Evaluating the Quality of Models Extracted from Embedded Real-Time Software

Joel Huselius, Johan Kraft[*], Hans Hansson, and Sasikumar Punnekkat
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{joel.huselius,johan.kraft,hans.hansson,sasikumar.punnekkat}@mdh.se

Abstract

Due to the high cost of modeling, model-based techniques are yet to make their impact in the embedded systems industry, which still persist on maintaining code-oriented legacy systems. Re-engineering existing code-oriented systems to fit model-based development is a risky endeavor due to the cost and efforts required to maintain correspondence between the code and model. We aim to reduce the cost of modeling and model maintenance by automating the process, thus facilitating model-based techniques. We have previously proposed the use of automatic model extraction from recordings of existing embedded real-time systems. To estimate the quality of the extracted models of timing behavior, we need a framework for objective evaluation. In this paper, we present such a framework to empirically test and compare extracted models, and hence obtain an implicit evaluation of methods for automatic model extraction. We present a set of synthetic benchmarks to be used as test cases for emulating timing behaviors of diverse systems with varying architectural styles, and extract automatic models out of them. We discuss the difficulties in comparing response time distributions, and present an intuitive and novel approach along with associated algorithms for performing such a comparison. Using our empirical framework, and the comparison algorithms, one could objectively determine the correspondence between the model and the system being modeled.

1. Introduction

Model-based techniques such as implementation prototyping and prototype performance analysis [16] are still not widely used by industrial system developers. According to our industrial contacts, the reluctance against using model-based techniques is largely due to the initial cost of modeling of a code-oriented legacy system where models are not used today. Our research focus is on reducing this cost, thus making methods that use models more attractive in industrial settings.

Our application domain is that of real-time systems, i.e. those systems where temporal and functional correctness are equally important. The most common type of requirement for real-time systems is a bound on the *response time* for given stimuli. In a complex multitasking system, determining the bound and distribution of response times is generally difficult in practice.

In the context of this paper, a typical legacy system has all or some of the following properties: it consists of millions of lines of code, it is maintained by a large team of engineers, it contains code that originated several years ago, and it is expected to be further developed for many more years to come. Real examples of these systems can easily be found within many domains such as the automation, automotive, and telecom industries. In such systems, a large effort must be spent on keeping complexity at acceptable levels [11]. If the complexity is allowed to increase without bound, the life expectancy of these systems will be drastically reduced. Though model-based analysis can help in limiting the complexity increase, there is a reluctance of the industry to adopt these technologies. Hence, tools that ease modeling and the maintenance of models are needed. We have developed tools for *model extraction* in order to ease both modeling and model maintenance [2, 6, 7]. The models that our methods extract reflects the general behavior of the system rather than the worst-case, which is a more common focus in real-time systems research.

1.1. Contributions

In this paper, we present a framework for evaluating our proposed tools with respect to real-time properties such as the ability to accurately model response time distributions. The proposed method can also be used to compare the effectiveness of different methods of automatic model extraction for the general system behavior proposed in the literature (e.g. [8]). In addition to the framework, a intuitive and novel method for comparing time distributions is introduced. We supply the method as well as a set of algorithms to perform

[*]Formerly Johan Andersson.

the comparison. In the framework, the comparison method plays a critical role as it provides a measurement that can be used to evaluate the performance of a model with respect to the system.

1.2. Organization

The remainder of this paper is organized as follows: Section 2 provides background on previous work and our problem domain. Section 3 presents framework for empirical testing and comparison of proposed methods for automatic model extraction. Section 4 provides a definition of an objective measurement to compare distributions of response times, and a set of algorithms to perform measurements. Section 5 concludes the paper.

2. Background

In our work, we are developing tools for automatic or semi-automatic modeling of legacy real-time software. The models are intended to be used in model-based implementation prototyping and prototype performance analysis. As a part of this effort, we have developed a unified method for dynamic model extraction [6, 7]. The basic idea is to input execution recordings of the legacy system that is to be modeled, covering context switches, inter-process communication (IPC), and updates of important variables. If the recordings contain enough information, a tool implementing the unified method automatically delivers a validated model, otherwise the user is advised to alter monitoring or extend recordings. The unified method consists of two separate methods; one for *automatic model generation* and one for *automatic model validation*. Model extraction is performed separately for each task, and a collection of models for all tasks in the system can be merged and used to analyze the system by means of simulation. A set of models produced by the method can be used to prototype future design options with respect to response time requirements and functional behavior. A case study has been performed on a state-of-practice industrial robot system to show the applicability of the method [7].

