Presenting: An Automated Process for Model Synthesis

Joel Huselius, Hans Hansson, and Sasikumar Punnekkat
Mälardalen Real-Time Research Centre,
Mälardalen University, Västerås, Sweden
joel.huselius@mdh.se

October 14, 2005

Abstract

This report presents an automated process for constructing models of legacy real-time systems based on recordings.

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. Even when modeling is used, due to time and resource restrictions, maybe even lack of knowledge, modeling and other documentation of software becomes fragmentary and obsolete over time. In industry today, many legacy real-time systems lack (up-to-date) models that can provide an abstraction of the often complex inner workings of these systems.

Traditionally, 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. Here, we present such a process of *model synthesis* for probabilistic models.

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 probabilism in the models allows us to obtain a more nuanced and abstract understanding of the system compared to traditional modeling with WCET – execution times can be viewed as distributions instead of intervals, behavior can be stimulated without a clear understanding of what data-state that is required [13].

2 System model

The system is an event triggered, fixed priority scheduled, real-time system of *tasks* that each execute one *job* at a time. Communication is available through IPC system calls. The jobs of a task can either be triggered with an approximate periodicity (time triggered task), or as the result the reception of a new message on a known IPC queue (event triggered task). Interaction with the environment is implemented using system calls.

We assume that system scheduling algorithm, blocking policy (e.g. direct inheritance), triggering for each individual task, task priorities, and task identifiers are known. Also, we assume that an interface to execute code at both context-switches and system calls is provided by the operating system. (Such functionality is available in commercial operating systems such as VxWorks.)

In the remainder of the report, we let $\mathbb{Z}^+$ denote the set of positive integers $\{1, 2, 3 \ldots\}$, $\mathbb{Z}^-$ denote the set of negative integers $\{-1, -2, -3\}$, $\mathbb{Z}^* \equiv \mathbb{Z}^+ \cup \{0\}$, and $\mathbb{Z} \equiv \mathbb{Z}^- \cup \{0\} \cup \mathbb{Z}^+$.

We define the data structure *list* according to Definition 1.

Definition 1. The data structure $list$ is an ordered series of values in $\mathbb{Z}$: $[v_0, v_1, v_2 \ldots v_n]$ such that the index of values must be respected. We let $|l|$ denote the length of list l, the function $\uparrow$: $list \rightarrow \mathbb{Z}^*$ denote the maximum value in the list, and $\downarrow$: $list \rightarrow \mathbb{Z}^*$ denote the minimum value in the list. □

3 The process of model synthesis

Figure 1, shows our process of model synthesis described as a set of activities. As indicated in the figure, there are four inputs to the process (these are detailed with the respective step that uses them):

- The system *implementation* that is the subject of the model synthesis.
- The *test case* that defines the context in which the system will be modeled.
- The *validity properties* that define the quality that the model should have in relation to the implementation.
- The *synthesis threshold* that defines resolution that is required for probabilistic choices in the model.

The output of the process is the *model*, which currently is described in the ART-ML modeling language [13].

The set of activities in the process are described in the following *steps*:

- The *probing analysis* checks if there are any untried probing configurations, and evaluates the previous configuration (if any).
- The *implementation probing* is responsible for preparing the implementation with the appropriate probes so that required information can be obtained.
- The two instances of *implementation execution* produces two recordings: one is used to produce the model, the other is used to validate the model.
- The *model generation* produces a synthetic model based on the recording from the previous step.

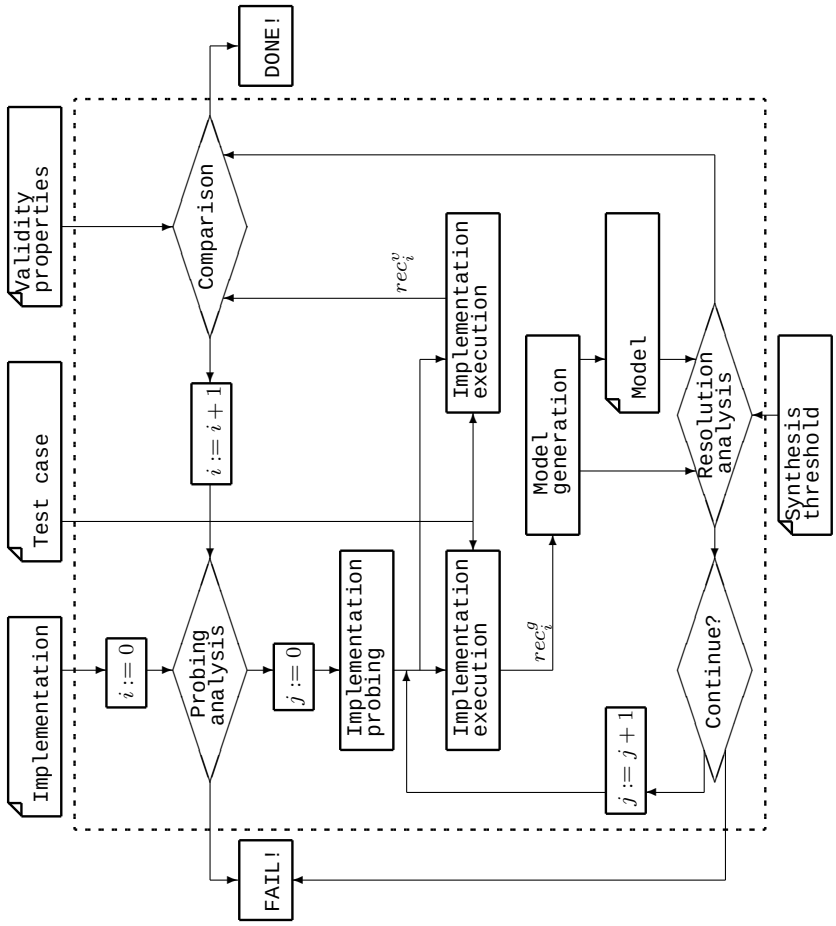

Figure 1: The process of model synthesis.

- The *resolution analysis* examines auxiliary output from the model generation to evaluate if the recording length used is sufficient.
- The *continue*-step decides if the search for a usable recording length is likely to terminate or not.
- The *comparison* evaluates the level of similarity between the model and a recording from the implementation.

These steps are explained in the sub-sections that follow:

3.1 Probing analysis

The *probe setup* is the current configuration of probes in the implementation. There is a minimal set of system-level probes that is required for model generation, and an optional set of task-level probes that can be added in order to improve the produced model. The process of determining the members of the set of optional task-level probes is a crucial part of the model synthesis process.

