

The Tempo Model Checker

User Guide and Reference Manual

May 23, 2010

1 Introduction

Model checking, the problem of deciding whether a correctness property specified in temporal logic holds of a system specification, is a well-established system and software verification technique. Tempo [2] is a formal language for modeling distributed systems with (or without) timing constraints, as collections of interacting state machines called Timed Input/Output Automata (TIOA) [5]. An associated Tempo Toolkit [8] supports several validation methods for systems described using Tempo, including static analysis, simulation, and interactive proof using the PVS theorem prover [7].

The Tempo toolset also supports model checking and this document serves as the User Guide and Reference Manual for TMC, the *Tempo Model Checker*. TMC has been implemented as an Eclipse plug-in for the Tempo toolset, and is based on T2X, a translator from the Tempo specification language to XTA, the input language of the UPPAAL model checker for Timed Automata [6].

T2X utilizes the language-processing front-end of the Tempo toolset as follows. After a Tempo specification is parsed and the corresponding abstract syntax tree (AST) generated by the front-end, T2X processes the AST to generate the corresponding XTA (i.e., Timed Automaton) specification. To carry out the actual model checking, the UPPAAL model checker can then be launched within Eclipse, with the resulting XTA code provided as input.

Tempo is a Turing-complete systems specification language, complete with infinite data types and very general forms of trajectories which describe how the values of state variables evolve over time. It is therefore not amenable to automatic model checking such as the kind UPPAAL provides for Timed Automata. It is therefore necessary to restrict the Tempo language before considering translation into XTA by identifying its most-general finitary (finite-state) core while allowing for real-valued variables that play the role of clocks.

By itself, identifying the most-general finitary core of Tempo is not sufficient for the purposes of translation into XTA. This is because Tempo supports what we call an *absolute-time* specification paradigm, where assignments of clock values are made to clock variables. UPPAAL, being based on the theory of Timed Automata, supports a *relative-time* specification paradigm, where clocks are reset after transitions and then restarted. This time-specification mismatch must be resolved before translation can occur. We refer to the UPPAAL-compliant, finite-state sublanguage of Tempo obtained in this manner as *TEMPAAL*, and it is actually this language that is handled by the T2X translator.

There is also a mismatch between Tempo and UPPAAL concerning the kinds of data structures the two languages support. Readyng a Tempo specification for translation into XTA therefore requires providing a standard, UPPAAL-based implementation of those data structures supported by Tempo but not by UPPAAL. For example, we provide a standard implementation of Tempo (finite) sets as finite arrays in UPPAAL.

The T2X translation scheme is based on the latest version (4.1.0) of UPPAAL, which includes extended support for array variables (quantification and nondeterministic choice over array indices), structures, external functions, and the use of logical implication within state invariants. Consequently, the resulting XTA code is essentially in a 1-to-1 relationship with the original Tempo (*TEMPAAL*) specification, resulting in very short learning curves for experienced Tempo users.

To illustrate the practical utility of the Tempo Model Checker, we have conducted a case study [4] on DHCP Failover (DHCPF), a fault-tolerant version of the original DHCP protocol that uses multiple servers to robustly manage a pool of IP addresses. We have succeeded in applying our Tempo-to-UPPAAL translation technique to a general Tempo specification of DHCPF. In so doing, our model-checking results confirm the proof sketches presented in [3] that DHCPF offers a solution to the timed mutual-exclusion problem of assigning IP addresses in the presence of servers that may fail and recover at any time. In what follows, we use snippets of the Tempo specification of DHCPF to illustrate various aspects of the T2X translation scheme.