In a parallel effort, we are also developing tools for semi-automatic static model extraction based on implementation code and execution recordings [2]. One of our future aims is to compare the performance of these two different strategies.

2.1 Automatic model generation

Introduced in [6], our automatic model generation can, based on a set of recordings of a running system, output a model of the system. The recordings cover system level events such as context switches and communication. Optionally data state manipulation on task level is included, allowing modeling of causal relations. Generation is performed in three stages: First, an event sequence for each task is extracted from each recording. Second, all event sequences for each task are merged into a tree structure. Third, each tree is translated into a model for the task.

To express abstraction from the implementation, the models contain probabilistic elements: Selections can be made based on probabilities (or on data state), and execution time requirements can be described as probability distributions. For the effective use of the modeling language, there is a tool-suite for simulating and analyzing the performance of models.

A limitation of this method is that it does not model loops within tasks as in [3]. However, our assessment is that such loops are often avoided in embedded real-time systems due to predictability requirements. Also, this limitation does not effect the contributions of the paper: the evaluation framework and the comparison of sampled distributions.

2.2 Automatic model validation

We validate that the recordings used to generate the model are sufficient to describe the system by answering the question “Would the model be drastically better if the length or number of recordings used during model generation were increased?”. Automatic model validation, introduced in [7], uses a set of system execution recordings to answer that question. The model and the recordings are transformed into a set of communicating timed automata with integer variables [1, 4]. While the model-automaton is a graph structure which may contain more than one transition from each label, the recording-automata are all sequential with one or zero transitions from each label. The validation is performed by reachability analysis of the final state in each recording-automaton when co-simulated with the model-automaton.

To allow the model to be an approximate abstraction of the system, the recording-automata are constructed using a *leeway*-parameter. The higher the leeway, the more forgiving the recording-automaton will be. The maximum allowed leeway can be supplied by the user as a parameter.

The stopping criteria of the validation is based on two factors: The *completeness measure*, i.e. the probability that the model can replicate any job that the system can exhibit, and the *accuracy measure*, i.e. the relation between the probability that the system exhibits a particular job and the probability that the model exhibits an equivalent job.

Validation can provide the maximum required leeway, the completeness measure, and the accuracy measure as auxiliary output. The primary output is the binary answer to the question posted at the top of this section.

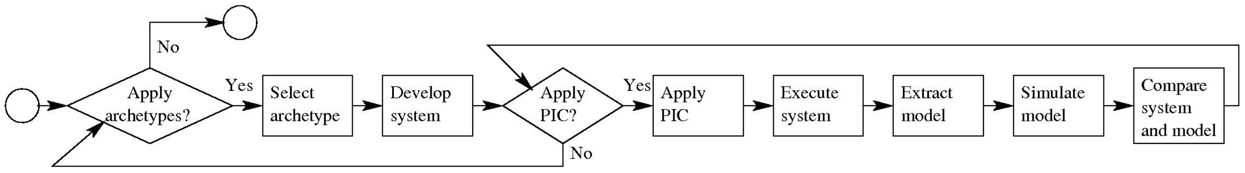

Figure 1. A process-view of the framework for empirical evaluation.

Together, model generation and model validation forms model extraction.

2.3 Response time

In real-time systems, the temporal and behavioral requirements are equally important. We assume that the embedded real-time systems we consider consist of a set of *tasks* that can either be event or time triggered. As a task is triggered, a *job* (a task invocation) is executed for some period of time, after which the task will await until further triggering. Two jobs of the same task cannot overlap. We label the time measured from the point in time where the job is triggered (the *release time*) until the end of a job the *response time* of the job.

3. Framework for empirical evaluation

In this section, we introduce the framework for empirical evaluation depicted in Figure 1. The purpose of the framework is to evaluate the effectiveness and general applicability of dynamic model extraction. We use a notion of archetypes to describe architecturally different system designs. For example, a system consisting of a set of periodic tasks without inter-process communication (IPC) is a different archetype than a system consisting of event triggered tasks where the exchange of messages trigger execution.