There are two parts to the probing analysis: *probe setup inventory* to list the possible alternative probe setups, and *probe effect analysis* to asses whether if the overhead of the current probe setup leads to a visible probe effect or not. These are described as follows:

3.1.1 Probe setup inventory

The subsequent model generation requires a minimal set of probing, which consists of probing task switches and performed system calls. In addition to that, task level probes are added to the implementation to record the dynamic state of variables in the system.

As the set of variables used in the implementation is finite, there is of course a finite set of possible probe setups, and these can be listed. Once these have all been tried unsuccessfully, it is concluded that model synthesis cannot be performed under current circumstances. The process is then aborted, but it may be possible to adjust the parameters of the process and make a subsequent attempt. The failure to complete the process may be due to one or several of the following reasons:

- The validity properties are too restrictive for the non-determinism of dynamic inputs etc. to the system.
- The probing required to capture the behavior of the implementation is too demanding. In this case, the overhead of probing has lead to that no real work could be performed by the system.
- The probing analysis failed to to find all alternatives during the probe setup inventory.
- The implementation is not suitable for model synthesis (e.g. the implementation does not conform to our system model specified in Section 2).

The probe setup inventory is not in any way inflicting on the performance of the model synthesis process – it is just a matter of whether there are more options or not. The analysis will consider the available variables in the implementation, and the history of used probe setups.

3.1.2 Probe effect analysis

In many cases, the probe effect [5] hinders removal of probes from the implementation because it (in the general case) cannot be determined that the presence of the probes, and the perturbation that this inflicts on the system, does not effect the behavior of the implementation. This is of interest as, if the behavior is effected, the performed testing of the implementation is invalidated by adjusting the perturbation of the probes (i.e. modifying the probe setup). In the case of model synthesis however, it is sometimes possible to make an exception.

The argumentation that present grounds for that this exception exists is as follows: As the probes are not explicitly cooperating with the implementation, the architecture of the implementation does not change with the addition or removal of probes. As long as the temporal behavior of the model that has been synthesized from recorded executions is modified accordingly (i.e. the execution time of the removed probes is subtracted from the measured execution time), some probes can be removed as the architecture of the implementation will not change based solely on the number of probes.

This is true, and can be performed as long as we can determine that the presence of the probes did not disturb the functionality of the system such that it triggers some *extraordinary behavior* (e.g. the probes are the source to some fault). If the recording was effected by extraordinary behavior, the model created from that recording is erroneous with respect to the implementation without probes. In that case, the probe setup must be kept until the system can be fully re-tested.

Assuming an implementation with two different probe setups, where one of the setups is a subset of the other, the following method can be used to determine whether if the presence of the additional probes changed the behavior of the system.

1. First, the records from probes not available in both the probe settings are removed from the recordings. This step includes modifying time-stamps by subtracting the time spent to execute the probes that are not available in both recordings. As motivated in other work, the execution time of probes should (for reasons of testability etc.) be constant [6].

2. Second, models are generated from each of the recordings.

3. Third, the two models are compared; if they are deemed to be equal, the probe effect had no impact.

Thus, if it is concluded that probes could be removed, the smaller of the two probe settings must be kept, but the additional probes can be removed. In this way, we can use the probe setups used in previous iterations of the model synthesis to remove subsequent additions to the probe setup. In the optimal case, the only probing that need to remain in the implementation is the minimal requirement of the model generation. The minimal probing is to probe the events task preemptions and performed system calls.

If the probe effect has no impact on the system, it has been avoided successfully in this case even if the probe setup is altered. The observation that is important to emphasize here is that if the overhead of task-level recording is low, the chance of avoiding the probe effect increases.

3.2 Implementation probing

The subsequent model generation assumes that it is possible to probe both *system*- and *task-level* events in the system. Probing of system-level events (i.e. context-switches and system calls) are mandatory, and will allow synthesis of a relatively crude and un-detailed model of the system. In order to refine that model, so that its behavior approaches that of the real implementation, task-level probes are added to extract information about the dynamic *state* (i.e. variable values) of the system.

Recording the state in the system by using task-level probes allow us to understand (and model) *causality* between jobs and events in the system – here, causality describes the fact that one event in one job effects the execution (i.e. the probabilities of future events and selections) of the continued execution. Understanding and being aware of these form of causal dependencies between actions and reactions is fundamental to making good behavioral models; in our work we allow causal dependencies to be reflected in the model generation by probing variables. Other possibilities to achieve a similar result are by using static analysis of source code or evaluating advanced guesses about causal dependencies between events using hypothesis testing etc.

The set of variables that are included in the task-level probing defines the quality of the model synthesis. However, excessive probing will lead to unnecessarily high overhead in the system and make the model more complex than necessary – thus, a trade-off has to be made: the set of variables that are probed should be sufficient, but the overhead of recording that set should be kept as low as possible.

Reducing the overhead could be achieved by deploying one or several of the following techniques:

- Selecting the set of variables that are the least demanding to record.
- Using logging algorithms that optimize logging in space or time requirements.
- Choosing read/write phase in which to probe the state of the variables such that the workload is lower (e.g. only on read if writes are frequent to the variable).
- Choosing system phase in which to probe the state of the variables such that the workload is disguised (e.g. recording while the system is idle).

Calculating the optimal probe setup is a non-trivial task. Thus, as this activity will have such an impact on the final product, several iterations of the model synthesis process are performed to search for an appropriate probe setup. Later in the process, a comparison between the implementation and the model is used as an evaluation to determine if the model from the current probe setup is sufficiently detailed. To reduce the number of iterations in the model synthesis process, the search for an appropriate setup uses heuristic methods (such as those for memory exclusive checkpoints as introduced by Plank [10]) to make suggestions for variables to be included in – or removed from – the probing setup.

3.3 Implementation execution

The probed version of the implementation is executed twice, and recordings rec_i^g and rec_i^v are thereby obtained (see Definition 2). The first of the recordings will be used for model generation (g), the second for model validation (v).

Definition 2. A $recording$ is an ordered sequence: $\langle entry_0, entry_1, entry_2 \ldots entry_{|recording|-1} \rangle$, where $entry$ describes an observation made in the execution. It is described by: $\langle e, t \rangle | e \in EventType$ where t is a list of observed time-stamps. The set $EventType$ is a specified set of events that can be observed in the system. An example is presented Table 1. □

The required length of the implementation execution is preset to a value and can then be renegotiated by the resolution analysis.

assign:	$\{variable_x, value\}$
send:	$\{queue_x\}$
receive:	$\{queue_x, blocking\ time\}$
lock semaphore:	$\{semaphore_x\}$
release semaphore:	$\{semaphore_x\}$

Table 1: Unique $EventTypes$ that are observed in the recordings.

