Automatic Translation from UPPAAL to C

JESPER KRISTENSEN ARNE MEJLHOLM SØREN PEDERSEN
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark
{cableman|mejlholm|peng}@cs.aau.dk

***Abstract*— In this paper we propose a deterministic semantic simplification of a given real-time UPPAAL model. We then propose a translation from the simple model to an actual implementation written in C and we argue that the implementation is correct. This translation is implemented in the compiler tool uppaal2c, which generates implementations that run on hard real-time systems (RTLinux). Furthermore we show test results and benchmarks for a number of UPPAAL models.**

I. INTRODUCTION

UPPAAL [1] [BD] is a tool to do verification of real-time systems. More specifically it uses networks of timed automata (as introduced in [AD94] and modified with invariants by [TXJS92]) extended with channel synchronization and data structures.

The process of first modelling a solution to a problem, then verifying the solution and finally implementing the solution may be a tedious task. While the first two parts of the task may not easily be automated, we propose to automate the last part in a tool called `uppaal2c`. This offers the following advantages: a vast reduction in the number of bugs introduced when doing an implementation of a verified model plus a reduction in the time spent on the implementation.

In this article we use the notion of a NTAS (Non-deterministic Timed Automata System), which is the kind of networks used in UPPAAL. We refer to it as a NTAS because it contains the possibility of choosing non-deterministically between possible transitions and doing delays. The first part of the article is concerned with converting any NTAS to a DTAS (Deterministic Timed Automata System, which has a close relationship to Deterministic Timed Automata as proposed in [AD94]). This will enable us to do an automated implementation of the simplified model.

[1]If unfamiliar with UPPAAL, read either [BDL04] or [LPY97]

The prototype of the tool `uppaal2c`, which can be found at [JKP], is written in Python and is made to be compatible only with the 3.4.x series of UPPAAL. It produces implementations of UPPAAL models, which runs on a RTLinux (see [Inc]) kernel. We provide a small how-to about getting RTLinux up and running on [JKP] as well as generated examples.

The rest of the article is organized as follows: Section II shows how to simplify a class of models to one that can be implemented. Section III addresses high-level decisions about choice of platform and other implementation specific aspects. Section IV shows the actual implementation. Section V argues that the implementation is correct. In VI we show test results and benchmarks of the tool. Section VII addresses things in the implementation that needs further elaboration and compares our work with that of others. Finally in Section VIII we conclude the paper.

II. CONVERSION FROM TIMED AUTOMATA SYSTEM TO DETERMINISTIC TIMED AUTOMATA SYSTEM

The reason why we want to convert a NTAS to a DTAS lies in the nature of implementing a timed model. Normally, as stated by [WDR04], you would refine a given high-level description into a low-level description that still preserves the important properties of the original model. One important step in doing so is to remove nondeterminism.

If we eliminate nondeterministic choice, then we will have a model that has the same behavior each time it is executed.

Furthermore it is impossible to implement clocks with infinite precision on hardware, as this requires a finite precision[WDR04]. Thus to implement a network of timed automata, we have to address at least two things: Non-determinism and the progression of time.

Determining which class of timed automata can be implemented is, however, an ongoing field of research and it is difficult to determine whether a given model can actually be implemented. Both [Kat99] and [WDR04] present a class of timed automata which cannot be implemented because they have the "zeno" property. This is a class of timed automata, which take an infinite number of transitions in a finite amount of time. It should be clear that this is un-implementable.

Bearing this in mind we apply the following informal restrictions to transform a given NTAS to a DTAS:

***Restriction** 1:* If, at any time step, more than one transition is enabled in a single automaton, then the execution is halted.

***Restriction** 2:* If a single transition (in a single automaton) is enabled, then it is always taken.

***Restriction** 3:* Time progresses by one for each time unit in the model and each automaton takes one transition or none at all when time progresses.

***Restriction** 4:* A binary channel can only be used for one synchronization between two automata per time unit.

A. Semantics

We now formally show the new semantics of a network of timed automata, therefore we introduce the following terminology:

- $A = \{a_1, a_2, \dots, a_n\}$ is a set of automata.
- $S = \langle A, U, V\rangle$ is the entire system comprising the set of automata A, which are running in parallel.
- U is the set of clocks in the system.
- V is the set of variables in the system, including channels.[2]
- $L = \{l_1, l_2, \dots, l_n\}$ is the set of locations in an automaton a
- C is the set of current locations in all the automata in S.
- $T = \{t_1, t_2, \dots, t_n\}$ is a set of transitions.
- $G = \{g_1, g_2, \dots, g_n\}$ is a set of guards.
- $I = \{i_1, i_2, \dots, i_n\}$ is a set of invariants.
- $F_G : T \rightarrow G$ is a function that takes a transition and returns a set of guards.
- $F_I : L \rightarrow I$ is a function that takes a location and returns the set of invariants of the location.
- $F_T : L \rightarrow T$ is a function that takes a location and returns the set of outgoing transitions from that location.
- α is the evaluation of the set of expressions on active transitions in all the automata and the side effects they may have[3].
- $\sum_a$ is the number of enabled transitions in the automaton a. If this is ever higher than 1, then the system contains nondeterminism and contradicts Restriction 1.
- $U \models F_I(l)$ means that U satisfies the invariant of location l and likewise when used with guards.
- $U' = [\alpha]U$ means that U' is equal to the values of U, except for the assignments made by α.
- $l \xrightarrow{\alpha} l'$ means that the automaton takes a transition from l to l' while executing assignments α.
- $l \xrightarrow{d} l$ means that the automaton performs a delay.

