-time [s]	edges	res.
1, task B	22	600	2	5	18	5	2	22	2537
2, task B	22	600	2	5	20	5	1	41	1317
1, task A	22	600	2	5	2	5	1	17	941
2, task A	22	600	2	5	2	5	1	17	758

Table 1. Test Case Descriptions

for task B, neither of the test cases required very high validity properties (see Table 2). For task A, the ratio is much higher, suggesting that more probing is needed for this task.

To perform the complete model synthesis, including probing, programming the test cases, and executing the system to produce the logs used, took no more than 1 day. To perform the same task again would most likely take considerably less time as the probes are already in place. With the actual model synthesis taking less than half a minute (see 1), it seems that model synthesis is considerably faster option than constructing the model manually – a task that would take considerably more than a week to perform. It seems that the time required for model synthesis is a sustainable delay in a development project – especially since the process can be fully automated and then possible to schedule to the night shift.

We measured the average resolution for the models that were synthesized – the resolution is the number of observations for a given event in the model, the measure aims to evaluate whether the length of the log is sufficient, given the complexity of the implementation. This is the resolution analysis of the model synthesis. In the case of task B, the average resolution of both test cases exceeded 1000. This translates to that if a previously unseen behavior would be discovered in subsequent traces, that behavior would have a less than 0.1% chance per observation to make an impact to the model. Our estimate is that this is sufficient guarantee for that the logs used are long enough. Task A had a slightly lower resolution, which suggests that longer traces should be used if possible.

However, we would like to mention that with this kind of model synthesis, the correlation between code and model is essentially lost – even though they could be reinstated, variable names etc. are lost in the process, and also, the structure of the model has no correlation to the implementation. These problems point to that, in order to meet the requirements of the industry, model synthesis should not be based on recording alone – some code or high-level description is needed to maintain the correlation with the software. Even so, the models that we produce here are well suited for simulation and the analysis possible there.

5. Conclusions

We have performed an experiment on an industrial system to test the capacity of our proposed method for model synthesis. A set of logs extracted from recording of two test cases were used as inputs to model synthesis, and a disjunct set of logs from the same test cases validated the result.

The performed experiment supports our hypothesis that the method is functional. More testing is required to establish the capacity of the method, but this initial study proved positive. Our approach can be used as a complementary technique to traditional timing and analysis, and also provide a viable mechanism to capture timing behaviours of interacting leagcy subsystems or bought-out components.

5.1. Future work

In our future work, we will work to complete the toolset with automated methods for probing analysis and implementation probing. This will allow us to synthesize models for all the tasks in the system – the models can then be co-simulated which will test the method even further.

In a later stage, we would like to device integrated methods for model synthesis based on both code and on logs. This dual input, it seems, is more suited for the complexity of industrial applications. While recording provide the real observed data needed for a realistic behavior model, as the code provides naming and other structural information, models based on dual input will be easier for humans to understand than models based solely on recording.

References

[1] Rapita systems ltd. www.rapitasystems.com.

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

tc	Vp-factor	Result
1, task B	0/ 978	4/5
1, task B	16/ 978	4/5
1, task B	17/ 978	5/5
2, task B	0/ 1048	0/5
2, task B	1/ 1048	2/5
2, task B	2/ 1048	3/5
2, task B	32/ 1048	3/5
2, task B	33/ 1048	4/5
2, task B	34/ 1048	4/5
2, task B	35/ 1048	5/5
1, task A	0/ 5996	0/5
1, task A	544/ 5996	0/5
1, task A	545/ 5996	1/5
1, task A	1082/ 5996	1/5
1, task A	1083/ 5996	2/5
1, task A	1170/ 5996	2/5
1, task A	1171/ 5996	3/5
1, task A	1294/ 5996	3/5
1, task A	1295/ 5996	4/5
1, task A	2441/ 5996	4/5
1, task A	2442/ 5996	5/5
2, task B	0/19871	0/5
2, task B	381/19871	0/5
2, task B	382/19871	1/5
2, task B	405/19871	1/5
2, task B	406/19871	2/5
2, task B	503/19871	2/5
2, task B	504/19871	3/5
2, task B	516/19871	3/5
2, task B	517/19871	4/5
2, task B	904/19871	4/5
2, task B	905/19871	5/5

Table 2. Synthesis Convergence based on Validity Property

[3] J. Andersson. Modelling the temporal behavior of complex embedded systems: A reverse engineering approach. Licentiate Thesis, Mälardalen University, Sweden, June 2005. ISSN 1651-9256, ISBN 91-88834-71-9.

[4] 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.

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

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

[7] J. Huselius and J. Andersson. Model synthesis for real-time systems. In *Proceedings of the* 9^{th} *European Conference on Software Maintenance and Reengineering*, pages 52–60, March 2005.

[8] J. Huselius, S. Punnekkat, and H. Hansson. Presenting: An automated process for model synthesis. MRTC Report 191, Mälardalen University, October 2005. Available at: www.idt.mdh.se/˜jhi/.

[9] G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level analysis of synchronous programs. In *Proceedings of the* 24^{th} *IEEE International Real-Time Systems Symposium*, pages 256–265. IEEE, December 2003.

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

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

[12] 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.