We have implemented an instance of the framework, where the system platform is a multitasking, fixed priority scheduled, instruction-set simulator that we have developed specifically for this purpose. A system of tasks is defined by a system definition file together with an assembler-file per task. Each task is either defined as triggered periodically or triggered as a result of input on an IPC-queue. The simulator can handle 16 types of instructions, including absolute and relative sleep, branching, IPC, explicit logging, register manipulation and testing, and random number generation. Each instruction takes one clock cycle to complete. As the system is executed, occurred task switches, performed IPC, and executed explicit log instructions are automatically recorded in a system specific log.

To each system definition, we can add an increasing portion of *Population*, *Imperfectness*, or *Complexity* (PIC) for each test performed on the system:

- *Imperfectness* regards the quality of the data-state recording in the system. With low imperfection, all relevant information is recorded. With high imperfection, some data state information is omitted, which leads to that the execution time distributions for a given data state are non-trivial.
- *Complexity* regards the task complexity, e.g. the number of tasks in the system, the nature of environment stimuli, and the number of data states.
- Finally, *Population* regards system wide issues such as the number of tasks in the system as well as the recording lengths and recording set sizes.

For each system definition and PIC combination, we perform two sets of simulations of the system: recordings from the first set is used to generate the model and the second set is used to validate the model (as described in Section 2). The intention is that as PIC changes the system, the model should follow and be affected correspondingly. Varying the PIC will test the robustness of the model extraction with respect to changes in the system – this form of robustness is imperative for successful implementation prototyping [13].

In our implementation of the framework, we compare the system and model by analyzing response time distributions from both system and model. After model extraction, the model is simulated and compared to new simulations from the system. The collected set of comparisons is the output of the framework implementation. These can then be analyzed to verify that the method of model extraction performs sufficiently well, or even to compare several methods of model extraction. As the set of archetype-PIC combinations is intended to be large, the comparison should be automated. We present a novel automated measurement in Section 4 that can be used in such a comparison.

3.1. Archetypes and PIC

The following are examples of archetypes that are used in our study:

1. **Client-server without reply.** This archetype describes a common design pattern in the industrial systems that

we have encountered. A client sends varying service requests to a server that services the requests. Results of the computations may effect the environment or successive requests to a third or fourth task. PIC applied are priority ordering, frequency increase, and execution time increase.

Specifically, this archetype is implemented with two tasks T_1 and T_2. With a fixed periodicity, Task T_1 sends a message to Task T_2 that reacts on the contents of the message. We distinguish between four different contents, representing four different commands, plus one default behavior in the case that the content in unrecognized.

2. **State machine.** Here, a task acts as a state machine which makes one transition per job. Transitions are triggered by messages from the environment or from another task. Task mode changes can be expressed by this archetype. In contrast to the client-server archetype, the same message can trigger different behavior at different points in time. PIC applied are reduced recording of variable assignments (simulating poor probing), environment stimuli, complexity of the state machine, priority ordering, frequency increase, and execution time increase.

   The archetype is implemented by two tasks T_1 and T_2. With a fixed periodicity, Task T_1 sends a message to Task T_2 with randomly selected contents 0 or 1. The event triggered Task T_2 consists of a finite state machine that can make one state transition per job. The target state of each transition is depending on the contents of the triggering message from T_1. A variable is maintained to keep track of the current state.

3. **Purely periodic without communication.** A task set of periodic tasks where execution times for any given task varies randomly between jobs within determined intervals. In this case, the PIC consists of increase of the task set size.

   In our experiments, the implementation consists of at most seven periodic tasks T_{1-7}, that execute a bounded random interval in each job. For each task, the worst case execution time (WCET), the period (T), utilization (U), and analytical worst case response time (R) are described in Table 1.

4. **Feedback loop.** Here, tasks exchange messages in a loop. Examples include client-server with reply, or a feedback control system. The PIC consists of priority ordering, message complexity, message reply frequency, and environment stimuli.

   The implementation consists of five tasks, two of which (T_1 and T_2) are implementing the feedback loop, and the three remaining are concurrently executing a client-server without reply and a simple periodic task.

	WCET	T	R	U
T_1	10	80	10	12.5%
T_2	30	120	40	25.0%
T_3	20	160	60	12.5%
T_4	15	180	75	8.3%
T_5	30	200	115	15.0%
T_6	40	300	300	13.3%
T_7	80	1000	960	8.0%
Σ				94.6%

Table 1. Maximum utilization for Archetype 3.

5. **State machine feedback loop** This archetype is a combination of archetypes 2 and 4, as is the PIC.