We then use this terminology to show relevant parts of the syntax of a timed automaton, the semantics of a NTAS and then the semantics of a DTAS. The part of the semantics of UPPAAL that

[2]This is actually not entirely correct, as UPPAAL allows for global declarations and declarations that are local to each automaton. We will omit this detail.

[3]see [BDL04, p. 5] or the UPPAAL help for an EBNF of how these expressions may look and behave.

have not been changed are left out[4].

***Definition** 1:* (*Timed (Safety) Automaton*) A timed automaton is a tuple $\langle L, l_0, F_I, F_G, \alpha\rangle$ where $l_0 \in L$ is the initial location.

Whereas the semantics for the regular timed automata concerns itself only with single automaton, we need to describe the entire system of automata taking transitions at the same time (on the tic of the clock).

***Definition** 2:* (*Nondeterministic Timed Automata System*) A NTAS is a tuple $\langle A, U, V\rangle$ and the transition system is defined by the transition relations:

- $\langle C, U, V\rangle \rightarrow \langle C', U', V'\rangle$ where:
- iff for each $l \in C$:
  - $U' = U + d$
  - $l \xrightarrow{d} l$ where $U', V \models F_I(l)$ and d is a non negative real.
- or:
  - $U' = [\alpha]U$
  - $V' = [\alpha]V$
  - a transition $t \in F_T(l) \in C$:
    - $l \xrightarrow{\alpha} l'$ where $U, V \models F_G(t)$ and $U', V' \models F_I(l')$
  - or two transitions $t_i, t_j \in C$ with synchronization in two automata $a_i, a_j \in A$:
    - $l_i \xrightarrow{c!, \alpha_i} l'_i$ and $l_j \xrightarrow{c?, \alpha_j} l'_j$ where α_i is executed before α_j, $U, V \models F_G(t_i), F_G(t_j)$ and $U', V' \models F_I(l'_i), F_I(l'_j)$ and c is a binary channel.
  - or a number of transitions $t_0, \ldots, t_n \in C$ in a number of automata $a_0, \ldots, a_n$ which does broadcast:
    - $l \xrightarrow{c!, \alpha_t} l'$ and $l_0 \xrightarrow{c?, \alpha_0} l'_0, \ldots, l_n \xrightarrow{c?, \alpha_n} l'_n$ where $U, V \models F_G(t), F_G(t_0), \ldots, F_G(t_n)$ and $U', V' \models F_I(l'), F_I(l'_0), \ldots, F_I(l'_n)$ and c is a broadcast channel. Assignments are carried out in the order: $\alpha_t, \alpha_0, \ldots, \alpha_n$

[4]We do not aim to provide support for urgent channels, broadcast synchronization, urgent or committed locations and as a consequence we disregard them completely. We will address them further in Section VII

[AD94] defines a Deterministic Timed Automaton, as one that only has one start location and if there is more than one outgoing transition from a location, then the guards on the transitions must be mutually exclusive. We relax this notion a bit, so we do not have to check this property. Instead we count the number of enabled outgoing transitions from one location, denoted by $\sum_a$.

As the NTAS may nondeterministically choose any delay d as long as it satisfies the guards and invariants, we also have to make time discrete. This is done by introducing a constant time unit δ.

***Definition** 3:* (*Deterministic Timed Automata System*) We define our DTAS as a NTAS except that we define the transition relation in a DTAS in this manner:

- $\langle C, U, V\rangle \rightarrow \langle C', U', V'\rangle$ where:
- $V' = [\alpha]V$
- $U' = [\alpha]U + \delta$ where δ is a constant non negative integer
- iff for each location $l \in C$:
  - for each transition $t \in F_T(l)$:
    - $l \xrightarrow{\alpha} l'$ where $U, V \models F_G(t)$, and $U', V' \models F_I(l')$ iff $\sum_a = 1$
    - or:
      - $l \xrightarrow{\delta} l$ where $U', V' \models F_I(l)$ iff $\sum_a = 0$

Furthermore the binary synchronization and broadcast transitions have not been altered.

B. Semantical Differences

The semantics further limits the class of models, which we can implement. To explain this we need further notation:

$* \in \{\alpha, d\}$ meaning that $\xrightarrow{*}$ is either a transition or a delay.

For each $*$ we count $n_c = n_c + 1$ where c is a clock in U. If c is updated by a transition n_c is set to the new value of c, $n_c = c$. Furthermore we define a set of non-negative integers $I = \{0, \ldots, n-1\}$.

***Definition** 4:* (*Time Progression Property*) If there exists a sequence of transitions $l \xrightarrow{*} l'$

where $l, l' \in L, i \in I$ in a model and the expression
$(U + i \models F_G(l')$ and not $U + n \models F_G(l'))$ or
$(U + i \models F_I(l')$ and not $U + n \models F_I(l'))$
is true, then the model has the Time Progression Property.

Informally explained: If there exists a sequence of transitions in an automata, where time does not progress at least as much as our semantics require, then we cannot implement the model. This property also includes the non-zeno property.

If the Time Progression Property is true for a model, then the model will not work with our DTAS semantics and we cannot implement it.

Furthermore the changes of semantics have a number of consequences. First we address time:

Time changes through the set of clock variables U. In a NTAS this is updated by assignments α and by doing delays d.

