Automatic Generation and Validation of Models of Legacy Software

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

The modeling approach is not used to its full potential in maintenance of legacy systems. Often, models do not even exist. The main reasons being that the economic implications and practical hurdles in manually maintaining models of in-use legacy systems are considered too high by the industry. In this paper, we present a method for automated validation of models automatically generated from recordings of executing real-time embedded systems. This forms an essential constituent of a unified process for the automatic modeling of legacy software. We also present a study in which we automatically model a state-of-practice industrial robot control system, the results of which are clearly positive indicators of the viability of our approach.

1. Introduction

A large part of the industrial systems in use today are complex and software controlled, consisting of millions of lines of code. Many person-years have been invested in these systems, and a large group of people have contributed, some of whose services may be unavailable today (i.e. they were hired by third party, have quit their employment, or even passed away). These systems are labeled *legacy systems*.

Models are important for understanding software and for ensuring quality in the kind of large scale development process required to develop and maintain legacy systems. However, in practice, a common problem is that the models and the actual implementations diverge over time, ultimately making the model invalid and thus unusable. This in turn, can lead to reduced quality of the code-base.

To increase the effectiveness of large scale modeling, we must solve two fundamental problems:

1. How to construct models of large and complex legacy systems, with minimum interference to their continuous evolution/maintenance?
2. How to enforce consistency between the model and the modeled system?

Our thesis is that an automated process for modeling based on execution traces can produce models of systems without interfering with maintenance. Additionally, if such a process can be fast and cost-effective, then the consistency problem can be taken care of by repeated application of the process as often as is necessary. In this context, we have worked towards a process for automated modeling based on data collected through recordings of executing real-time systems. We have published a method for automated model generation [4]. However, in a process for automated modeling, the model generation must be complemented with a test of model validity, thereby making sure that the model faithfully captures relevant aspects of the modeled system.

In this paper, we extend our previous work on model generation with a method for automated validation. Using these two essential ingredients, viz., model generation and model validation, we present an iterative process for model construction. Iterations in the process are made when a generated model is invalidated with respect to the system it models. In this way, the risk of inconsistency is reduced. We have investigated the feasibility of this process on a state of practice industrial robot control system.

The outline of this paper is as follows: Section 2 provides background and basic definitions. Section 3 presents automated model validation. Section 4 presents a case study on applying model generation and model validation to a legacy system. Section 5 presents related work, and Section 6 concludes the paper.

2. A unified process

Using automated model generation and automated model validation, a unified process for automated model construction can be designed, as shown in Figure 1. The process uses information extraction, model generation, and model validation techniques in order to generate a valid model using as little information extraction (e.g. monitoring) as possible.

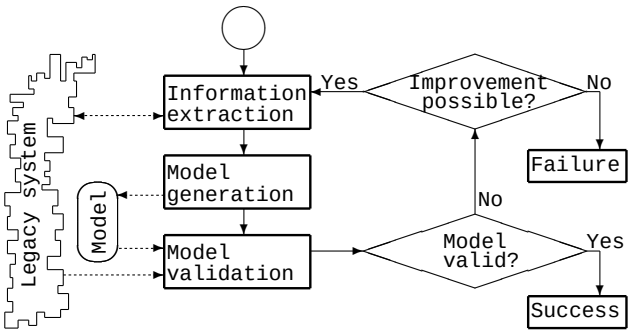

Figure 1. A process for automatic generation and validation of legacy software.

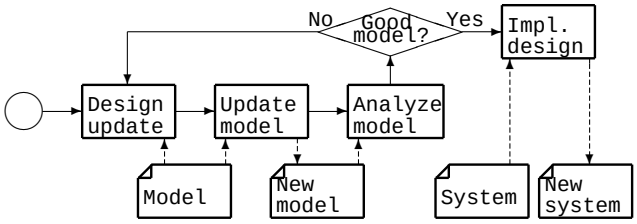

Figure 2. Model-based impact analysis.

One important motivation for our work is that system designers can use the automatically constructed models to prototype future design propositions [2] (see Figure 2). First (not shown in the figure) a valid model of the existing system is automatically generated. Second, a design proposition is constructed. Third, the generated model is manually modified to reflect the proposed modifications. Fourth, the properties of the modified model are analyzed with respect to timing and resource utilization. If the analysis show no evidence of problems with the design proposition (i.e. requirement violations), the design is implemented. Otherwise, a new design proposition is formed, and the process is restarted. We label this *model-based impact analysis*, the purpose of which is to avoid bad designs without actually implementing them. As it can reduce the time spent and wasted on bad designs, early identification and rejection of infeasible design alternatives has the potential of substantially reducing the cost of maintenance.