   The implementation consists of two tasks T_1, and T_2. Both tasks are state machines, Task T_1 generates input to trigger Task T_2, Task T_2 generates input that, if available, will affect the execution of Task T_1.

4. Comparison of sampled time distributions

The framework proposed above assumes that it is possible to objectively compare a system with a model of that system. We have chosen to implement this by comparing distributions of e.g. response times from the system and the model. However, methods known to us from literature prove unintuitive in this setting:

The Euclidean distance metric and the χ^2 test of independence [5] are both *categorical* in the temporal dimension, which results in that they are not sensible to the difference that two samples have almost the same response time if they are in different categories. They are only sensible in the sample dimension, which means that they can acknowledge that *almost* the same number of samples in both distributions have response times in the same category. This leads to unintuitive results due to false negatives.

The Kolmogorov-Smirnov test [12] assumes that one of the distributions in a comparison is mathematically modeled [10, 14]. However, the execution time distribution of a program is often very complex. On the source code level in a system implementation, selections where one path has a significantly longer execution time than the other are common. Execution time distributions that cover both legs of such selections does not follow a simple pattern. Therefore, it cannot be assumed that a response time can be classified to a known distribution (e.g. a normal distribution). We know of no universal method of determining or estimating similarity between unclassified finite discrete distributions.

To amend this lack of a suitable method of comparison, we introduce a novel objective measurement for sampled distributions based on the two notions of *divergence* (see Definition 1) and *difference* (see Definition 2).

Definition 1 (divergence)**.** Let U be the set of samples. There is a function $time : U \rightarrow \mathbb{Z}^*$. For a given sample $u \in U$, we use $time(u)$ to denote the value of that sample.

Let $A, B \in 2^U$ be two sets of samples from two sources (e.g. a model and a system) with equal cardinality. We define a *match* between these two as a *bijective* mapping between A and B, $\delta_{AB} : A \rightarrow B$. Let C_{AB} be the full set of matches (i.e. the full set of bijective mappings) between A and B.

We are now able to define a measurement of a match $\delta_{AB} \in C_{AB}$ by the function $[\,] : C_{AB} \rightarrow \mathbb{Z}^*$ as:

$$[\delta_{AB}] = \max_{a \in A} |time(\delta_{AB}(a)) - time(a)|$$

Then, the *divergence* between two sets of samples is the measurement of the most favorable match in the sense that the measurement is minimized:

$$\text{divergence}(A, B) = \min_{\delta_{AB} \in C_{AB}} \{[\delta_{AB}]\}$$

□

Intuitively, for two equally sized sampled distributions A and B, δ_{AB} describes a mapping from each sample in A to a unique sample in B (the uniqueness follows from that the mapping is bijective). *Divergence* is then considering the best possible mapping in the sense that the largest difference of response times that the matched samples represent should be as small as possible. In the following sections, we will present algorithms for measuring the divergence between two distributions.

Definition 2 (difference)**.** Given Definition 1, the *difference* between two sets of samples with equal cardinality is defined by:

$$\text{difference}(A, B) = |\{a \in A \mid time(\delta_{AB}(a)) \neq time(a)\}|$$

□

Intuitively, for a mapping of samples, *difference* counts the number of mapped samples whose response times are not equal.

Then, with divergence C_{min} and difference D, the comparison between distributions A and B is defined by the tuple $cmp(A, B) = \langle \frac{C_{min}}{C_{max}}, \frac{D}{lcm(|A|,|B|)} \rangle$, where C_{max} is the difference between the smallest and the largest samples from both distributions.

The relation between divergence and difference is used to quantify the correspondence between two sampled distributions as described in Figure 2. If the relations between divergence and difference is in an area with same shade of gray, two distributions are considered similar. The two are more similar the darker the shade is. As explained by the figure, if both divergence and difference are small, the distributions are very similar, if the divergence is small but the difference is large, the distributions are similar, etc. In the figure, we have exemplified four comparisons between imagined distributions. Distribution A is compared to distributions B_1, B_2, B_3, and B_4 respectively. The relation between comparisons *cmp*(A, B_1) and *cmp*(A, B_4) tells us that B_1 is more similar to A than B_4 is, since the former lies in a darker area than the latter. Analogous, *cmp*(A, B_2) and *cmp*(A, B_3) tells us that B_2 and B_3 are equally similar to distribution A, since they lie in an area with same shade.