A DTAS is able to do assignments α, but each clock in the set U is always incremented by one time unit for each transition. Thus any delay d in a given DTAS is turned into a discrete representation, namely a sequence of integer valued time steps, $D = \{i_{0\delta}, i_{1\delta}, \ldots, i_{k\delta}\}$, where $k = d$ and δ is a constant time difference separating the elements.

It is not difficult to see that D is a partitioning of d and therefore $D \subseteq [0 : d]$. While a NTAS will have a set of regions:

$$[i_0],]i_0 : i_1[, [i_1],]i_1 : i_2[, \ldots,]i_{k-1} : i_k[, [i_k]$$

a DTAS will only have the regions:

$$[i_0], [i_1], \ldots, [i_{k-1}], [i_k]$$

because we have to reserve time to actually perform transitions or delays. Therefore we call the $]i_{n-1} : i_n[$ a dead region, and require a given model does not rely on guards or invariants that are only satisfied in these dead regions. An example of such a guard would be $2 > x < 3$.

C. Relation between DTAS and NTAS

Now we show the relation between a DTAS and a NTAS, to explain this we introduce the terms *state* and *trace*.

Definition 5: (*State*) A single state is the state of the different automata in a timed automata network, $state$, at a given time, and it is written as a triple $\langle C, U, V\rangle$. A state $state_n^X$ is the state n in trace X.

Normally, in timed automata theory you would define a trace as a sequence of actions and delays, $a_i, d_i, a_{i+1}, d_{i+1}$. As UPPAAL does not have actions on transitions, only assignments, and we always have a constant delay, we will define a trace as:

Definition 6: (*Trace*) A trace of a timed automata network, $trace$, is a sequence of states, $trace = \langle state_0, \ldots, state_n\rangle\}$.

NTAS do not have a single trace, but a set of different traces, which grows exponentially with each possible transition in the system. This number of transitions is limited to just one by our semantics, thus we give the following theorem:

Theorem 1: A DTAS trace is an element in the set of traces for a NTAS.

PROOF IDEA: We show that one and only one element is chosen from the set of traces of a NTAS. We do this by addressing the changes in the semantics.

PROOF:

a) Transitions: To do transitions the NTAS requires that $U, V \models F_G(t)$ and $U, V' \models F_I(l')$ (time does not progress on a transition, so it is not U').

A DTAS requires that $U, V \models F_G(t)$, as well as $U', V' \models F_I(l')$ iff $\sum_a = 1$. The $\sum_a = 1$ clause makes sure that only one transition is enabled in one automaton at a time. Thus a DTAS is only valid if there is one and only one transition which satisfies its guards in an automaton. If there is no more than one valid transition in each automaton at a time, then we know that there is only one possible trace for the given DTAS.

b) Delays: Finally, we have to take delays into consideration. A delay in a DTAS requires that

V' is satisfied (as one automaton in the system may update V while another automaton is doing a delay) and that $\sum_a = 0$. As $\sum_a = 1$ and $\sum_a = 0$ are mutually exclusive, there can only be one transition or delay per automaton per time unit.

This proves that the set of traces of a DTAS is of size 1.

Given Definition 4 we know that time in the NTAS progresses at least by a constant factor. As the DTAS progresses by the same factor we know that they progress along the same trace. Therefore a trace of a DTAS must be one of the possible traces, in the set of trace, of the NTAS, which the DTAS is derived from.

□

III. Realizing the implementation

To actually implement a DTAS as proposed in the previous section, we need a few more guarantees.

Since the models to be translated are a number of automata running in parallel, it is obvious to suggest a framework that is running each automaton on a separate CPU. This demand for concurrency is not strictly necessary and a satisfactory solution can also be obtained by the use of multiprogramming on a single CPU. Due to the use of discrete time we only need to guarantee that each automaton takes exactly one transition during each step. On this matter the use of multiple CPU will increase the chance that all automata has adequate time in a large model.

Secondly when a network of timed automata are running we need to make sure that they will not be subjected to context switch. If a process is allowed to be switched out, then there is a high probability that the clock will skew. This is because its impossible to make any guarantees about execution order and time progression in the different timed automata.

One way to overcome these obstacles is to make use of a hard real-time system[5] .

Definition 7: (*Hard Real-time System*) A hard real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

By using a hard real-time system we get guarantees that a process will execute once and only once for every time unit and if an execution is longer than one time unit, a system failure will occur. Real-time systems guarantee that an active process is not switched out of context and we can decide which process is executed on which CPU. Thus we are able to implement our design of the DTAS.

Furthermore, as UPPAAL is designed to do verification of real-time embedded systems, the choice of a hard real-time operating system does not seem unreasonable as a choice of platform.

IV. Translation of the DTAS

Given the guarantees provided by real-time systems as discussed in the previous section, we can now describe the translation from DTAS to C code. The translation is performed by the algorithm in Listing 1.

```
read UPPAAL xml-file
system := find all timed automata
    from templates, declarations and
    instantiation
for each timed automaton in system:
    write system initializing
        information for timed
        automaton
    write local declarations
    for each location in timed
        automaton:
        for each invariant in
            location:
            write invariant check
```

[5]There seems to be no agreed definition of a hard real-time system in the literature, but this one (from http://www.faqs.org/faqs/realtime-computing/faq/) is in concordance with our view.

```
        for each outgoing transition in location:
            for each guard in transition:
                write guard check
            for each synchronization in transition:
                write synchronization check
            for each assignment in transition:
                write code that makes assignments iff transition is chosen
        write nondeterminism checking code
write main file that initializes global declarations and timed automata as threads
```

Listing 1. The translation algorithm from DTA to C code

The first thing it does is to read the `xml` file and return an AST (Abstract Syntax Tree), which we run through recursively. We create objects from the information in the tree, which are then linked together in a hieratical structure by an `Environment` object, which is the top element.