2.1. Recordings and data-structures

We assume that the legacy system consists of a set of *tasks* that can either be event or time triggered. As a task is triggered, a *job* is executed for some period of time, after which the task will await until further triggering.

Recording of selected *events* is fundamental to our approach for automatic modeling. We are capable of handling the following events: send and receive operations providing interprocess communications (ipc), variable updates, and context switches. The list can easily be extended to incorporate semaphore operations as well. On processing the recordings, selected *actions* are identified. With the current list of events, the types of actions are: ipc *send* and *receive* operations, *end* of a job, variable *updates* and *execute* statements, the latter of which model the consumption of CPU time. For each type of action, we assume that the following properties are associated:

receive: queue id, timeout, prm;

send: queue id, message, prm;

update: variable id, value, prm;

execute: time interval, next action, prm; and

end: time interval, prm;

where *id* denotes a unique identifier, *previously received message (prm)* denotes the value of a message received by a receive action that occurred immediately before the current action, and *next action* denotes the type of the action immediately following the current action.

We use two different data-structure representations for the actions observed in the recordings: The *recorded sequence (hereinafter referred to as recseq)* is a sequential list of elements $\langle id, g, a \rangle$, where a is the action guarded by the data-state g (a data-state being a unique assignment of values to the monitored variables). The *model tree (hereinafter referred to as modtree)*, which is a representation of the model of a task produced by model generation.

A modtree is a tree of modtree-elements, which are tuples $\langle id, G, a, S, c \rangle$, where: G, the guards, is the set of valid data-states for the modtree-element, $a \in Actions$ is the action of the modtree-element, S, the set of successors, is a list of possible modtree-elements for subsequent execution after the action a has been performed, and c is a counter that reflects the number of observations of the modtree-element, made in the recordings used to construct the modtree.

For recseq r and modtree m, we use a dot-notation (e.g. $r.a$ and $m.id$) to refer to their corresponding attributes.

2.2. Examples of recseq and modtree

As an illustration we will use a running example of modtree and recseq introduced in figures 3 and 4, respectively.

The intuition of the recorded sequence recseq is that it is a sequential list of jobs of a task; Figure 3 shows two subsequent jobs of Task T. Each event in the list corresponds to one and only one observation. Thus, the elements of the recseq are well defined in the sense that there is only one

valid data-state for each element, and if the type of the action is execute, there is only one time in the time interval. We have omitted the guard of the recseq since it is only used in model generation and the main focus of this example is model validation.

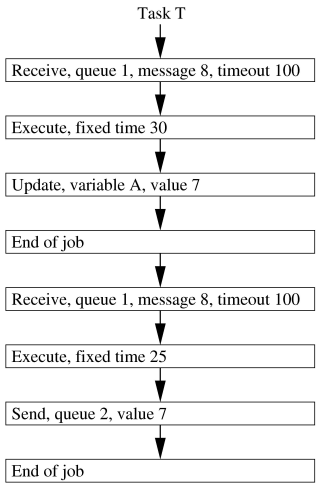

Figure 3. A recorded sequence (recseq).

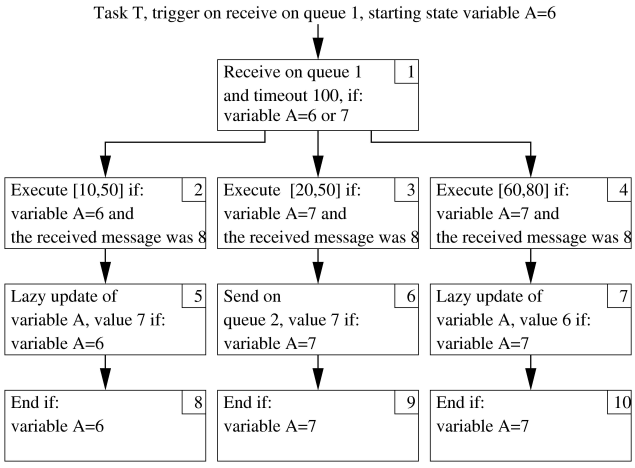

Figure 4. A model (modtree).