The rest of this document develops along the following lines. Section 2 presents an overview of XTA, the input language of the UPPAAL model checker. Section 3 considers the restrictions imposed on Tempo specifications by the T2X translator. Section 4 details our translation scheme from Tempo to XTA. Section 5 provides guidance on how a Tempo specifier can move from an absolute-time specification to a more UPPAAL-friendly relative-time specification. Section 6 discusses function templates, which allow TMC users to specify function definitions in XTA code for imported Tempo (data type) operators. Sections 7 and Section 8 offer TMC installation instructions and a TMC user guide, respectively. Section 9 lists the various examples provided with the current release of the Tempo model checker. Section 10 offers our concluding remarks and directions for future work.

This document, especially Section 4, assumes a basic familiarity with TIOA [5], the formal basis for the Tempo language.

2 UPPAAL and XTA

The Tempo model checker is based on the T2X translator from the Tempo specification language (actually the TEMPAAAL sublanguage of Tempo) to XTA, the input language of the UPPAAL model checker. UPPAAL [6] is a model checker for real-time system based on the theory of Timed Automaton [1], jointly developed by Uppsala and Aalborg Universities. It is designed to verify Timed Automata extended with global shared variables, structured data types, procedure calls, and channel synchronization.

The XTA language, where XTA stands for eXtended Timed Automata, provides a clear-text, human-readable description of a network of Timed Automata. In XTA, an automaton is called a *process*, which may contain automata parameters, variables, in-process function declarations, locations, and transitions. An UPPAAL transition is decorated with a source and destination location, a list of nondeterministic selections (variable assignments), a guard expression, a channel synchronization, and updates to process-private or global variables (with the guard, channel synchronization, and variable updates falling within the scope of the nondeterministic selections). Updates may contain function calls whose definitions are written in a C/C++/Java-like language. An XTA program may have global variables and function declarations which are accessible to all processes. One or more processes make up a system, which is the target of model checking or simulation.

Version 4.1.0 of the UPPAAL toolset supports the following new language features:

- New types, including record, type alias, and meta variable.
- Broadcast channels.
- Comparison between and assignment to arrays.
- User-defined functions.
- For-all and exists quantifiers.
- Nondeterministic selections.
- Urgent and committed locations.

The task of T2X then is, given an arbitrary but finite-state Tempo (i.e. TEMPAAAL) specification, translate it to an equivalent, and by and large equally succinct, UPPAAL specification. This would allow one to model check the Tempo specification via its corresponding humanly-comprehensible UPPAAL specification.

3 Restrictions Imposed on Tempo Specifications by T2X

The T2X translator is implemented for the TEMPAAL sublanguage of Tempo. Consequently, the TMC user typically must rewrite the original Tempo specification before translation into XTA can begin. There are a number of reasons why such a rewriting may be required:

1. The original specification is not a closed system and, as such, the TMC user may need to add an environment automaton to make it closed.
2. The original specification is infinite, containing analog state variables that cannot be processed by the UPPAAL model checker, or containing infinite data structures, such as unbounded queues, stacks, and sets. (A variable v is said to be *analog* in Tempo if v is *Real* and its dynamic type consists of piecewise continuous functions from time intervals to Reals.) Although the translator supports a bounded implementation of such data structures, the user is tasked with ensuring that such bounds are globally not exceeded during model execution. It is therefore always a good choice to avoid the use of infinite data structures.
3. The UPPAAL language has its own language restrictions. For example, a clock can only be assigned or compared with an integer expression; a transition guard must be a conjunction of simple boolean expressions; and a clock cannot be the value of a record field. Therefore, a Tempo specification cannot contain those features whose direct translation will violate any language restrictions imposed by UPPAAL itself.
4. Unlike UPPAAL, which is based on the visual formalism of Timed Automata, Tempo is a textual specification language. It may therefore be the case that for the purposes of increased readability, a Tempo specification contains redundant state variables or inefficient statement sequences. For the purposes of model checking, it may need to be rewritten to improve performance.

In sum, the following restrictions are placed on Tempo specifications by T2X in order for translation into XTA to proceed.