When all information has been read and put into objects in the `Environment` structure, we have to process the information. This step mainly converts `Template` objects to `TimedAutomaton` objects, but also creates global lists of clocks and other global declarations. Finally whenever the environment has been processed the actual code generation can begin.

Each `TimedAutomaton` object in the `Environment` is translated to a single thread in the running C system and an additional thread is generated to handle the clocks.

As the algorithm runs through every element in Definition 3 on page 3, we can conclude that the algorithm at least runs through all information in the xml file and the elements of a Timed Automaton.

Given this overall view, the following subsections address the details of the implementation.

A. Threads

In the generated system there will be total of $n+1$ threads, where n is the number of timed automata in the system. Each automaton will run in its own thread in a `while` loop that will only terminate if an error is detected. This loop is designed in such a way that it makes exactly one iteration for each time unit elapsed in the corresponding DTAS. The structure of the generated code is shown in Listing 2 and the code generated for each is described in the following sections.

The thread that governs the time is designed in such a way that all the clocks is incremented once for each iteration of the `while` loops in the automata threads. This synchronization is obtained by using a functionality in the target platform that ensures that each thread is run only once during each time unit.

```
// Thread synchronization
while (!error) {
  switch (location) {
    case 0: // State A
      // Check Invariant
      // Check outgoing transitions
      // Set synchronization variables
      // Thread synchronization
      // Check for synchronization
      // If synchronization then check execution order of assignments
      // Take selected transition
      // Sleep until next time unit
      break;
    case 1: // State B
      .
      .
      .
      break;
  }
}
```

Listing 2. `Switch` structure for locations

1) Thread synchronization: We synchronize the threads to ensure that no transitions are taken before all automata has checked the guards on their outgoing transitions. This synchronization is done once each time unit. The thread synchronization is illustrated for three threads in Figure 1 with one CPU and with a multi CPU system in Figure 2.

The marked positions in Figure 1 are:

a. The first thread reaches the synchronization point and halts.
b. The second thread reaches the synchronization point and halts.
c. The final thread reaches the synchronization point and awakens the other threads, then finishes its run and sleeps for the remainder of the time unit.
d. Thread number one has performed its run and sleeps the rest of the time unit.
e. Thread number two has performed its run and sleeps the rest of the time unit.

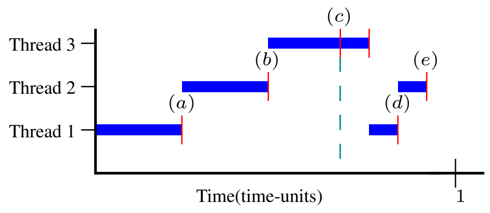

Fig. 1. The synchronization of 3 threads during a single time unit on one CPU

In the multi CPU system, Figure 2, the synchronization is a *line-up* of the threads:

a. The first thread reaches the synchronization point and halts.
b. The second thread reaches the synchronization point and halts.
c. The final thread reaches the synchronization point and awakens the other threads, which then finish their run and sleeps for the remainder of the time unit.

The synchronization is done by using a conditional variable (which is a mutual exclusion primitive), which is governed by the function in Listing 3. The function uses the `cond_counter` variable to count how many times the function is called. While the counter is not equal to the number of threads,

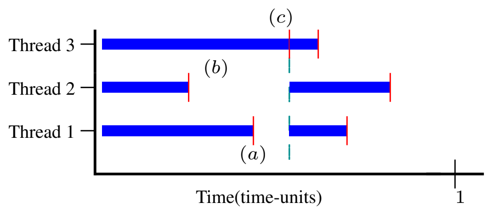

Fig. 2. The synchronization of 3 threads during a single time unit on three CPUs

`THREADS`, the caller is blocked in line 10. The last thread to call the function unblocks all the other threads by making a broadcast signal.

```
1   int uppaal2c_sync_of_threads(void)
      {
2     int error = 0;
3     (*cond_counter)++;
4
5     if (*cond_counter == THREADS) {
6       if ((pthread_cond_broadcast(&
            cond)) != 0) {
7         error = 1;
8       }
9     } else {
10      if ((pthread_cond_wait(&cond,
           &mutex) != 0)) {
11        error = 1;
12      }
13    }
14    pthread_mutex_unlock(&mutex);
15    (*cond_counter) = 0;
16    return error;
17  }
```

Listing 3. Synchronization of thread execution

B. *Declarations*

UPPAAL have two different scopes local and global which is translated likewise, except for clocks and channels, which are practical to keep global. As a consequence we have chosen to prefix all variables with their respective scope-name. We provide an example of this in Listing 4.

```
global_x =
    create_shared_memory_int("
    global_x", sizeof(int));
```

```
  *global_x = 0;
```

Listing 4. Declaration an initialization of a shared clock x

Booleans are translated to integer variables with values 0 representing false and 1 as true. Constants are translated to constant integers in C. The integers is translated directly to integers, but the bounds are discarded. The bounds are discarded because, if the model is verified correctly in UPPAAL the bounds will not be violated under runtime. Finally arrays are translated to arrays of their aforementioned corresponding types, except for arrays of channels which will be explained in Section IV-D.4 on the next page.