In the example of a modtree in Figure 4, we have omitted some information such as the $m.c$, but in the rightmost corner of each modtree-element, we show the corresponding unique identifier. In element 1, which we refer to as $m|m.id = 1$, a message is received on queue 1. Regardless of the value of a received message, a selection dependent on the value of variable A follows. In case the variable has the value 7, a stochastic selection is made between elements 3 and 4. The probabilities in the stochastic selection are calculated based on the $m.c$ property of the elements in the selection. Say that $m.c|m.id = 3$ has the value 360 and that $m.c|m.id = 4$ has the value 688, then the probability of selecting the path that starts with element 3 would be $360/(360 + 688)$.

The modtree implements a *lazy update* model, where updates take effect only after the job is completed [4]. This can be observed in Figure 4 in the result of updates in elements 5 and 7, and guards on elements 8 and 10. The concept of lazy updates is not important for this paper. It is used to maintain a compact, yet correct, modtree when guards of elements can accept several values for the same variable. Without lazy assignment, the model may allow false positives later in the job if paths are split based on the original value of the variable.

2.3. Automated model generation

Our method for model generation [4] requires as input a set of recordings covering at least system calls (e.g. send/receive actions) and context switches. Optionally, data-state in the form of variable assignments can be included for more detailed models. Each recording is first processed to separate the individual tasks of the system. For each recording, this results in one recorded sequence *recseq* for each task. According to a set of rules, the collected recseqs of each task are then composed into a *modtree*, which can be translated into the probabilistic modeling language ART-ML introduced in [11]. In this way, a separate model is produced for each task in the implementation. The collection of ART-ML models for all the tasks in the system can then be co-simulated and analyzed. Using our method for model generation, without loss of functionality, we are thus able to separate the concurrent tasks of the system into one modtree per task.

3. Automated model validation

As the model generation phase may have to be iterated in order to find an appropriate probe setting resulting in an acceptable model, efficient automated modeling requires automated model validation. This section shows how to perform validation by testing the generated model of each task (in the form of a modtree) against a new set of recorded traces (a set of recseqs) obtained by recording the execution of the modeled system. The test serves to determine whether if more, longer, or more detailed traces are needed for model generation to produce models of the required quality. To this end we need techniques to compare the modtree with the set of recseqs. The necessary techniques can be found in automata theory; model checking would allow us to compare the two, since it is possible to formulate the inclusion checking as a reachability problem. As we are concerned

with real-time systems, timed automata theory is a natural choice.

We follow the definitions of timed automata given in [3], which extends [1] by adding integer variables. Using integer variables allows for expressing causality between jobs (for example modeling mode changes) and for communicating action-properties between the automata.

The two automata are constructed so that the automaton for the model is a tree-like structure while the automaton for each recording is sequential. A proof that the recorded traces are contained in the model is constructed by using model-checking to verify the reachability of the final state of each trace automaton when composed with the model automaton. The objective is to show that the action-properties in the recseqs does not contradict the modtree. The modtree is discarded if any of the tests fails. Otherwise, we conclude that the model is valid. The size of the set of recseqs determines the *validity measure*: the level of confidence that can be placed in the final model.

The following must be solved to achieve automated model validation:

A. Obtaining the automata. We need a translation from the modeling language to timed automata and from traces to timed automata.

B. Stopping criteria. It must be possible to determine when sufficient validity has been established.

C. Allowing leeway. Since the model is meant only to be an abstraction of the system that is modeled, the model should be allowed to differ slightly from the traces. The validation must be able to allow a user-defined leeway for the model with respect to the system. A variable leeway will provide the user with the ability to decide the granularity of the solution taking into account the constraints on cost and effort as well as the precision necessary for the intended use of the model.

Since the solutions to C influences the solution to A above, we present our solutions in the order they appear in the solution.

3.1. Allowing leeway

A useful model should be an abstraction from the system that it models. Since we are primarily interested in modeling real-time systems, this requires the model to be a timely abstraction of the system. As an abstraction, the model cannot be a perfect reflection of the system. Rather, it should provide a similar behavior while being significantly less complicated than the modeled system.