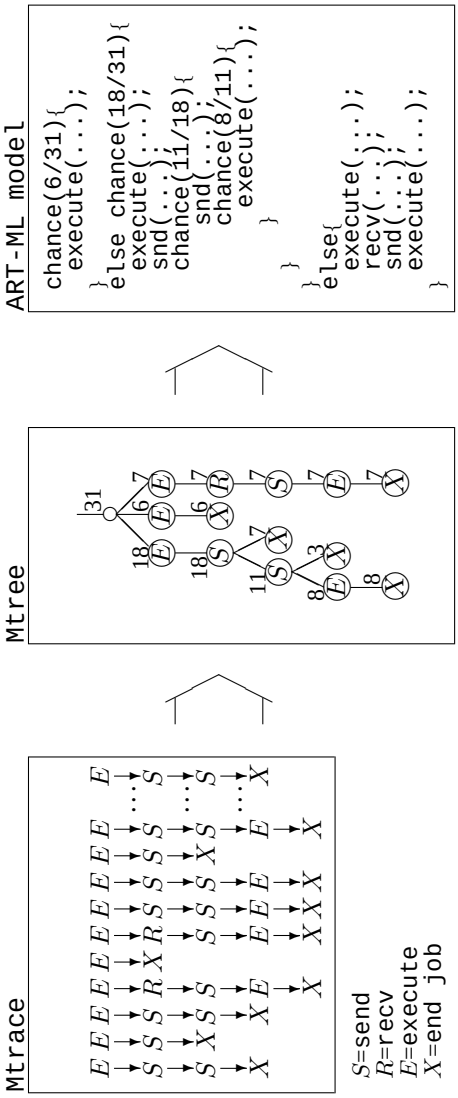

Figure 2: Model generation with chance-statements assuming system-level recording only.

$assign$:	$\{var_x, val\}$
$send$:	$\{q_x, msg\}$
$receive$:	$\{q_x, msg, blk\ time\}$
$execute$:	$\{time, prev\ action\ type\}$
end:	$\{time\}$

Table 2: Unique actions $a|a \in Actions$ in the Mtrace and in the Mtree have asociated properties.

3.4 Model generation

Our method for model generation has previously been described in [7].

The method has three steps, see Figure 2. The input to the process is the rec_i^g recording from the implementation execution, Definition 3 defines the output $Mtrace_i^g$ from the first step. Then, $Mtrace_i^g \backslash job_0$ is input to the second step.

Definition 3. For a given task, an Mtrace an ordered list of jobs J, where each job is an ordered list of events E whose elements are given by: $\langle a, G \rangle$, where:

- $a \in Actions$ is the action of the event, and
- $g_i \in G$ is the list of observed states for the event.

□

The motivation for removing the first job of the Mtrace is that the first instance of a task tends to have a rather different behavior compared to other jobs and it only occurs once in a recording. This provides too little data to make accurate predictions. To model a behavior that only occurs in the first instance of a task would require analyzing several measurements in order to get a sufficient amount of data on execution times and probabilities.

Definition 4 defines $Mtree$ that is the output from the second step.

Definition 4. $\mathbb{M}$ is a set of $Mtree$ whose elements are given by $\langle id, G, a, S, c \rangle$, where:

- id is a unique identifier,
- G, the guards, is an ordered list of variable constraints, where a $g_i \in G$ is a list of variable values that represent an accepted state for the Mtree,
- $a \in Actions$ is the action of the Mtree,
- $S \subseteq \mathbb{M}$, the set of successors, is the ordered list of the alternatives for subsequent execution after that a has been performed, and
- c is a counter.

□

3.5 Resolution analysis

The objective of the resolution analysis is to determine whether the execution time spent to produce the recordings is sufficient to capture a model in the given modeling language with respect to the complexity of the implementation in the current test case. In other words: is the execution time sufficient to describe a model in the language that we have chosen. The analysis is made on the data assembled to ensure that a longer execution time is not likely to give any additional detail.

We argue that it is reasonable to assume that the complexity of the implementation (and a given test case) and therefore also the complexity of the model will effect the required length of the recordings rec_i. Resolution analysis of the model is used to determine if the recording length used to obtain rec_i^g was sufficient.

Basically, resolution analysis is performed by observing the probabilistic assumptions of the model with respect to the *synthesis threshold* criteria that is supplied as input to model synthesis. The synthesis threshold is defined according to Definition 5.

Definition 5. The *synthesis threshold* consists of two parts: the *individual threshold*, which defines how many observations that are required for each event observed, and *overall threshold*, which controls how large part of the observed events that must fulfill the individual threshold. □

If a sufficient number (defined by the synthesis threshold) of parts of the model have sufficient observations (defined by the synthesis threshold) so as to make it likely that they are truthful, the length used to obtain the recordings is deemed sufficient – otherwise, the length is increased with some factor C. Note that the value of C has no impact on the quality of the model, but will effect the time required to successfully perform model synthesis and the amount of memory required to house the recordings. The same recording length is used for both instances of implementation execution as well as for the model simulation.

If the recording length is modified, the current iteration of model synthesis is back-traced, to the point of the two implementation executions. Both these are performed with the new length, and the process is continued from that point.

3.6 Continue

As a consequence of the resolution analysis, it is possible to conclude a failure of this instance of model synthesis. If the decision by the resolution analysis to increase the length of the recording does not help the resolution, we must sooner or later abort this instance of model synthesis. It is the task of the current step to determine at what point the process is no longer likely to succeed with its objective.

If the process seems unable to deem the resolution of the model fit under current conditions, it is likely that the test context supplied as input to the process is not restrictive enough. This will lead to that the resolution analysis is performed for the same i more than a predefined number of times so that the counter j exceeds a threshold value $iteration_{max}$. The optimal value of $iteration_{max}$ depends on the complexity of the implementation, the value of C, and the initial length of the recording.

In this case, if the process failed, it could be possible to redefine the test case and make another attempt to perform model synthesis.

3.7 Model validation

Following [8], we define *model validation* as the process of determining whether a model is an accurate representation of the system, for the particular objectives of the use. Thus, there is no immediate relation

between model validation and system validation, which uses models to validate that the requirements from the system specification are fulfilled by the system implementation/design (e.g. by model checking or testing).

In the context of model validity, it is important to bear in mind that the model is an abstraction and thus inherently diverse from the system. This is one of the difficulties with model validation – the model will be incorrect (or at least incomplete). The task is to validate that it is sufficiently correct to be useful in its intended context.