C. Expressions

UPPAAL's expressions are similar to C's expressions. In Listing 5, we show the code that converts UPPAAL expressions to C expressions.

```
    string = string.replace("true", "
        1").replace("false", "0")
    string = string.replace("and", "
        &&").replace("or", "").replace
        ("not", "!")
    string = replaceImply(string)
    string = dereferenceVariables(
        environment, ta, string)
    string = replaceMaxMin(string)
    string = string.replace(":=", "="
        )
    string = string.replace("=>", ">=
        ")
```

Listing 5. Conversion of expressions

First we exchange `true` and `false` with 1 and 0 respectively. The logical operators are then replaced with their C counterparts. Then we replace the expression `a imply b` with its equivalent C code `!a || b`. Then we dereference the variables to see which scope they are in. We also need to replace the minimum and maximum expressions `a <? b` and `a >? b` with equivalent C code, so this is done by calling two predefined functions `uppaal2c_min(a,b)` and `uppaal2c_max(a,b)`. The last thing we have to do is replace assignments to C assignments and turn greater than or equal to expressions around.

We will not formally show that the expressions are semantically equivalent, but the reader should be able to convince himself that as long as the functions that replaces `imply` and the min/max functions parse the expression correctly, then the UPPAAL expression and the converted expression are equivalent.

D. Locations in the Generated Code

Each location in an automaton is represented in the `C` code by a `case` in a `switch` statement. This switch construct is nested in a `while` loop which is evaluated once each time unit. The `location` variable holds the current location of the Timed Automaton until a transition is selected. When a transition is selected the `location` variable is set to the target location thus we change the current location.

1) Invariant Generation: The first code generated inside each `case` checks that the invariant is satisfied. We have built the invariant as an `if`-statement, which marks an error and prints a message if the invariant is violated (see Listing 6).

```
  if (!(*global_x <= 7)) {
    error = 1;
    rtl_printf("Invariant is broken
        in state A in %s\n",
        automaton_name);
    break;
  }
```

Listing 6. Code example of a generated invariant

This is a detail, where the implementation made by the tool differs from the semantics, as we only check invariants when we enter a location, as opposed to checking it when selecting a transition. The consequences of this will be discussed in Section VII-A on page 14.

When the invariant has been checked and is not broken, then it is time to actually execute the code that should be performed in the location. Of course we can not generate this code, so we mark the place in the code, thus leaving it up to the user to customize the generated automata.

The user should himself make sure that the customized code does not violate the time bounds of the current time unit, since it will lead to system failure.

2) Transition Generation: The selection of enabled transitions is, like the invariant, generated as an `if`-statement. Each guard is translated to an equivalent expression in `C` as described in Subsection IV-C on the previous page. These guards are written as a logical formula, in such a way that it is only satisfied when all the guards are satisfied. If a guard is satisfied then the transition will be marked as enabled by setting the variable `selected_transition` to the number of the transition we are examining. For each enabled transition detected the `enabled_transitions` variables is incremented by one. The `transition_check` variable is used as

```
  if (*global_x <= 2 && y >= 10 &&
      !*global_b) {
    enabled_transitions++;
    selected_transition = 1;
    transition_check = 1;
  }
```

Listing 7. Code example of a generated selection of a transition

When all transitions have been examined we make a transition iff there is exactly one enabled transition. If there is more than one enabled transition, then the model contains nondeterminism and we note the location, as seen in Listing 8 lines 9-13, and stop the execution by flagging an error. If only one transition is available, we evaluate the `switch`-statement on the `selected_transition` variable to execute the actions associated with the selected transition. Furthermore the `location` variable, which is used in Listing 2 line 3, is set to the target location of the selected transition. When the transition has been activated the `enabled_transitions` variable is reset to zero.

```
1       if (enabled_transitions == 1) {
2         switch (selected_transition) {
3           case 1:
4             *global_x = 0; *global_b =
                1;
5             location = 1;
6             uppaal2c_print_transition("A
                ", name);
7             break;
8         }
9       } else if (enabled_transitions >
            1) {
10        nondeterminism = location;
11        error = 1;
12        break;
13      }
```

Listing 8. Example of activation of a selected transition

3) Channel Synchronization Generation: The channel synchronization code is defined in three different places in the generated code. First we mark the transition, which we want to synchronize and then the actual channel synchronization takes place. The last thing we do is to reset all channel synchronization variables.

The synchronization process is illustrated in Figure 3 on the following page, where Thread 1 and 2 are synchronizing with each other. The marked positions in Figure 3 are:

- a-b. The threads mark their synchronization variables to true, to signal that they are available for synchronization.
- c. The threads are synchronized as explained in Section IV-A.1 on page 6.
- d. The thread is blocked. This is to guarantee the right order of assignments in the synchronization.
- e. The assignments are executed in Thread 2.
- f. Thread 2 sends an unblock signal to Thread 1.
- g. Thread 1 then execute its assignments.
- h-i. The threads sleep until the next time unit and the synchronization variables are reset.