We choose to realize the abstraction, or leeway, by relaxing the timing requirements in the timed automata for recseq. Each time observation t in the recseq is replaced by the interval $(t-pp, t+pp)$, where pp is a user supplied *precision parameter*, thus providing the ability of passing validation even though the model does not exactly correspond to the modeled system.

3.2. Obtaining the automata

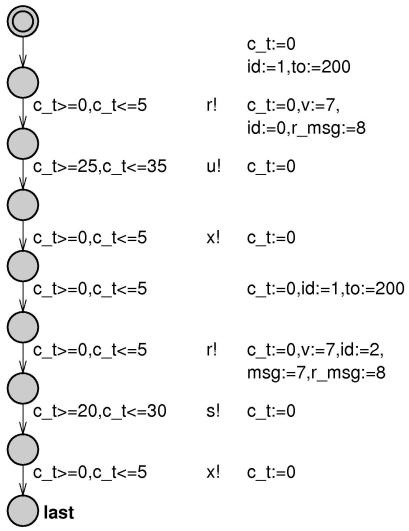

Figure 5. A timed automata representation of the recseq, using a precision parameter of 5.

As our model is represented by a modtree and the traces by recseqs, translations from modtree to timed automata and from recseq to timed automata are required. As per the definition of timed automata [1, 3], we need to obtain the following to perform a translation to timed automata: the set of locations in the automaton, its initial location, and its set of edges. Each edge is given by: the guards (originating both from variables, clocks, and action properties), the action, and the clock resets and the variable updates.

Using a set of recursive functions (published in a technical report [5]), we can define the translation of the modtree and the recseq into the corresponding automata.

For our example in figures 3 and 4, this results in a set of automata as described by figures 5 and 6, respectively. The set of locations and the initial location are derived from the unique identifiers of the modtree and the recseq, respectively. For each valid data-state in an element of a modtree or recseq there is an edge in the corresponding automata. In the modtree-automata, the handling of several states in one element is exemplified by the two edges with receive-action from the initial location L0, which has two outgoing edges to the subsequent location L1.

The execute- and end-actions need to be given special

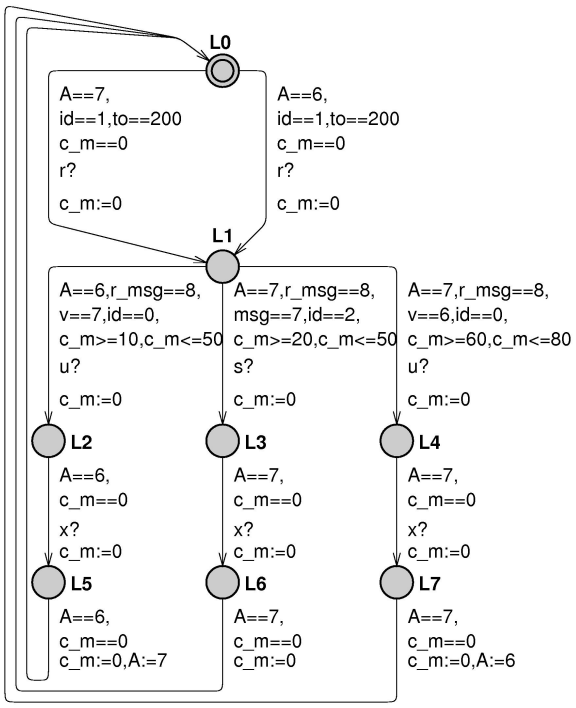

Figure 6. A timed automata representation of the modtree.

treatment. The execute-actions are modeled as clock guards on edges of other events in the automata, as can be seen in the two update edges (see transitions L1-to-L2 and L1-to-L4) and the send edge (transition L1-to-L3) of the modtree-automata. Intuitively, the automata will remain in the location with the guarded edges (e.g. L1) until execution time has been consumed. If the modtree-automata and the recseq-automata have contradicting views on the required execution time, the composed automata will result in a deadlock, which signifies that the model is invalid. In the recseq-automata, we note the precision parameter used to allow leeway in the validation.

The end-actions are modeled as two edges, one with the action *end* (e.g. L2-to-L5), and one sequential ε-action (e.g. L5-to-L0). Following the concept of lazy update [4], in the ε-action, the set of updates-actions performed during the job are actuated. For periodical tasks, the blocking time preceding the next consecutive job of the task is modeled as a clock guard on the ε-action.