The plot of Figure 2 is conceptual, the relations between divergence and difference must be defined for a given use of the comparison. If two methods of model extraction are compared in the framework above, the divergence-difference relation is defined once and used throughout the comparison. The intention is that the divergence-difference relation should reflect the quality that is required of the models. This is defined by the intended use of the models and the application domain of the system.

In the case of performing this measurement on response times, the observability problem [15] must be respected: Take the example of a system with a set of strictly periodic tasks. Here, especially if the system load is high, it is likely that jobs of tasks are ready to execute long before they receive their first quanta of processing time. According to our definition of response time (see Section 2.3), the time of the

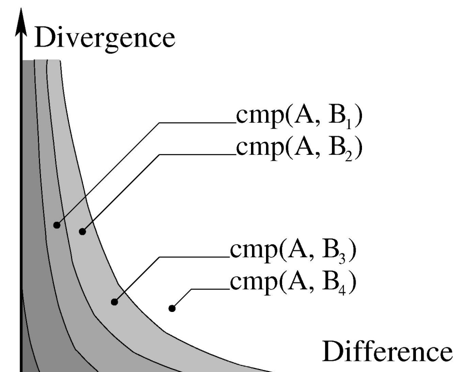

Figure 2. Conceptual divergence and difference plot.

triggering of the task must be known. Thus, probes that can access the ready queue of the operating system must be used to obtain a truthful measure of the response time.

4.1. Divergence and difference algorithms

We have implemented algorithms for calculating the divergence and difference as defined above. The following terminology will be used:

A distribution, or series of measured response times, *d* is represented as a binary tree where each node has the attributes *value* and *count*. The values of the attributes of node *n* of *d* are referred to with a doted notation (e.g. *n.value*). So are the siblings of n (e.g. *n.left*). The distribution has two operations *add* and *remove*, these are called using a similar doted notation (e.g. *d.add(e)*, where *e* is a response time). An *element* is a response time measurement represented in a series of response times.

A stack *s* has operations *push*, *pop*, *top* and the attribute *size*, which are called with a doted notation (e.g. *s.pop()*). The attribute is transparently manipulated by the operations *push* and *pop* to reflect the number of elements on the stack. The function *top* is used to investigate the topmost element on the stack without removing the element. The stack will be used to store elements on the form $\langle e_1 \in \mathbb{Z}^*, e_2 \in \mathbb{Z}^*, alternative \in \{true, false\}\rangle$, where the first element e_1 of the tuple is a response time from the first distribution, the second element e_2 is a response time from the second distribution, and the third element *alternative* denotes whether there are other feasible matches for the pair formed by the first two elements. To refer to the elements in the topmost entry of the stack, we use a doted notation (e.g. *s.top().alternative*).

To perform the comparison, the sizes of the two series of response times are first normalized by linearly adjusting the *count* of *value*'s such that the total sum of *count*'s in each of the two trees is equal to the Least Common Multiple (LCM) of the original total sum of *count*'s of both trees.

Then, the divergence C_{min} and the difference D for the comparison $cmp(d_1, d_2)$ are defined as follows: C_{min} is the smallest C such that $IsDivergence(d_1, d_2, C)$, as defined in Algorithm 1, evaluates to true. D is the size of the difference between the two distributions as defined by Algorithm 4.

4.2. Algorithms for measuring the divergence of two distributions

Algorithm 1 matches elements in two distributions so that the values of each match are less than or equal to a specified relative divergence. If all elements in the two distributions are successfully matched, the divergence between the two distributions is less than or equal to the specified relative divergence. In this process, Algorithm 1 uses Algorithm 2 to find the lowest value with a count larger than zero in a specified distribution. In turn, Algorithm 2 uses Algorithm 3 to decide which of two values is the smallest value in a given distribution.

Intuitively, if both values are in the specified distribution, Algorithm 3 returns the smallest of these. Otherwise, if only one value is in the specified distribution, that value is returned and the return value is deemed *valid*. If none of the values are in the distribution, *invalid* is returned.

Algorithm 2 compares the value of the current node, if the count of that value is larger than zero, to the valid results of recursive calls for the left and the right siblings of the binary tree. If the count of the value is less than or equal to zero, only the valid results of the recursive calls are considered. Out of all valid values, Algorithm 3 is used to find the smallest value within the specified distribution. This smallest value is returned and the return value is deemed *valid*.