4) The generated synchronization code: The channel synchronization is performed by us-

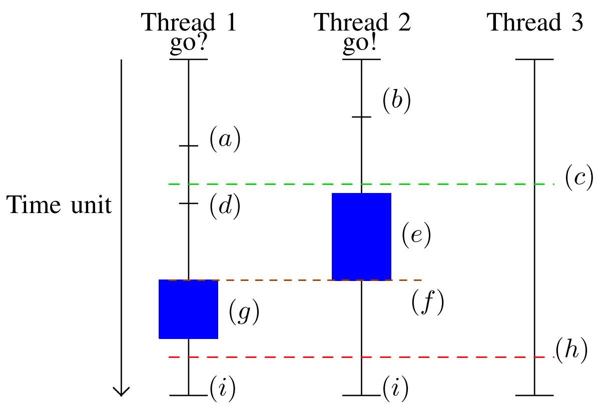

Fig. 3. Synchronization between two automata threads

ing global arrays of integers with values in the range $\{0, 1\}$, which represents `False` or `True`. A single channel is represented by two arrays named `<channel>_emit[]` and `<channel>_receive[]` which represents each end of a channel. The array is then used to indicate that the thread is ready to make a channel synchronization with the other thread. In Listing 9 the `receive` end of the channel is `True` and the transition bookkeeping variables are set to the transitions internal number representation.

```
1    if (*global_x==6) {
2      go_receive[0] = 1;
3      transitions[1] = 1;
4      transition_check = 1;
5    }
```

Listing 9. Set the receiving end of the `go` channel to true

When all transitions have been checked and synchronization channels have been set, the threads are synchronized. This is done to guarantee that all transitions are performed at the same time.

If both threads have their channel synchronization arrays set to true, then the variable `selected_transition` is set to the current transitions value, which was set in Listings 9 line 3. We increment `enabled_transitions`, to count how many outgoing transitions there are.

```
1    if (go_receive[0] && go_emit[0])
       {
2      selected_transition =
          transitions[1];
3      sync_done = 1;
4      blockid = 0;
5      enabled_transitions++;
6    }
```

Listing 10. Check if the marked transition is ready in both ends of the channel

If the synchronization transition was activated and the current thread holds the receiving end of the channel, `<channel>_receive[]`. Then the code in Listing 11 will block the thread until the automaton in the emitting end of the channel unblocks it. This is necessary to ensure that the assignments are executed in the right sequence as the semantics of UPPAAL dictates (see Definition 2 on page 3).

The `block_<channel>` array is used to check if the emitting end of the channel has been executed before the receiving end on a single CPU computer. It is done to guarantee that the thread is not blocked if the emitting end has been executed first on the CPU.

```
if (sync_done && block_go[0]) {
  uppaal2c_execute_order_wait(&
     cond_go[0], &mutex_go[0]);
}
block_go[0] = 1;
```

Listing 11. Block thread if it is the receiving end of the synchronization channel

In the emitting end of a synchronization channel the thread executes its assignments and then sends an unblocking signal to the receiving end of the channel. The variable `blockid`, which was set in Listings 10 line 4, is used as an index in the different arrays in the unblocking process (see Listing 12). The `blockid` variable is set to ensure that the right indexing occurs in the case where the index was a variable. This is done because its value could have been manipulated by the assignments done in the meantime.

```
if (transition_check == 1) {
    if (sync_done) {
      block_go[blockid] = 0;
      uppaal2c_execute_order_signal
          (&cond_go[blockid], &
          mutex_go[blockid]);
    }
  }
```

Listing 12. Unblock the other thread if it is the emitting end of the synchronization channel

The transition is now taken as any other transition, as explained in Listing 8. The first thing to happen the next time unit is that the variables involved in the channel synchronization process are reset to their initial values as shown in Listings 13. This is done in relation to the transitions on which the channel synchronization took place, which is saved in the variable `transition_check`.

```
    if (sync_done &&
        transition_check == 0) {
      go_emit[blockid] = 0;
      blockid = -1;
      sync_done = 0;
  }
```

Listing 13. Reset of channel synchronization variables

V. The Correctness of the Translation

This section proves that the trace of the DTAS and the C implementation of a given model are the same.

The idea is to show that for each state in the trace of an automaton, there is an equivalent state in the trace of the running C-code. This is done by first looking at how the initial state is represented in the C-trace and then demonstrating what would happen in the case of a transition and in the case of a synchronization. We denote the trace of the model as $trace_{TA}$ and the trace of the running C code as $trace_C$ (in accordance with Definition 6 on page 4) and introduce the notion of correspondence between states and traces as follows:

***Definition** 8:* (*Correspondence of states*) A correspondence between two states ($state_z$ and $state_w$) (which might be in two different traces) is written as $state_z \sqsubset state_w$. This means that all information in $state_z$ is also present and identical in $state_w$.

***Definition** 9:* (*Correspondence of traces*) A correspondence is also said to exist between two traces $trace_x$ and $trace_y$ if each state $state^x_n$ in $trace_x$ corresponds to state $state^y_n$ from $trace_y$ and there are no extra states in either of the traces. Correspondence between traces is denoted $trace_x \sqsubset trace_y$.

Our equivalence relation correspondence, $\sqsubset$, is not that different from bisimulation (as defined by [Mil95]) used to show timed language equivalence. As we have a definition of a trace that is different from the usual definition of a trace, we also use another notion of equivalence.

Given these definitions, we show that $trace_{TA} \sqsubset trace_C$. This correspondence between the two traces show that the two systems progresses along the same trace.

A. Initial configurations

As previously described, the state of a system is the set of all current locations C, the set of values for all clocks in the system U and the set of values of all variables in the system V.

The initial state of the system is then $state_0 = \langle C_0, U_0, V_0 \rangle$, where $C = \langle l_{0,A1}, l_{0,A2}, \ldots, l_{0,An} \rangle$ is the initial states of all automata in the system, U_0 is set of all initial clock values and V_0 is set of all initial variable values.