To ensure correspondence between the action-properties in both automata, the preceding edge of the recseq will perform a set of updates that are checked by the modtree-automata. This is apparent in the first edge of the recseq-automata, where an ε-action is used to initiate the global variables `id` and `to`. The variables are subsequently used in the guard of edges in the modtree-automata from L0-to-L1.

3.3. Stopping criteria

We use two validity measures on the jobs observed to determine whether we have performed enough testing to make us confident that the model is an adequate abstraction of the system. We also require that the validation fulfills certain user defined criteria based on these measures:

- *Completeness measure*, i.e. the probability that the model can replicate any job that the system can exhibit.
- *Accuracy measure*, i.e. the probability that the system exhibits a particular job must be sufficiently close to the probability that the model exhibits an equivalent job.

To this end, we need a new notion for estimating equivalence between observed jobs. Using such a notion, we will be able to group the jobs into classes and estimate the above measures based on the properties of the classes.

Definition 1 (Syntactic equivalence). Two jobs of a task are syntactically equivalent iff their action sequences, variable updates, and outputs coincide. □

Intuitively, syntactic equivalence can be tested using a symmetric version of the timed automata test described above, except from the demands on timing. In our example, due to the functionality of the model generation process, the number of syntactic equivalence classes (SECs) in a modtree is the same as the number of end-actions in the modtree.

The completeness measure is assessed by analyzing the frequency with which new SECs are discovered in the recording of the system. We start out with the set of SECs identified in the recordings used to generate the model (see Figure 1), and we assume that the probability of discovering a job of an unknown SEC follows a binomial distribution.

There is a plentitude of ways to determine how many recseqs needs to be tested to satisfy a given completeness requirement. Essentially, we want to know the number of tests (i.e. recseqs) needed to attain a certain confidence in the modtree. For example, if we state the null hypothesis h_0 that there are more SECs than have discovered so far, with probability $\pi_0 \leq 0.01$ of falsely rejecting h_0, with significance $\alpha = 0.01$ and margin of error $\varepsilon_0 = 0.01$, we can estimate that the required sample size is lower than 1000 according to Jeffrey's $100(1-\alpha)\%$ confidence limits [7].

The models produced are probabilistic – that is, the model can use probability to determine selections at behavioral level – which requires that the model has valid estimate

of the probabilities of different SECs. The *accuracy measure* should determine if these estimates are valid, i.e. within allowed tolerances.

Let G denote the set of jobs used to generate the model, and V the set of jobs used for validation. Internally, we group the jobs into SECs and analyze the probability distributions of equivalence classes in the two sets G and V. For the analysis, we let the function $sec\_cnt : 2^{job} \times SEC \rightarrow \mathbb{Z}^*$ denote the number of jobs of a SEC in a set of jobs. The function $h : SEC \times 2^{job} \times 2^{job} \rightarrow \mathbb{Z}^*$ compares the probabilities of a SEC n in G and V.

$$h(n, G, V) = \left| \frac{sec\_cnt(G, n)}{|G|} - \frac{sec\_cnt(V, n)}{|V|} \right| \quad (1)$$

The lower $h(n, G, V)$ is, the more accurate the model is. We specify an accuracy threshold h_a that is the maximum allowed difference between probability of equivalence classes in the two traces. The objective is to ensure that, for all identified equivalence classes n, h is below this threshold, i.e.:

$$\forall n : \ 0 \leq h(n, G, V) \leq h_a \quad (2)$$

In our example, three different SECs are identified: sec_0, sec_1, and sec_2. Based on the $m.c$ information, we can calculate the probability of each SEC, for example $P(sec_0) = 30\%$, $P(sec_1) = 50\%$, and $P(sec_2) = 20\%$. The accuracy threshold is set to $h_a = 1\%$.

After model checking recseqs with a total of c jobs, without finding any error in the model, the validation measures are examined: Suppose that the accuracy measures found where $P_c(sec_0) = 20\%$, $P_c(sec_1) = 55\%$, and $P_c(sec_2) = 25\%$ – which is not within h_a for this first measurement, thus implying that the number of traces is inadequate.

Suppose that, after a total of $2c$ jobs have been examined, the accuracy measure is $P_{2c}(sec_0) = 33\%$, $P_{2c}(sec_1) = 46\%$, and $P_{2c}(sec_2) = 21\%$ – which is within h_a. Then, $2c$ is an acceptable sample size.