- The following identifiers are reserved keywords in UPPAAL and therefore cannot be used as identifiers in a Tempo specification: `chan`, `clock`, `bool`, `int`, `commit`, `const`, `urgent`, `broadcast`, `init`, `process`, `state`, `guard`, `sync`, `assign`, `system`, `trans`, `deadlock`, `and`, `or`, `not`, `imply`, `for`, `forall`, `exists`, `while`, `do`, `if`, `else`, `return`, `typedef`, `struct`, `rate`, `before_update`, `after_update`, `meta`, `priority`, `progress`, `scalar`, `select`, `void`, `default`, `switch`, `case`, `continue`, `break`.
- All declared state variables can be of the following Tempo types only: *Int*, *Nat*, *Bool*, *Real*, *AugmentedReal*, **Enumeration**, **Array**, **Set**, **Seq**, and **Tuple**. Generic types are not allowed.
- Imported vocabularies may only contain type definitions (and associated operators) involving built-in types **Enumeration**, **Tuple**, **Set**, **Seq**, and **Array**.
- Duplicate identifiers may not appear among the following: enumeration values, type aliases, names of imported operators, names of input/output actions, names of input/output action parameters.
- All variables of type *Real* must be initialized to 0. If a *Real* variable does not have an initial value, it will be initialized to 0 by default.
- Constant real numbers are defined as *Int* in UPPAAL since UPPAAL does not allow variables of type *Real* or *Float*. An UPPAAL clock variable can only be assigned to or compared with an expression involving non-negative integer values.
- T2X only supports tuples of non-clock types. *Real* and *AugmentedReal* (*Real* numbers plus the two additional elements $+\infty$ and $-\infty$) are not allowed in a tuple field.

- T2X only supports sets of finite types, namely, *Bool* and **Enumeration**, and tuples of these two types.
- T2X only supports sequences of non-clock types. *Real* and *AugmentedReal* are not allowed in a sequence field.
- T2X supports the **Null** extension of types *Bool*, **Enumeration** and *Nat* only. (**Null** is a type constructor that, given any type not containing the special value *nil*, produces a new type that is the same as the original, with the addition of *nil*. The *embed* function can be applied to a Null-extended type to return an element of the underlying type.)
- When defining the transitions, trajectories or states of a primitive automaton, the following clauses are ignored: **where**, **let**, **choose**, **initially**, **urgent when**, and **ensuring**. It is the user's responsibility to choose proper initial values for process parameters and state variables.
- **Where**-clauses in composite automaton definitions are not allowed.
- In order for two automata to communicate, the name of an output action in one automaton must be the same as the name of an input action in the other automaton.
- Due to UPPAAL's restriction on transition guards, a transition precondition in Tempo must be a conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. As such, the Or-operator is not allowed in preconditions. The Tempo user, however, can easily decompose a transition having a disjunctive precondition into several transitions whose preconditions are each of those disjunction units, with the transition signature and effect remaining the same. For example: transition

```

internal update
  pre (flag=true /\ next_time=0) \/ (flag=false /\ next_time=u);
  eff ...

```

can be divided into

```

internal update
  pre flag=true /\ next_time=0;
  eff ...

```

and

```

internal update
  pre flag=false /\ next_time=u;
  eff ...

```

- Due to the fact that UPPAAL restricts expressions that check the value of a clock from appearing anywhere along a transition edge, clock comparisons are not allowed in Tempo transition effects. This means in particular that clock comparisons should not be used in if-conditions nor loop-conditions.
- Tempo variables of type *Real* will become clock variables in UPPAAL. Recall that in UPPAAL, variables of type *Real*, other than clock variables, are not supported. All clocks must evolve at the same rate and cannot be temporarily or permanently stopped. A clock cannot be assigned the value of another clock. The only possible operation on a clock during a transition is to reset it to 0. Therefore, the common Tempo usage of a clock, as in the following example, is not accepted by T2X:

```

output send(m)
  pre ~failed /\ now=next_time;
  eff next_time := now + u;

trajdef traj
  stop when ~failed /\ now=next_time;
  evolve d(now) = 1;

```

To solve this problem, we use the so-called *stopwatch* mechanism. See Section 5 for the details.