***Theorem** 2:* All initial locations, initial clock values and initial variable values are present in the C code and have correct values. $state^{TA}_0 \sqsubset state^C_0$

Proof Idea: Show that all initial data is correct since the translation algorithm translates correctly.

Proof: In the C code the current locations are always stored in the `location`-variables in each of the threads and as there is a thread for each of the automata, the entire set C is represented. The variable is initialized in the code to the initial location of the automata, so C_0 is also represented in the C-system. The sets V and U is represented as variables shared among the threads (as shown in

Section IV-B on page 7). These variables are initialized to the correct values by the code generation so the sets V_0 and U_0 is also correctly represented. □

B. Transitions

***Theorem** 3:* The generated automata will perform correct transitions from one state to another. If $state_n^{TA} \sqsubset state_n^C$ and there is a state $state_{(n+1)}^{TA}$ in $trace_{TA}$, then there will also be a state $state_{(n+1)}^{C}$ in $trace_C$ such that $state_{(n+1)}^{TA} \sqsubset state_{(n+1)}^{C}$.

PROOF IDEA: First show that guards will only be satisfied in the generated code if they are also satisfied in the model. Then show that synchronizations only happen in the generated code if they also happen in the model. And finally show that updates and resets of clocks are done in accordance with the model.

PROOF:

a) Transitions with guards: Transitions can happen in two ways: The usual transition and transitions with synchronizations attached. In the following we will argue that in both cases the correspondence between the traces is upheld.

A transition can only be taken if its guards are satisfied. The code in line 1 of Listing 7 on page 9 shows an example of one such check in the generated code. Given that $state_n^{TA} \sqsubset state_n^C$ and that we translate the expressions to semantically equivalent C code (confer Section IV-C on page 8), we can guarantee that the generated checks would yield the same result as the guards in the automata. This covers the case of transitions without synchronizations.

b) Transitions with synchronizations: In the case of synchronization we must examine exactly what a synchronization means. The semantics for NTAS's implies that a synchronization is that two automata take a transition only when they both take it and then they take it at the same time. Since the semantics for DTAS's already enforces that all transitions are taken at the same time we only need to show that transitions with synchronizations are only taken when both participating automata are ready. One should be able to convince oneself of that be studying the code and descriptions in Section IV-D.3 on page 9. Since the synchronization relies on both automata setting a shared variable to announce its intention to partake in the synchronization, both automata are required to be ready.

c) Transitions updating the environment: When the automata take transitions, the clocks would increase or be updated and variables would possibly change, as shown in lines 2-7 of Listing 8 on page 9. The resets of clocks and updates of variables is governed by the transitions taken. The transition code is generated to match those of the actions on the edges in the model and, as can be seen in Section IV on page 5, covers all actions on each edge. This completeness ensures that we can change from a $state_n$ to $state_{(n+1)}$ and still have the correspondence between the model and the executing code.

□

C. Correspondence between traces

The final corollary, that the two traces are in fact in correspondence, following from Theorem 2 and 3.

***Corollary** 1:* $trace_{TA} \sqsubset trace_C$

D. Notes

Invariants are not covered in this section, because they potentially can break the correspondence relation. As mentioned we do not (in the tool) check invariants of the target location before taking a transition. Thus taking a transition, which will not satisfy the invariant of the target location, may lead to a deadlock as the system breaks down when invariants are broken. We address this issue further in Section VII.

VI. TEST RESULTS

To test our implementation we have run a number of different models on two different hosts. The goal of these tests are partly to test whether the

Host	Architecture	Speed	Bogomips
1	Pentium 4	2600 MHz	5203.55
2	K6-3	400 MHZ	799.53

TABLE I

SPECIFICATIONS FOR MACHINES USED IN TEST

Model	Aut.	loc.	tra.	clo.	cha.	var.
Fischer	4	16	20	4	0	9
Tutorial	2	8	10	0	0	9
Main−Reset	2	4	4	1	1	0
Pack−Arm	8	30	39	9	7	12

TABLE II

PROPERTIES OF THE TESTED MODELS

Model	LOC
Fischer	926
Tutorial	500
Main−Reset	441
Pack−Arm	2136

TABLE III

LINES OF GENERATED CODE

generated models fail at some point and to provide some benchmarks of the time it takes to execute transitions (without customized code) on different computers running version 3.2 of RTLinux. Table I shows some information on our test machines. Bogomips is the result of an inscientific benchmark calculation that Linux does during boot, the higher the number the more powerful the machine.

The models we tested are listed in Table II along with the number of automata in the systems, as well as the total number of locations, transitions, clocks channels and variables in each. Fischer is an example distributed with UPPAAL, tutorial is the automata that is used in [BDL04] to introduce UPPAAL. The rest of the models have been developed by us and are described further on [JKP].

All these models have been translated, and run on our test machines with a time unit of 1 second. All these models have been run and monitored, and these tests shows that the systems successfully repeats their traces. Afterwards the systems were left running for extended periods of time. And again all were able to keep running without breaking invariants, reporting nondeterminisn or exhibit bad behaviour in any other way.

A. Benchmarks

The first benchmark we make is to count the lines of the generated code. Even though this is not a particularly interesting benchmark it still shows something about the amount of work that can be saved by using our tool. The results are presented in Table III.