Algorithm 1 $IsDivergence(d_1, d_2, C)$ compares two response time distributions d_1 and d_2 represented as binary trees with respect to divergency $C \in (0, C_{max})$.

```
Require: $srb.size = 0 \wedge d_1.size = d_2.size$
 1: $cMin := C$
 2: repeat
 3:   $e_1 := FindLow(d_1, 0, \infty, 0)$
 4:   $e_2 := FindLow(d_2, e_1, C, cMin)$
 5:   if $e_2$ is valid then
 6:     $d_1.remove(e_1)$
 7:     $d_2.remove(e_2)$
 8:     $e_2^{alt} := FindLow(d_2, e_2 + 1, C + (e_1 - e_2), 0)$
 9:     if $e_2^{alt}$ is valid $\wedge$ $(e_1 \neq srb.top().e_1 \vee e_2 \neq srb.top().e_2)$ then
10:       $srb.push(\langle e_1, e_2, true\rangle)$
11:     else
12:       $srb.push(\langle e_1, e_2, false\rangle)$
13:     end if
14:     $cMin := C$
15:   else if $(srb.size > 0)$ then
16:     repeat
17:       $d_1.add(srb.top.e_1)$
18:       $d_2.add(srb.top.e_2)$
19:       $popped := srb.pop()$
20:     until $(srb.size \leq 0 \vee popped.alternative = true)$
21:     if $popped.alternative = true$ then
22:       $cMin := popped.e_1 - popped.e_2 - 1$
23:     end if
24:   else
25:     return false
26:   end if
27: until $d_1.size \leq 0$
28: return true
```

Algorithm 2 $FindLow(d, value, cMax, cMin)$ finds the lowest value in the binary tree *d*, with attributes *value* and *count*, such that *d.value* is in the interval $(value - cMin, value + cMax)$ and *d.count* is larger than zero.

```
Require: $cMax \geq cMin$
1:  $min :=$ invalid
2:  if $d$ is valid then
3:      if $d.count > 0$ then
4:          $min :=$
5:              $MinInInt(d.value, min, value, cMax, cMin)$
6:      end if
7:      $a := FindLow(d.right, value, cMax, cMin)$
8:      if $a$ is valid then
9:          $min := MinInInt(a, min, value, cMax, cMin)$
10:     end if
11:     $b := FindLow(d.left, value, cMax, cMin)$
12:     if $b$ is valid then
13:         $min := MinInInt(b, min, value, cMax, cMin)$
14:     end if
15: end if
16: return $min$
```

Algorithm 3 $MinInInt(a, b, value, cMax, cMin)$ returns the lowest of a and b which is in the interval $(value - cMin, value + cMax)$, if any. Otherwise, the return value is invalid.

```
1:  if $a \in (value - cMin, value + cMax)$ then
2:      if $b \in (value - cMin, value + cMax)$ then
3:          if $a < b$ then
4:              return $a$
5:          else
6:              return $b$
7:          end if
8:      else
9:          return $a$
10:     end if
11: else
12:     if $b \in (value - cMin, value + cMax)$ then
13:         return $b$
14:     else
15:         return invalid
16:     end if
17: end if
```

If the algorithm is unable to find a value in the distribution, *invalid* is returned.

For a given divergence, for each count of each value in the first distribution, Algorithm 1 attempts to find a match in the second distribution. A match results in that the count of the matched value in both distributions is decreased, and the match is pushed on a stack together with information (*alternative*) of whether if there are other potential candidates for the particular match. If the search for a match fails, the stack is popped and the distributions are consequently repopulated until an old match with potential for another match is popped. Then the process is resumed. If, during that rollback process, the stack becomes empty, the match is said to have failed.

Pushed on the stack are a tuple, where one element (*alternative*) describes if the popping of the element on the stack will lead to that new untried combinations of matches can be performed. The value of *alternative* is decided by two criteria tested on Line 9 of Algorithm 1: First, the result of the call to Algorithm 2 on Line 8 of Algorithm 1, where the second of the two distributions are searched to determine if other combinations exists for the chosen element of the first distribution. Then, by checking the top of the stack to determine if a similar match as the one about to be made has been made just recently. The second criteria is important in terms of computational complexity; with that criteria, many unnecessary tests are avoided, but it is strictly speaking not important for the operation of the algorithm.