In our case, we intend to use the model for *impact analysis*: i.e., before a planned modification is implemented, the up-to-date model of the system is manually updated to correspond to that change. The new model can then be analyzed to evaluate the effects and the feasibility of the planned modification. The assumption is that it is considerably cheaper to modify the model than to modify the real implementation. As the starting point for impact analysis, we require an up-to-date model that has approximately the same temporal properties as the system it models. Model synthesis will deliver that model.

Previously, model validation has been a manual process performed by people with extensive knowledge in the system and/or in the modeling language. Balci [2] suggests a rage of approaches out of which this paper implements one: In *predictive validation*, in which the model is provided with authentic inputs and its output is compared to that of the system. In this paper, we implement this by extracting auxiliary data and comparing it with the model.

Szemethy and Karsai [12] 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. [11] provide an automated translation from their extended version of UML to timed automata. Contrary to our work, the goal of both Szemethy and Karsai and Shu et al. is to perform system validation rather than model validation. Further, their work does not consider models with data state, and queries performed on the model has to be tailor made for the specific system.

3.7.1 Preliminaries

The model generation delivers a model in a modeling language called ART-ML [13], which is a probabilistic modeling language. However, in this work, we use an equivalent notation based on internal structures of the model generation, called *Mtree* and *Mtrace*. An Mtrace can be always be transformed into an Mtree, which can always be translated into an ART-ML model. We use Mtree to represent the model that we wish to validate, and Mtrace to represent the recording used in the validation.

Both the Mtree and the Mtrace use a set $Actions$ as described by Table 2. Each element of $Actions$ has a type and a set of attributes as defined by the type of the element (e.g. the action *send* that has the attributes: queue identifier (q_x) and message body (msg)). The table should be seen as an example of $Actions$ – should the system require so, it is possible to extend the set to include other actions (e.g. semaphore actions) in an analogue manner to the IPC-actions send and receive.

Given an Mtree λ, we shall write $\lambda.id$ to refer to the unique identifier of the Mtree, and we shall write $|\lambda.S|$ to denote the number of successors of the node. Please note that the size of a guard (the number of variables in the data state) can be expressed as $|g|$, since it is an invariant for the specific model.

The recording used in model validation is of the form of an Mtrace, formally defined as follows:

In this paper, we present algorithms for how to transform these inputs to a set of two timed automata. We then describe how a simple analysis of the two automata and an auxiliary automaton can be used as model validation.

Timed automata In [1], Alur and Dill introduced the theory of timed automata. Since then, tools (e.g. UPPAAL [4]) for analysis of timed automata models have been constructed. Timed automata has also been extended, e.g. to incorporate integer variables [3].

Following [3], assuming that $\mathbb{Z}^{\mathcal{A}} \equiv \{0, 1, 2, 3 \ldots n_{max}\}$, a timed automaton is defined according to Definition 6. The automaton can use variables in the value domain $\mathbb{Z}^{\mathcal{A}} \cup \{-1\}$, where -1 represents an undefined value.

Definition 6 (timed automata with variables)**.** Assume a finite set of real-valued variables $\mathcal{C}$, ranged over by c and d, a finite set of variables $\mathcal{W}$ with values in $\mathbb{Z}^{\mathcal{A}} \cup \{-1\}$, ranged over by x and y, and a finite alphabet Σ of actions including the internal ε-action, ranged over by a and b. Let $\mathcal{B}(\mathcal{C})$ be the set of clock constraints of the form: $id \sim n$ or $id - d \sim n$ where $id \in \mathcal{C}$, $d \in \mathcal{C}$, $\sim \in \{\leq, \geq, =\}$, and $n \in \mathbb{Z}^*$. Let $\mathcal{B}(\mathcal{W})$ be the set of variable constraints on the form: $x \sim n$ or $x - y \sim n$ where $x \in \mathcal{W}$, $\sim \in \{\leq, \geq, =\}$, and $n \in \mathbb{Z}^{\mathcal{A}} \cup \{-1\}$.

Then, a timed automata $\mathcal{A}$ is a tuple $\langle N, l_0, E, I \rangle$ where:

- N is a finite set of locations,
- $l_0 \in N$ is the initial location,
- $U \subseteq N$ is a set of urgent locations that are exited before time is advanced,
- $E \subseteq N \times N \times \mathcal{B}(\mathcal{C}) \cup \mathcal{B}(\mathcal{W}) \times \Sigma \times 2^{\mathcal{C}} \cup 2^{\mathcal{W} \times \mathbb{Z}^{\mathcal{A}}}$ is the set of edges.
- $I : N \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations.

We write $l \xrightarrow{g,a,r} l'$ when $\langle l, l', g, a, r \rangle \in E$ where g is the constraint, or guard, of the transition, a is the action that caused it, and r is the clock reset and variable update performed during the transition. □

3.7.2 Example

To concretize the presentation, this section provides an example of how the set of generated automata can look like. We will refer to this paper later in the paper to exemplify our contribution.

In Figure 3, we show a simple example of how the automaton for an Mtrace can look like. In this example, we let $|g| = 2$.

Figure 4 is a timed automaton for an Mtrace that is used to test the Mtree. The automaton is serial, that is, from each label, there are no more than one edge.

To facilitate proper parameter passing, we use the auxiliary automaton in Figure 5. The use of this automaton is due to limitations in the expressiveness of timed automata – guards of an edge can only be checked before performing the action, and updates can only be performed after performing the action – that prevent us from communicating information about queue identifiers etc. before performing actions. There are other analogous ways to solve this problem.

3.7.3 Algorithms for automata generation

The two generated automata use an auxiliary automaton to communicate their parameters on send, receive, and update actions. This auxiliary automaton is generic and defined according to Definition 7. This can easily be extended to handle semaphore operations etc.

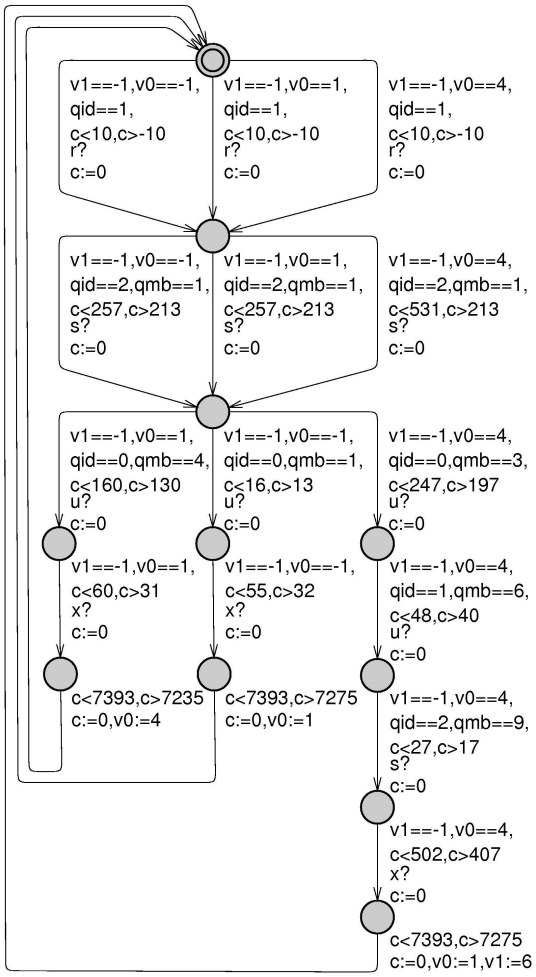

Figure 3: A timed automaton of an $Mtree_t$ example with $|g| = 0$.