The next benchmark (in Table IV) is a measurement of the time it takes to switch states on the different machines. This is more a benchmark of the test machines and we have chosen only to present two models: a very simple one and a more advanced one. All these times is expressed as nanoseconds. For each system on each test host we show:

- The average and maximum time it takes to make a single iteration in each type of automata in the system.
- The average and maximum times it takes to make a transition for the entire system (including the periods of time where the individual threads are sleeping as well as all the bookkeeping overhead).
- The remaining time when the entire system has made one iteration. The last measurement shows the actual upper time bound on the code the user adds into the automata. It should however be noted that this time must be divided amongst all the automata that have custom code.

From the results of the test it can be inferred that even if the model is advanced it can still be executed on slow hardware. While it might seem very impressive that the code for each iteration runs in less than 1 millisecond, it is not that surprising, because all the code does is mainly to evaluate logical formula in if-statements. This low demand for processing power plays well into using the generated code in an embedded environment.

Model		Avg. use	Max. use	Rem. Time
Fischer on Host 1				
	Automaton	512	1568	
	System	44910	48416	$9.999 \cdot 10^8$
Fischer on Host 2				
	Automaton	13221	21184	
	System	819703	829600	$9.992 \cdot 10^8$
Pack−Arm on Host 1				
	Arm	46288	51392	
	Pack	91903	119936	
	Feed	835	2048	
	System	61233	66272	$9.999 \cdot 10^8$
Pack−Arm on Host 2				
	Arm	125339	150016	
	Pack	216927	252160	
	Feed	4955	11360	
	System	361833	440128	$9.996 \cdot 10^8$

TABLE IV

TIME BENCHMARKS FOR GENERATED CODE

VII. RELATED AND FUTURE WORK

In this section we evaluate related work and comtemplate about extending the translation.

Both [Hen] and [AFP+02] are concerned with translating UPPAAL to another language, and both choose the LEGO mindstorm RCX brick as the target platform.

Of these [Hen] is the least interesting piece. Hendriks only able to establish the fact that he cannot relate the implementation formally to the model and does not attempt to make any guarantees as to whether the validated properties of the model is correct in the generated code.

[AFP+02] addresses the elimination of nondeterminism and extends timed automata with a notion of *tasks*. By imposing time bounds on the tasks, they are able to compute scheduling strategies that do not result in deadlocks and thus are able to guarantee which models they can implement.

As for the class of models we can implement given our semantics, we know that as long as the Time Progression Property of Definition 4 on page 3 is not satisfied, then we are able to implement the model. Furthermore we are restricted to implement only models, which does not rely on checking invariants of target locations, by the tool.

Now we give a small elaboration about the UPPAAL constructs that we have omitted from our translation. Some of the features (urgent and committed locations as well as urgent channels) has to do with forcing the model to make transitions without passing time. Those constructs are impossible to retain in our framework.

Another construct that we have not implemented is broadcast synchronizations, even though this would be a rather simple to add in a future version of the tool. Most of the new features that is planned in the next (currently unreleased) version of UPPAAL could also be added to the tool without big changes to the framework.

A. *Tool specific notes*

First we address the fact that our implementation does not check whether invariants will be broken in the target location when deciding if a transition is enabled. The consequence of this is that the implementation may select a transition, which will lead to a deadlock of the system when the invariant is broken.

It should however be noted that it is only a practical limitation, not theoretical. To actually implement a reasonable solution you would have to create a framework that would enable one automaton to be informed about which transitions are to be chosen in the other automata and which assignments may happen as the transitions are taken. This opens up another problem, if every automaton depends on the automata deciding which transition to take, then we would have a circular wait or deadlock. This could, as [AFP+02] suggests be overcome by introducing unique thread priorities and using UPPAAL to calculate evaluation schedules for such situations at compile time. It is however not given that such an evaluation strategy exists, thus the set of models we can implement would again be limited.

Another matter that is worth addressing is the fact that our synchronization is based only on shared variables. If an implementation should be able to use value passing through channels (as has been planned for UPPAAL for some time), then our simple solution will not work. Another solution would be to make use of kernel message passing (FIFO in RTLinux) or actual sockets. While on the subject of network protocols, at the time of writing RTLinux and network traffic makes an unstable platform, so to make use of sockets we would also have to find another platform.

VIII. CONCLUSION

In conclusion our work has contributed to the field by proposing a semantic translation of networks of timed automata to a simplified deterministic network for a class of automata. We have shown a way to implement the simplified models in C and presented the compiler tool `uppaal2c`, which automates the process. Furthermore we have tested and benchmarked implementations of known models. The tests showed that the runtime environment uses a very small fraction of the processing time availiable every time unit.

REFERENCES

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFP$^+$02] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code synthesis for timed automata. *Nordic J. of Computing*, 9(4):269–300, 2002.

[BD] Basic Research in Computer Science at Aalborg University and Department of Information Technology at Uppsala University. Uppaal homepage. http://www.uppaal.com/.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[Hen] M. Hendriks. Translating uppaal to not quite c.

[Inc] Finite State Machine Labs Inc. Rtlinux homepage. http://www.fsmlabs.com/.

[JKP] Arne Mejlholm Jesper Kristensen and Søren Pedersen. Uppaal2c homepage. http://www.cs.aau.dk/˜mejlholm/dat4/.

[Kat99] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik*, chapter 4: Model Checking Real-Time Temporal Logic, pages 202–204. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[TXJS92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[WDR04] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. Almost asap semantics: From timed models to timed implementations. In *HSCC*, pages 296–310, 2004.