3.4. Automated model validation procedure

As we have provided solutions to the problems above, our procedure for automated model validation of a given modtree is as follows:

Produce a sufficient number of recseqs to accommodate the stopping criteria.

Produce the automata for both the model and the traces as described in Section 3.2.

Check for trace inclusion of each recseq-automaton in the modtree-automaton. Assuming that the last location of each recseq-automaton is named `last`, this is done by model checking each modtree-automaton-recseq-automaton composition with the Computation Tree Logic-formula `E<> recseq.last`, expressing that there is an execution in which the state `last` is reachable.

According to our simplified method of automatic modeling in Figure 1, we then should check if there is any additional information that we can extract from the system. If so, this will allow a more detailed model with a better chance to pass a subsequent model validation test.

4. Case study

We have performed an industrial case study in which we applied automated modeling to a subset of the motion control system of a commercial industrial robot.

The >2500 KLOC object-oriented C-code base of the robot runs under VxWorks 5.5 on Intel Pentium 3 industrial PC hardware. We study the main computer (MC), running almost 100 tasks with preemptive fixed priority scheduling. Many of the tasks are event triggered, typically executing one of several services requested by the triggering task.

Motion control is a critical subsystem of the MC, 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 ($f > 200$ Hz). It is critical that the MC replies to each request within a given time. The motion control subsystem consists of tasks *A*, *B*, and *C*. Task *A*, with low priority, calculates the motion control commands on a high level of abstraction and submits results to task *B*. Task *B*, with medium priority, communicates with *C* and *A*, to reduce the abstraction of the motion control commands from *A*. Task *C*, with high priority, is responsible for maintaining the communication with the DSP.

In the experiment, we have focused on tasks *A*, *B* and *C*, whose implementation consists of more than 250 KLOC. The test case that we analyzed was a complex, fast, moving pattern, where the robot moves short distances and halts its movement at designated coordinates. This pattern will result in high calculation intensity for the recorded tasks.

Probes where introduced to record context switches, explicit delays, selected variable updates, ipc-send, and ipc-receive events. Each probe took about $0.8\mu s$ to execute, which is quite negligible w.r.t the task execution times, which are often measured in milliseconds.

4.1. Results

We successfully generated models of the three tasks based on five recordings. However, during validation using five other recordings, it was discovered that the two recorded variables in task *A* required a leeway in excess of 10 percent of the task's Measured Worst Case Execution Time (MWCET) to pass all tests (see Figure 7). After we

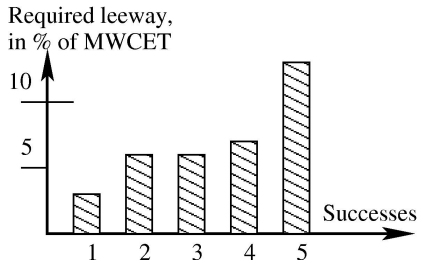

Figure 7. Leeway requirements for the first probe setting when modeling task A.

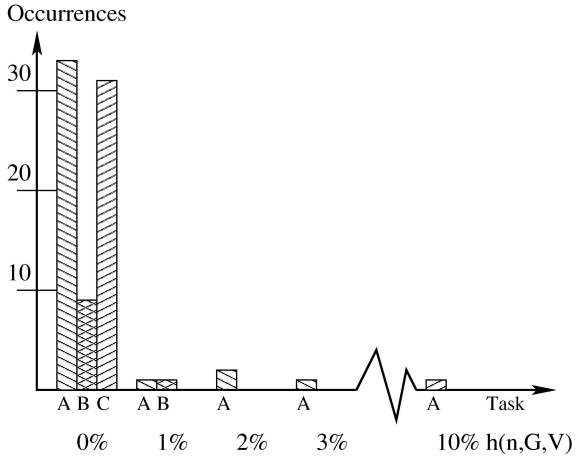

Figure 8. $h(n, G, V)$ **distribution of all tasks.**

had iterated the information extraction two times, we found a set of variables that when probed gave better data-state information. This allowed us to pass the test with less than 10 percent leeway. In the final information extraction setting, we recorded two variables in task *A*, one in task *B*, and one variable in task *C*.

As we analyzed the stopping criteria for the final set of recordings we found that, even though the completeness measure was very low (our sample size for validation was only five recordings), the accuracy measure was $h_a \geq 10\%$, as can be seen in Figure 8. This is probably due to the length of the recordings: each recording was approximately 22 seconds long and spanned thousands of jobs.