- Trajectory evolve clauses are limited to $d(\tau)=1$ (time evolves at a constant rate), where the τ in $d(\tau)$ must be an absolute clock variable.
- Task blocks, automaton invariant statements, schedule blocks, and simulation (forward simulation, backward simulation) statements are ignored.
- The Tempo user must define at most one composite automaton which will be considered the **system** of the UPPAAL specification. The only case that does not require the definition of a composite automaton is when the whole Tempo system has only one basic automaton and this automaton is devoid of automaton parameters. In this case, T2X will define the UPPAAL system as the UPPAAL process corresponding to this basic automaton. Furthermore, hidden actions in composite automata are not allowed.

4 Translation Scheme

This section describes the basic translation scheme implemented in T2X. We consider the translation of the following Tempo language elements: types (*Real*, *Bool*, *Int*, *Nat*, **Enumeration**, **Set**, **Seq**), basic TIOA automata (transitions, states, transition effects), TIOA trajectory invariants and **stop when** conditions, and composite TIOA automata. Additionally, every imported operator in a TIOA specification—unless it is a TIOA built-in operator recognized by T2X (see Section 6)—needs a corresponding UPPAAL definition. T2X allows the user to specify the UPPAAL definition of these operators in a C-like grammar. See Section 6 for the details.

4.1 Tempo Type Definitions

The translation scheme for Tempo types is as follows.

- Variables of type *Real* in Tempo become clocks in UPPAAL. Variables of type *Bool* in Tempo become variables of type *Bool* in UPPAAL. *Int* or *Nat* variables are translated to UPPAAL *Int* variables.
- Enumeration names are translated to a type definition for a bounded-integer type. Enumeration values are translated to UPPAAL global constant integers. For example:

```
vocabulary ProgramCounter1
  types Status: Enumeration [idle, begin, wait, critical]
end
```

is translated to

```
typedef int[0,3] Status;
const int idle = 0;
const int begin = 1;
const int wait = 2;
const int critical = 3;
```

- Enumeration variables are translated to UPPAAL bounded-integer variables. For example:

```
mode: Status := idle;
```

is translated to

```
Status mode = idle;
```

- **Tuple** of a certain type is translated to an UPPAAL record over a corresponding type definition. For example:

```
Index: Enumeration [n0, n1, n2, n3],
Edge : Tuple [source:Index, target:Index],
```

is translated to

```
typedef int[0,3] Index;
typedef struct {
  Index source;
  Index target;
} Edge;
```

- **Set** is translated to boolean array. For example:

```
vocabulary ProgramCounter1
  types Clients: Enumeration [c1, c2, c3]
end
live:Set[Clients];
```

is translated to

```
bool live[3];
```

- **Seq** is translated to an UPPAAL record with the following fields: an array of the **Seq** field type; an integer indicating the starting position of the sequence in the array; an integer indicating the position of the end of sequence; and a boolean indicating whether or not the array bound has been exceeded while performing a sequence operation. For example:

```
queue:Seq[Clients];
```

is translated to

```
typedef struct {
  Clients array[10];
  int[0,10] front;
  int[0,10] end;
  bool invalid;
} Seq_Clients;

Seq_Clients queue;
```

Here 10 is the bound on the size of sequence size. The user can set the size via the command-line option “-seqbound=<num>”.

- The **Null** type constructor applied to an enumeration type is translated to the bounded integer `int [-1, 1]`, where 1 is the size of the enumeration type. Special value *nil* is the constant number -1, and *embed(i)* is simply *i*. For example:

```
cur_client:Null[Clients] := nil();
cur_client := embed(c1);
```

is translated to

```
typedef int[-1,2] Null_Clients;
Null_Clients cur_client = nil();
cur_client = c1;
```

4.2 Basic TIOA Automata

A basic TIOA automaton is translated to an UPPAAL process with only one state S , which is also the initial state. All TIOA transitions are translated to UPPAAL transitions over this state.