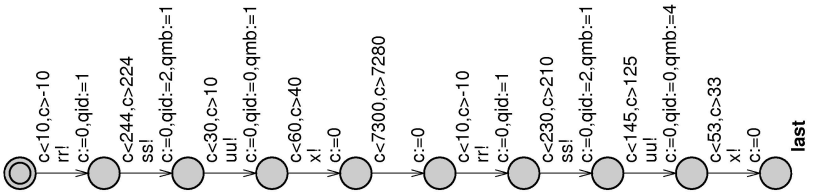

Figure 4: A timed automaton for a (short) $Mtrace_t$ that is accepted by the timed automaton of $Mtree_t$.

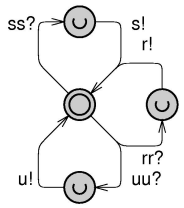

Figure 5: An auxiliary automata for parameter passing between the two automata, the labels marked with U are *urgent* (see Definition 6).

Definition 7 (timed automata with variables for the auxiliary automaton)**.** Let $\Sigma \equiv \{ss, s, rr, r, uu, u\}$, $\mathcal{C} \equiv \emptyset$, and $\mathcal{W} \equiv \emptyset$. Then, the auxiliary automata $\mathcal{A}^{aux}$ is defined as follows:

- $N \equiv \{l_0, l_1, l_2, l_3\}$,
- l_0 is the initial location,
- $U \equiv \{l_1, l_2, l_3\}$ is a set of urgent locations,
- $E \subseteq \{\langle l_0, l_1, \emptyset, ss, \emptyset\rangle, \langle l_1, l_0, \emptyset, s, \emptyset\rangle, \langle l_0, l_2, \emptyset, rr, \emptyset\rangle, \langle l_2, l_0, \emptyset, r, \emptyset\rangle, \langle l_0, l_3, \emptyset, uu, \emptyset\rangle, \langle l_3, l_0, \emptyset, u, \emptyset\rangle$.
- $I \equiv \emptyset$.

□

The remainder of this section will describe how to produce one automaton each from the Mtree (the model) and the Mtrace (a recording of a test-case) defined above. The automaton for Mtrace will be a simple serial automaton with one transition (edge) exiting from each node (location) except the last. The automaton for the Mtree will have a more elaborate tree structure. The actions found in the Mtree and Mtrace respectively will have their equivalence in the automata (plus the internal ε action). With the exception of execute-actions, the actions of the Mtree or Mtrace will be actions of an automaton, execute-actions are instead translated as clock guards on edges.

In order to produce these two automata, evident from Definition 6, we need to define the following:

- the maximum number of labels in the automaton,
- the initial location of the automaton,
- the set of edges of the automaton, these are given by:
  - the guards (originating both from variables, clocks, and parameters),
  - the action,
  - and the clock resets and the variable updates.
- and the invariants of each label.

To determine the exact number of locations for the Mtree automaton would require an extensive analysis – we choose to settle for determining an upper bound. As we will use the Mtree identifiers to identify the labels of the automaton, finding highest identifier max will provide that upper bound of labels ranging from l_0 to l_{max}. In the appendix A, this bound is supplied by the $mec : Mtree \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$, the **m**odel **e**vent **c**ount function. The corresponding number of max is supplied for the Mtrace by the $tec : Mtrace \times \mathbb{Z}^* \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$, the **t**race **e**vent **c**ount function. The difference is that, by counting all events save those with execute-actions (as the execute-actions are not modeled as actions on edges, but guards on an edge with another action), and adding one for each completed job, the tec-function delivers the real number of labels in the automaton rather than an upper bound on that number.

The initial location of both automata will always be l_0. Please note that, according to Definition 4, no Mtree can have identifier 0, which allows us to use the identifier to label the remaining labels of the Mtree automaton.

The edges of the automata are obtained from thorough analysis of the Mtree and Mtrace respectively. We have implemented this (see Appendix A) as recursive functions on the respective inputs: $mge : Mtree \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times list \rightarrow E$, the **m**odel **g**enerate **e**dges function, for Mtree and $tge : Mtrace \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times list \times \mathbb{Z}^* \rightarrow E$, the **t**race **g**enerate **e**dges function, for Mtrace. Both will analyze each action in the input and create edges to correspond to each non-execute-action – the execute actions are modeled as clock guards on the edge subsequent to the execute-action. Also, after the occurrence of an end action, an edge with an ε action will follow (see Figure 3). In the case of the Mtree automaton, performing the ε action will lead the automaton to l_0 – a transition with an ε action for the Mtrace automaton will lead to the label just after the current.

The guards of edges will hold both clock guards, variable guards, and parameter guards. The parameter guards ensure that send actions are concerning the same queue and message body in both the Mtree and the Mtrace etc. To perform the check, as timed automata does not allow us to transfer data without synchronization, we use an auxiliary automaton. First, the auxiliary automaton synchronizes with the Mtrace automaton, which assigns its parameters to global variables. Then, the auxiliary automaton synchronizes with the Mtree automaton so that it can use guards to check the parameters that were sent. In the case of receive actions, we shall neglect the message body parameter since the state of a message does not necessarily force the model in a certain path (should this be the case, subsequent actions will show this). We check both the variable identifier and the variable value of update actions.

Clock guards are obtained both from the parameters of execute and end actions – the execute action describes computation as the passing of time while parameters of the end action describes idling between jobs. The variable guards of the accumulated data-state are obtained from the guard of the Mtree – as the automata for Mtrace is serial, there is no need for data-state there.

The timed automata for Mtrace includes a *validity property* vp, supplied by the user, that is used to express the fact that the model is an timely abstraction from the system. The validity property is expressed as a time, and the usage is that any time observation t observed in the Mtrace is adjusted to be in the interval $(t - vp, t + vp)$. This will provide the ability to allow the model to pass validation event hough it is not exactly identical to the same. In the example above, the Mtrace automaton of Figure 4 use a validity property of 10. The example shows a situation where the validity property comes into play in the third Mtrace transition on the uu action. Since the guard is in the $[10, 30]$ interval and the validity property was 10, the original Mtrace measurement was 20 time units. The corresponding u action of the Mtree automata (in the center of the third row) has the clock guard $[13, 16]$. Without the validity property, these two automata could never synchronize on these premises.