In the test, if the divergence *C* is large enough, all elements in the first distribution can be matched with any element in the second distribution. From understanding Algorithm 1, we see that such a situation must lead to that $IsDivergence$ returns *true*. The smallest value of this largest *C*, denoted C_{max}, when $IsDivergence$ must return *true* is equal to the difference between the largest observed response time and the smallest observed response time.

Plotting the result of $IsDivergence$ with two given distributions as a function of *C*, we get a monotonous function which starts at *false* and subsequently, at some value of *C*, turns *true*. As the function is monotonous for two given distributions we can use binary search [9] in the interval $C \in (0, C_{max})$ to find the smallest divergence C_{min} that satisfies Algorithm 1. Seen in relation to the size of the interval $(0, C_{max})$, C_{min} provides an objective measure on the likeness of the two distributions.

The use of a stack in the realization of this algorithm is essential to the implementation due to the potentially large memory requirements. As the distributions in the measurement are normalized by size, the total sum of the counts of values is potentially large, which will inflict on the number of matches that need to be identified. Storing this large number of matches will require large amounts of memory, but the use of a stack provides us with the opportunity to reduce the amount of physical memory used without large penalty to the execution time: One of the properties of a stack is that, at any given time, only the topmost element is needed. Thus, as the order of element usage is, if not determined, then at least restricted, a large part of the stack can be written to secondary storage (i.e. to file). We have implemented this by defining a threshold for the maximum number of elements that are allowed in primary memory, when this number is reached, the stack is flushed to file.

The theoretical worst case complexity of the *IsDivergence* algorithm is $O(N!)$, where N is the size of the distribution sizes when normalized by size. As N is likely to be large, the worst-case complexity is very high. However, due to the *alternative* field in elements on the stack, it is unlikely that the worst-case ever occurs. Typically, with N of approximately $1,000,000$, the algorithm takes in the order of minutes to execute.

4.3. Algorithms for measuring the difference of two distributions

The difference of two series of response times is computed as follows: Algorithm 5 finds, if any, the node in the binary tree representation of a distribution that has a given value. Algorithm 4 use Algorithm 5 to calculate the size of the difference between two distributions, which are normalized by size. Intuitively, the algorithm counts all elements of the first distribution that have no corresponding element in the second distribution.

The theoretical complexity of this algorithm is $O(PQ)$, where P is the number of unique sample values in the first distribution, and Q is the number of unique sample values in the second distribution.

Algorithm 4 $Difference(d_1, d_2)$ calculates the difference between the distributions d_1 and d_2.

```
1:  $s := 0$
2:  if $d_1$ is valid then
3:      $s := Difference(d_1.right, d_2)$
4:      $s := s + Difference(d_1.left, d_2)$
5:      $d := Find(d_2, d_1.value)$
6:      if $d$ is valid then
7:          if $d_1.count - d.count > 0$ then
8:              $s := s + d_1.count - d.count$
9:          end if
10:     else
11:         $s := d_1.count$
12:     end if
13: end if
14: return $s$
```

Algorithm 5 $Find(d, value)$ implements binary search [9] to find the node in the distribution d that matches the value *value*.

```
1:  if $d$ is valid then
2:      if $d.value = value$ then
3:          return $d$
4:      else
5:          if $d.value > value$ then
6:              return $Find(d.left, value)$
7:          else
8:              return $Find(d.right, value)$
9:          end if
10:     end if
11: end if
12: return invalid
```

4.4. Example

To exemplify, for a given task, assume that a series $\langle 1,1,1,1,1,2,5,6 \rangle$ of response times has been observed in the system, and that the series $\langle 1,2,3,5 \rangle$ has been observed in the model. The LCM of the number of elements in the series (8 and 4) is 8. Thus, normalization of the size of the series gives that the series observed at the system remain unchanged, and the series of response times observed at the model are $\langle 1,1,2,2,3,3,5,5 \rangle$. In this example, we will not use binary search, but to illustrate the algorithm pick values of divergence that gives interesting executions of the algorithm.

After normalization, the series of system response times $\langle 1,1,1,1,1,2,5,6 \rangle$ are represented as the binary tree d_1, and the series of model response times $\langle 1,1,2,2,3,3,5,5 \rangle$ are represented as the binary tree d_2.

In our search for the measure of divergence, performed in the interval $(0,5)$, we start with divergence value $C = 0$. In Algorithm 1, Line 3 will set $e_1 = 1$. At Line 4, e_1 is assigned 1. At Line 9, the first criteria is evaluated to false because there is no other match when the divergence is zero. The second criteria will also evaluate to false because the stack is empty. Thus, the tuple $\langle 1, 1, false \rangle$ is pushed on the stack.