The translation of basic TIOA automata involves the translation of TIOA input/output transitions, states, and transition effects.

- Input/output transition names are mapped to UPPAAL global broadcast channel variables. If an input/output transition has finite type parameters, the T2X translator will construct a multi-dimensional array of type broadcast channel. The number of finite parameters will determine the array arity (number of dimensions) and the range of each finite parameter serves as the length of the corresponding dimension.

On the other hand, infinite parameters of input/output transition parameters are mapped to UPPAAL global meta variables. Transmission of values for parameters of this kind then becomes a matter of simply writing and reading the corresponding global variable.

To illustrate the translation scheme for input/output transitions, recall (Section 4.1) the 4-valued enumeration type `Index`, and consider a TIOA transition with the signature

```
output send(m:Nat, const i, j:Index)
```

in the automaton

```
automaton BellmanFordRoot(W:WeightedGraph, u:Int, i:Index)
```

Note that this transition has both finite type and infinite type parameters. Translation of a transition of this type will result in the broadcast channel array

```
broadcast chan send[4][4];
```

and, assuming $m=1, i=2, j=3$ when the transition is executed, the UPPAAL transition

```
S->S { guard ...; sync send[2][3]!; assign ..., m=1; },
```

where the ellipses correspond to the translated code for the transition's precondition and effect, respectively (see the translation scheme for transition preconditions and effects just below).

- All TIOA state variables, the valuation of which determine the states of a TIOA automaton, are translated to process variables in UPPAAL.
- A Transition precondition is translated to an UPPAAL expression following the `guard` keyword in the corresponding UPPAAL transition.
- The name of a TIOA input/output transition will appear after the `sync` keyword in the corresponding UPPAAL transition.
- Transition effects are translated to an UPPAAL in-process function and a call to that function after the `assign` keyword in the corresponding UPPAAL transition. For example, the TIOA transition

```
output write_set(id)
pre mode = begin /\ id = pid;
eff mode := wait; time := 0;
```

is translated to

```

void write_set_eff() {
    mode = wait;
    time = 0;
}

```

```

S->S { guard mode==begin&& id==pid; sync write_set!; assign write_set_eff(); },

```

- Transition locals are mapped to local variables in the effect function.

```

internal swap( )
    locals t:Int;
    eff t := a; a := b; b := t;

```

is translated to

```

void swap_eff( ) {
    int t;
    t = a;
    a = b;
    b = t;
}

```

4.3 Trajectory Invariants and Stop-When Conditions

TIOA trajectory invariants and **stop when** conditions are used to define time constraints on UPPAAL state. For example, a trajectory with the simple invariant condition

```

trajdef begin
    invariant mode = begin;
    stop when time = 3;
    evolve d(time) = 1;

```

is translated to

```

S{mode==begin imply time<=3};

```

A trajectory with the slightly more complicated **stop when** condition

```

trajdef normal
    stop when (live[j] /\ j==arr_min(live) /\ last_rectime=delta+e /\ ~elected)
    evolve d(time) = 1;

```

is translated to

```

S{live[j] && j==arr_min(live) && !elected imply last_rectime<=delta+e};

```

A trajectory with an *exists* quantifier in the **stop when** condition

```

trajdef normal
    stop when \E j:Clients (next_status[j]=delta);
    evolve d(time) = 1;

```

is translated to

```

S { forall (j:Clients) next_status[j] <=delta };

```

Note that all clock comparisons in a **stop when** condition, say, $\text{time} = e$, are translated to $\text{time} \leq e$ in UPPAAL state constraints.