For the Mtree automaton, the mge-function calls $mcv : \mathbb{Z}^* \times \mathbb{Z}^* \times list \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathcal{B}(\mathcal{C}) \cup \mathcal{B}(\mathcal{W})$, the **m**odel **c**lock and **v**ariable constraints function, to obtain the guards for a given edge. The guards of the

Mtrace automaton are simply defined by the tge-function.

For both automata, the local clock is reset on every edge. Apart from that, as described earlier, the Mtrace automaton use updates and synchronization with the auxiliary automaton to transfer parameters to the Mtree automaton. The updates are handled by the tge-function. In the case of the Mtree automaton, apart from the clock reset, the updates are concerning the state accumulated through update-actions. The semantics of these for the Mtree is that they should take effect only after after the end of the current job (or task instance) [7]. This can be observed in the Figure 3, where some updates are performed, but the state variables of the automaton are not changed until the ε action at the end of the job. This functionality is provided by $mss : list \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \rightarrow list$, the **m**odel **s**ave **s**tate function, that record the updates as they appear in update actions, and $mus : list \times list \times \mathbb{Z}^* \rightarrow \mathcal{W} \times \mathbb{Z}^{\mathcal{A}}$, the **m**odel **u**pdate **s**tate function, that use the recordings to produce the updates for the updates on the edge with the ε action.

Last, we will not use invariants in any of the two automata.

According to the above and Appendix A, the automata are defined according to definitions 8 and 9:

Definition 8 (timed automata with variables for $Mtree_t$)**.** Assume $\Sigma \equiv \{\varepsilon, s, r, a, x\}$, $\mathcal{C} \equiv \{c_m\}$, and $\mathcal{W} \equiv \{qid, qmb, g_0, g_1, \ldots, g_{|guard|-1}\}$. Then, the automata $\mathcal{A}_t^{Mtree}$, is defined as follows:

- $L \subseteq \{l_0, l_1, \ldots l_{mec(\lambda,0)}\}$,
- $U \equiv \emptyset$
- l_0
- $E \equiv mge(Mtree_t, 0, u)$, where u is a list such that every element in u is set to -1 and $|u| = |\mathcal{W}|$.
- $I \equiv \emptyset$

□

Definition 9 (timed automata for $Mtrace_t$)**.** Assume $\Sigma \equiv \{\varepsilon, ss, rr, a, x\}, \mathcal{C} \equiv \{c_t\}$, and $\mathcal{W} \equiv \{qid, qmb\}$. Let vp denote the validity property as specified by the user. Then, the automata $\mathcal{A}_t^{Mtrace}$, representing $Mtrace_t$, is then defined as:

- $L \equiv \{l_0, l_1, \ldots l_{tec(Mtrace_t)+1}\}$,
- $U \equiv \emptyset$
- l_0
- $E \equiv tge(Mtrace_t, 0, 0, 0, NIL, vp)$
- $I \equiv \emptyset$

We let the state $l_{tec(Mtrace_t)+1}$ be labeled "last". □

3.7.4 Decision procedure

The intention is that, after both the $Mtree_t$ (the model) and the $Mtrace_t$ (the test-case) have been transformed into their respective timed automaton, co-simulation of these is possible. In fact, if model generation has succeeded, there should be a way to co-simulate these two automata and a generic auxiliary automaton so that the last location of $\mathcal{A}_t^{Mtrace}$ is reached. This property can be checked with reachability analysis using the same techniques as in a model checker for timed automata (e.g. UPPAAL [4]) for small examples. However, for realistic examples, the number of locations in the automaton are too many for a general purpose model checker like UPPAAL to handle. As our requirements are quite simple, we need to test only one property of two automata with one clock each, and out of which at least one automata ($\mathcal{A}_t^{Mtrace}$) has a maximum of only one edge per location and no edge out from the accepting state, it is quite feasible to implement a lightweight model checker that can perform only this test. The complexity of the test is linear to the number of edges in the Mtrace automaton multiplied by the length of the Mtrace – even for traces of up to 5000 events, the computation time for our tailor made model checker is in the order of seconds.

The motivation for that this test will validate the version of the model produced by model generation is as follows: As the last location in the Mtrace is reachable, the Mtrace is included in the model. As the Mtrace is included in the model, the model refines the Mtrace, and therefore the model is a representation of the Mtrace which in turn is a representation of the implementation.

3.7.5 Model validity

Above, we presented a way to perform a test on the model to validate that it faithfully captures the system behavior. However, since our model is dependent on the test case traces, it is important that the traces used for model building are sufficiently representative of the systems behavior with respect to the properties of concern. By model validity we mean measures and procedures for ensuring the same. In other words, this implies that for a full-fledged automated model validation, we need to formulate a stopping criteria, which answers the question: Have we performed enough testing to make us sufficiently confident that the model is an adequate abstraction of the system? We use two validity measures on the jobs observed to determine this:

- *Completeness measure*, i.e. how likely is it that the system cannot exhibit a job that the model cannot replicate.
- *Accuracy measure*, i.e. how likely is it that the probability of that the system exhibits a particular job is the same as the probability that the model exhibits an equal job.

To assess these measures, we require a notion that allows us to determine whether two jobs can be considered equal in terms of the properties of interest. To meet this end, we define the notion of *syntactic equivalence* in the following manner.

Definition 10. Considering the validity properties assigned to the validation process, if the action sequences, variable updates, and the output of two jobs are the same, syntactic equivalence exists. □

By using syntactic equivalence, we can group observed jobs into several syntactic equivalence classes.

The completeness measure In order to estimate the completeness measure, we analyze the frequency with which new syntactic equivalence classes are discovered in the recording of the system. The set of known syntactic equivalence classes is the set identified in the recordings used to generate the model.

We assume that the probability of discovering a job of an unknown syntactic equivalence class follows a binomial distribution.

According to Piegorsch [9], there are more than one way to establish how many samples are required to obtain a good estimate of the probability that we seek for a binomial distribution. Piegorsch suggests one method that require the following inputs to obtain the sample size estimate: α – the significance, i.e. probability of falsely rejecting the null hypothesis, π_0 – the anticipated probability of the null hypothesis, and ε_0 – the margin of error.