When validated models had been obtained for all three tasks, we merged the models and co-simulated them together with a hand made environment model that partly emulated the remaining tasks on the MC. Thus allowing validation of the collected model from a system perspective. We performed a large series of simulations, using several different starting conditions. During these, we could observe a strong similarity in the behavior of the model and the legacy

	Execution time percentiles	Task A	Task B	Task C
Measured	Minimum	3	12	2
	25:th percentile	13	807	13
	50:th percentile	110	851	29
	75:th percentile	292	867	708
	Maximum	21281	2821	1117
Simulated	Minimum	4	20	6
	25:th percentile	13	735	29
	50:th percentile	121	852	115
	75:th percentile	356	926	833
	Maximum	23017	2723	1164

Table 1. Execution time distributions from system execution and model simulation.

system. However, as the accumulated model does not represent the entire system, we cannot perform realistic measures of response times. We can however analyze the execution time distributions for the three tasks.

Comparing the execution times measured on the system with that of simulations (see Table 1), it is evident that there are deviations from the expected results: Due to limitations in the handmade environment simulation, the event triggered task *C* did not receive input as intended. This insufficiency in the environment model caused many short jobs never to be triggered in the simulation. This is seen in the 25^{th} and 50^{th} percentile of task *C*. With better environment modeling or with larger scope in the case study, this error can be remedied.

The final model, including the environment model, occupied 4.0 KLOC without any optimizations, which is less than 1.6% of the size of the implementation.

5. Related work

There are a number of methods available for model generation. Sifakis et al. [9] proposed a static method that use tagging of the real-time software with time constraints to facilitate automated modeling based on the code as input. Yan et al. [12] presents DiscoTect, a tool that can obtain high-level architecture models from traces of non-real-time system implemented in Java. Moe and Carr [6] present automated modeling from recordings of RPC-calls in CORBA-based legacy systems. The product of the method is not an operational model that can be simulated, but rather a visualization of occurred events. The tool has reportedly helped discover and identify a number of bugs in an operation and maintenance system for cellular networks by Ericsson. However, we are not aware of any methods of model generation that includes a method for automated model validation.

Szemethy and Karsai [10] present a method for translating their handmade SMOLES models of component based real-time systems into timed automata as a step in model-based development. Shu et al. [8] provide an automated translation from their extended version of UML to timed automata. Contrary to our work, their work does not consider models with data state, and queries performed on the model has to be tailor made for the specific system.

6. Conclusions

We have presented a unified process for automatic modeling of legacy real-time software. Based on recordings of the legacy system, it is possible to automatically produce and validate probabilistic models, suitable for simulation based analysis of dynamic system properties, such as timing and resource usage. The applicability of the process has been shown in an industrial case study on a state of practice system. Although the impact of recording was very low, the quality of the model was high and the model size was small.

The inputs utilized for automated modeling construction characterizes the properties of the finalized model. Using only recordings from the system, the direct coupling back to the code of the implementation is weakened. As a consequence, the model may look radically different than the implementation, but the behavior will still have a strong correspondence. In this perspective, it may be more attractive to use the code or the documentation as input – but these inputs are associated with other drawbacks such as portability, completeness, and complexity. We are investigating this in a concurrent research effort.

Future work will address automated monitoring setup, which is the only part of our proposed process that is not yet automated.

References

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

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

[3] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Automated analysis of an audio control protocol using uppaal. *Journal of Logic and Algebraic Programming*, 52-53:163–181, July-August 2002.

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

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

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

[7] W. Piegorsch. Sample sizes for improved binomial confidence intervals. *Computational Statistics & Data Analysis*, 46(2):309–316, June 2004.

[8] G. Shu, C. Li, Q. Wang, and M. Li. Validating objected-oriented prototype of real-time systems with timed automata. In *Proceedings of the* 13^{th} *IEEE International Workshop on Rapid System Prototyping*, pages 99–106, July 2002.

[9] 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.

[10] T. Szemethy and G. Karsai. Platform modeling and model transformations for analysis. *Journal of Universal Computer Science*, 10(10):1383–1407, October 2004.

[11] A. Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. Phd thesis no. 5, Mälardalen University, September 2003. Available at: www.mrtc.mdh.se.

[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.