When the match progressed such that two matches have been found, distribution d_1 has two occurrences of the value 1 remaining, while distribution d_2 has none, the search will rollback to find a new path. However, none of the matches found has had any alternative matches, hence the test with divergence $C = 1$ will fail.

If $C = 1$, the first two matches will match value 1 from d_1 with 1 from d_2. Then, a third and fourth match will find value 1 from d_1 and 2 from d_2, leaving the distributions as follows: $d_1 \equiv \langle 1,2,5,6 \rangle$ and $d_1 \equiv \langle 3,3,5,5 \rangle$. The next search for a match to $e_1 = 1$ will fail, and a rollback will commence.

It is then discovered that the third match, performed between 1 and 1, could also have been performed between 1 from d_1 and 2 from d_2. This satisfies the first criteria of Line 9. The second criteria is also satisfied as the previous match performed was between 1 and 1. Thus, the rollback stops, and the next found match is the aforementioned alternative match between values 1 from d_1 and 2 from d_2.

Subsequently however, also this divergence will prove to conclude a failure. It is not until the divergence is higher than or equal to three ($C_{min} = 3$) that the algorithm finds matches for all samples in the distribution. In relation to the span of the values in the distributions, which is 5, this is a relatively large divergence.

The difference of the two distributions is computed using Algorithm 4 in the form $Difference(d_1, d_2)$. The algorithm concludes that the difference is four ($D = 4$). In relation to the size of the distributions when normalized by size, which is 8, this is a relatively large difference.

The result of the comparison $cmp(d_1, d_2)$ is $\langle \frac{3}{5}, \frac{4}{8} \rangle$. Interpreting these measurements in the light of Figure 2, we conclude that the distance between the distributions is not negligible. For example, if the second series of observations would have been $\langle 1, 1, 2, 5 \rangle$, the conclusion would have been different.

5. Conclusion

We have presented a framework for evaluating the feasibility of using the automatic model extraction approach to obtain models of real-time software. We have presented examples of generic and commonly used archetypes and described their role in the framework with respect to PIC. The method has the same scalability issues as has any other application of testing: it may take time to construct and implement archetypes and PIC as well as to perform an evaluation. However, archetypes and PIC are generic and can be reused by other evaluations, and a method needs only to be evaluated once (evaluations of different methods can then be inter-related). Once a library of archetypes and PIC has been constructed, evaluation is essentially an automated task. Also, as the evaluation needs only to be performed once per method, our opinion is that the complexity of performing the evaluation is acceptable.

Further, we also discuss the fundamental issues of how to compare models against systems and we provide an objective measurement that solves this problem within the context of comparing response times of models and systems.

In our future work, we plan to explore and identify more relevant archetypes as well as perform evaluation of our method for dynamic model extraction. We are also developing more efficient algorithms for measuring the divergence, and are investigating other potential comparison methods.

5.1. Acknowledgments

We would like to acknowledge the fruitful discussions we had with Stefan Bygde in formulating the definition of divergence and Dr. Gustav Öquist in formulating the evaluation framework. We also thank the anonymous reviewers for their valuable comments.

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, J. Huselius, C. Norström, and A. Wall. Extracting simulation models from complex industrial real-time systems. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA)*, October 2006.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

[4] 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.

[5] D. Freedman, R. Pisani, and R. Purves. *Statistics*. W. W. Norton & Company, 3rd edition, 1998.

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

[7] J. Huselius, J. Andersson, H. Hansson, and S. Punnekkat. Automatic generation and validation of models of legacy software. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 342–349, August 2006.

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

[9] D. E. Knuth. *The Art of Computer Programming*, volume 3. Sorting and searching. Addison-Wesley, 1973.

[10] D. E. Knuth. *The Art of Computer Programming*, volume 2. Seminumerical algorithms. Addison-Wesley, 2nd edition, 1981.

[11] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, October 1996.

[12] F. J. Massey, Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, March 1951.

[13] A. K. Mok. Tracking real-time systems requirements. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 333–336, December 2000. Invited Address.

[14] NIST/SEMATECH. e-handbook of statistical methods. Internet URL http://www.itl.nist.gov/div898/handbook/, November 2006.

[15] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

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