If we state the null hypothesis that there are more syntactic equivalence classes than we have discovered so far, and let $\pi_0 \leq 0.01$, with $\alpha = 0.01$ and $\varepsilon_0 = 0.01$, we can deduce from [9] that the required sample size is lower than 1000 according to the Jeffreys $100(1-\alpha)\%$ confidence limits.

In the future, this estimate of the sample size should be complimented by an online method to evaluate the progress of discovering syntactic equivalence classes.

The accuracy measure ART-ML, the modeling language in which the model is expressed, is probabilistic [13] – that is, the model can use probability to determine selections at behavioral level – which requires that the model has valid estimate of the probabilities that the system exhibits jobs of different syntactic equivalence classes. The accuracy measure should determine that these estimates are valid.

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 syntactic equivalence classes and analyze the probability distributions between equivalence classes in the two sets G and V. The more similar the distributions are, 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. For the analysis, we let the function $sec\_count : 2^{job} \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$ calculate the number of jobs of a syntactic equivalence class in a set of jobs. The objective is to ensure that, for all identified equivalence classes n, the absolute value of the difference between $\frac{sec\_count(G,n)}{|G|}$ and $\frac{sec\_count(V,n)}{|V|}$ is in the interval $(0, h_a)$.

For example, say that we have identified three syntactic equivalence classes for a model of a task, these are labeled sec_0, sec_1, and sec_2. For these, the probability distributions are $P(sec_0) = 30\%$, $P(sec_1) = 50\%$, and $P(sec_2) = 20\%$. The accuracy threshold is set to $h_a = 1\%$.

After that validation has model checked Mtraces with a total of c jobs, of which no Mtrace found an error in the model, the validation measures are examined: The accuracy measure is $P_c(sec_0) = 20\%$, $P_c(sec_1) = 55\%$, and $P_c(sec_2) = 25\%$ – which is not within h_a.

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.

Using the validity measures The validity measures described above can be used as stopping criteria for the automated model validation process. The user just specifies a set of intuitive parameters, and initiates the process – the process will then continue for as long as needed to either fulfill the validity measure requirements or conclude a failure to meet the required validity measure.

4 Conclusion

In this report, we have presented a process for model synthesis. The iterative process does not require intervention from humans; it is capable of evaluating its own operation.

4.1 Future work

In our future work, we will attempt to describe the search for an optimal probe setting as an optimization problem, which may provide a quicker way to an optimal setup.

References

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

[2] Osman Balci. Guidelines for successful simulation studies. In *Proceedings of the Winter Simulation Conference*, pages 25–32, December 1990.

[3] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated analysis of an audio control protocol using uppaal. *Journal of Logic and Algebraic Programming*, 52-53:163–181, July-August 2002.

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang 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] Jason Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

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

[7] Joel Huselius and Johan 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] Averill M. Law and Michael G. McComas. How to build valid and credible simulation models. In *Proceedings of the Winter Simulation Conference*, pages 22–29, December 2001.

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

[10] James Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, 1999.

[11] Guoqiang Shu, Chao Li, Qing Wang, and Mingshu 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.

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

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

A Formulas for automata generation

This appendix present the set of formulas used for generating the automata for the Mtrace and the Mtree.

A.1 Timed automata for $Mtree_t$

The non-deterministic automata is constructed by the aid of five formulas. We assume that the automata has a set of clocks $\{c_m\}$, and a set of variables $\{qid, qm, g_0, g_1, \ldots, g_{|guard|-1}\}$, where qid will represent IPC-queue identifiers, qmb will represent IPC-queue message bodies, and g_x is a member of the variable state.

$$mss(u, id, v, p) = \begin{cases} NIL & when\ p{=}|g| \\ [v|mss(u, id, v, p{+}1)] & when\ p{=}id \\ [u_p|mss(u, id, v, p{+}1)] & otherwise \end{cases} \tag{1}$$

Intuitively, $mss : list \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \rightarrow list$ (Formula 1, **m**odel **s**ave **s**tate) updates a list according to the input that describe the index in the list that should be changed, and the value to which that index should be set. The function is used to record the state information as it is updated in the model.

$$mus(guard, u, n) =$$

$$\begin{cases} \{\langle g_n, u_n\rangle | g_n \in \mathcal{W} \wedge u_n \in \mathbb{Z}^{\mathcal{A}}\} \cup mus(guard, u, n+1) & when\ u_n \neq -1 \wedge 0 \leq n < |g| \\ mus(guard, u, n+1) & when\ u_n = -1 \wedge 0 \leq n < |g| \\ \emptyset & otherwise \end{cases} \tag{2}$$

Intuitively, $mus : list \times list \times \mathbb{Z}^* \rightarrow \mathcal{W} \times \mathbb{Z}^{\mathcal{A}}$ (Formula 2, **m**odel **u**pdate **s**tate) produces a set of updates from the list that has previously been compiled by a series of calls to mss. The semantics of the model is that variable updates should take effect only after task completion [7]. mus is called to produce the observed updates as the last action of the task before it makes a transition back to the initial location l_0 of the automaton.

$$mcv(e_\uparrow, e_\downarrow, v, id, mb) =$$

$$\begin{cases} \{c_m{\leq}e_\uparrow, c_m{\geq}e_\downarrow\} & when\ v \equiv NIL \\ \{c_m{\leq}e_\uparrow, c_m{\geq}e_\downarrow\} \cup \{qid{=}id\} \cup \{qmb{=}mb\} \bigcup_{p=0}^{|g|-1} \{v_p{=}g_p | g_p \in \mathcal{W}\} & when\ qid{\neq}{-}1 \wedge qmb{\neq}{-}1 \\ \{c_m{\leq}e_\uparrow, c_m{\geq}e_\downarrow\} \cup \{qid{=}id\} \cup \bigcup_{p=0}^{|g|-1} \{v_p = g_p | g_p \in \mathcal{W}\} & when\ qid \neq -1 \\ \{c_m{\leq}e_\uparrow, c_m{\geq}e_\downarrow\} \cup \bigcup_{p=0}^{|g|-1} \{v_p = g_p | g_p \in \mathcal{W}\} & otherwise \end{cases} \tag{3}$$

Intuitively, $mcv : \mathbb{Z}^* \times \mathbb{Z}^* \times list \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathcal{B}(\mathcal{C}) \cup \mathcal{B}(\mathcal{W})$ (Formula 3, **m**odel **c**lock and **v**ariable constraints) produces a set of clock and variable constraints. The result is used to populate the guards on edges.

$$
mge(\lambda, r, e_{\uparrow}, e_{\downarrow}, u) =
$$

$$
\begin{cases}
\{\langle l_r, mcv(e_{\uparrow}, e_{\downarrow}, NIL, -1, -1), \varepsilon, CRM \cup mus(\lambda.G, u, 0), l_0 \rangle\} & when\ \lambda \equiv \emptyset \\
\bigcup_{p=0}^{|\lambda.S|} mge(\lambda.S_p, r, \max(\lambda.a.time), \min(\lambda.a.time), u) & when\ \lambda.a = execute \\
\bigcup_{p=0}^{|\lambda.G|} \{\langle l_r, mcv(e_{\uparrow}, e_{\downarrow}, \lambda.G_p, \lambda.a.var_x, \lambda.a.val), a, CRM, l_{\lambda.id} \rangle\} \cup & when\ \lambda.a = assign \\
\quad \bigcup_{p=0}^{|\lambda.S|} mge(\lambda.S_p, \lambda.id, 0, 0, mss(u, \lambda.a.val, \lambda.a.var_x, 0)) & \\
\bigcup_{p=0}^{|\lambda.G|} \{\langle l_r, mcv(e_{\uparrow}, e_{\downarrow}, \lambda.G_p, -1, -1), x, CRM, l_{\lambda.id} \rangle\} \cup & when\ \lambda.a = end \\
\quad \bigcup_{p=0}^{|\lambda.S|} mge(\lambda.S_p, \lambda.id, \max(\lambda.a.time), \min(\lambda.a.time), u) & \\
\bigcup_{p=0}^{|\lambda.G|} \{\langle l_r, mcv(e_{\uparrow}, e_{\downarrow}, \lambda.G_p, \lambda.a.q_x, -1), r, CRM, l_{\lambda.id} \rangle\} \cup & when\ \lambda.a = receive \\
\quad \bigcup_{p=0}^{|\lambda.S|} mge(\lambda.S_p, \lambda.id, 0, 0, u) & \\
\bigcup_{p=0}^{|\lambda.G|} \{\langle l_r, mcv(e_{\uparrow}, e_{\downarrow}, \lambda.G_p, \lambda.a.q_x, \lambda.a.msg), s, CRM, l_{\lambda.id} \rangle\} \cup & when\ \lambda.a = send \\
\quad \bigcup_{p=0}^{|\lambda.S|} mge(\lambda.S_p, \lambda.id, 0, 0, u) &
\end{cases}
\tag{4}
$$

We let CRM denote the set of clock resets $\{c_m\} \in \mathcal{B}(\mathcal{C})$. Intuitively, $mge : Mtree \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times list \rightarrow E$ (Formula 4, **m**odel **g**enerate **e**dges) produces the set of edges for the automaton. Of the set of actions listed in Table 2, all but the execute-action are modeled as a transition with that action, but the idle period following an end-action is also modeled as a guard on the subsequent ε-edge to l_0. To model the execute-action, the time stamps observed for the action are forwarded and included in the subsequent transition.

$$
mec(\lambda) = \begin{cases} 0 & when\ \lambda \equiv \emptyset \\ \max_{p=0}^{|\lambda.S|}(mec(\lambda.S_p), \lambda.id)) & otherwise \end{cases}
\tag{5}
$$

Intuitively, by finding largest Mtree identifier, $mec : Mtree \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$ (Formula 5, **m**odel **e**vent

count) calculates the number of locations required to construct the automaton for the Mtree.

A.2 Timed automata for $Mtrace_t$

For the deterministic Mtrace automata, we use two functions. We assume that the automata has a set of clocks $\{c_t\}$, and a set of variables $\{qid, qm\}$, where qid will represent IPC-queue identifiers, and qmb will represent IPC-queue message bodies.

$$tge(\theta, p, q, n, t, vp) =$$

$$\begin{cases} NIL & when p \geq |\theta| \\ tge(\theta, p+1, 0, n, NIL, vp) & when\ q \geq |\theta.J_p| \\ tge(\theta, p, q+1, n, \theta.J_p.E_q.a.time, vp) & when\ \theta.J_p.E_q.a{=}execute \\ tge(\theta, p+1, 0, n+2, NIL, vp) \cup & when\ \theta.J_p.E_q.a = end \\ \quad \langle l_n, l_{n+1}, GRDEXE, x, \{c_t\}\rangle \cup & \\ \quad\quad \langle l_{n+1}, l_{n+2}, GRDEPS, \varepsilon, \{c_t\}\rangle & \\ tge(\theta, p, q+1, n+1, NIL, vp) \cup & \theta.J_p.E_q.a = receive \\ \quad \langle l_n, l_{n+1}, GRDEXE, rr, \{c_t, \langle qid, \theta.J_p.E_q.a.q_x\rangle\}\rangle & \\ tge(\theta, p, q+1, n+1, NIL, vp) \cup & \theta.J_p.E_q.a = send \\ \quad \langle l_n, l_{n+1}, GRDEXE, ss, & \\ \quad\quad \{c_t, \langle qid, \theta.J_p.E_q.a.q_x\rangle, \langle qmb, \theta.J_p.E_q.a.msg\rangle\}\rangle & \\ tge(\theta, p, q+1, n+1, NIL, vp) \cup & \theta.J_p.E_q.a = assign \\ \quad \langle l_n, l_{n+1}, GRDEXE, uu, & \\ \quad\quad \{c_t, \langle qid, \theta.J_p.E_q.a.var_x\rangle, \langle qmb, \theta.J_p.E_q.a.val\}\rangle & \end{cases} \tag{6}$$

For simplicity, we make use of the following definitions of guards:
$GRDEXE \equiv \{\min(t) - vp \leq c_t, \max(t) + vp \geq c_t\}$
$GRDEPS \equiv \{\min(\theta.J_p.E_q.a.time){-}vp \leq c_t \max(\theta.J_p.E_q.a.time){+}vp \geq c_t\}$
Intuitively, $tge : Mtrace \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times list \times \mathbb{Z}^* \rightarrow E$ (Formula 6, **t**race **g**enerate **e**dges) returns the set of edges in the automaton. The philosophy of the construction is identical to that of the automaton for the Mtree in the modeling of end-actions and execute-actions.

$$tec(\theta, p, q) = \begin{cases} 0 & when\ p \geq |\theta| \\ tec(\theta, p+1, 0) & when\ q \geq |\theta.J_p| \\ tec(\theta, p, q+1) & when\ \theta.J_p.E_q.a = execute \\ tec(\theta, p, q+1) + 1 & otherwise \end{cases} \tag{7}$$

Intuitively, by counting the events of the Mtrace save all execute-actions, $tec : Mtrace \times \mathbb{Z}^* \times \mathbb{Z}^* \rightarrow \mathbb{Z}^*$ (Formula 7, **t**race **e**vent **c**ount) calculates the number of locations required to construct the automaton for Mtrace.