4.4 Composite TIOA Automata

A TIOA composite automaton is translated to a number of UPPAAL process instantiations (one per component automaton) and a system definition. An indexed collection of automata in a composite automaton will be flattened. For example, the following composite TIOA automaton:

```
automaton LeaderSystem
  components
    E[j:J]: Elect(j, delta, e, u);
    FD: FailureDetector(delta);
```

will become

```
E_c1 = Elect(c1, delta, e, u);
E_c2 = Elect(c2, delta, e, u);
E_c3 = Elect(c3, delta, e, u);
FD = FailureDetector(delta);
system E_c1, E_c2, E_c3, FD;
```

5 Adaptation to Stopwatches

Most Tempo programs have an absolute timer and several Real variables as time bounds. They use timers in the following way: in a transition, one of the Real variables is set a specific value, usually current time plus a constant value; in a stop when condition, it says the automaton stops at the moment that the absolute clock advances to the value previously set.

Unfortunately this commonly used pattern does not agree with UPPAAL. In UPPAAL clock variables are essentially logic relations. Any effort trying to remember a clock value is not feasible. Thus the absolute clock and time bound usage is not right for UPPAAL. To solve this problem we have to seek the help of relative timers.

A relative timer is a TIOA clock that 1) evolves at a fixed rate; 2) only can be reset to zero; 3) only can be compared to an integer expression. Relative timer can mimic time bound as this way: at every place where the time bound is set to be current time plus a value e , reset the relative timer to be zero; at every place where absolute clock is compared with the time bound, rewrite it to be the comparison between the relative timer and e . The following code is the relative timer adaptation for the snippet in Section 3 where stopwatch is the first time mentioned in this manual:

```
output send(m)
  pre ~failed /\ next_time=u;
  eff next_time := 0;

trajdef traj
  stop when ~failed /\ next_time=u;
  evolve d(now) = 1;
```

One relative timer alone is usually not enough to replace a time bound because TIOA program may set it with different values in different transitions. Therefore when rewriting clock comparisons we need to know which value has been set to the time bound most recently. We introduce an enumeration variable as a flag to indicate which value is set to the time bound previously (can be boolean variable if the bound is set with only two different values). For example, if a time bound `next_send` is set to three different values: `now+e`, `now+u` and `\infty`, the flag shall be set with `plus_e`, `plus_u` and `set_inf`, correspondingly. Therefore the clock expression `now=next_send` shall be rewritten as

```
(next_send_flag=plus_e /\ next_send=e) \/ (next_send_flag=plus_u /\ next_send=u).
```

Note that `now=\infty` will never be satisfied so the `set_inf` case can just be ignored.

A relative timer with its accompanying flag together are called a stopwatch by our definition. With the stopwatch rewriting technique one can always convert a TIOA absolute clock usage to UPPAAL-compliant clock usage. See more stopwatch examples in our adaptation to new user guide examples.

6 Function Templates

Tempo provides support for externally defined operators that can be imported into a Tempo specification. Users can define an operator for any data type in a vocabulary and then import the vocabulary. For example:

```
vocabulary ASC
  types J
  operators min: Set[J] -> J
end

imports ASC
```

Here `J` is a generic type, `min` is an operator that takes a set of `J` and returns a `J`. It is actually only understandable to a human that `min` will return the minimum element in the set; a model checker will not know this unless the user explicitly specifies what transpires during a call to `min`. Many other operators, such as `push_queue`, `append_list`, and `sort_array`, are in the same boat.

To solve this problem, T2X allows users to specify function definitions in XTA code for imported TIOA operators. We refer to such a function definition as a *function template*. All function templates should be written in a text file that shares the same base name with the TIOA file to be checked. Furthermore, the file extension should be `.import`. For example, the name of the file for the function template corresponding to the TIOA source file `leader.tioa` would be `leader.import`. One can also use the “`-fundef=<path>`” option to indicate another non-default template file to be loaded.

The syntax of a function template is:

```
TemplateDecl ::= 'template' ['inline'] ID '<' ParamList '>' '\n'
              XTALines
ParamList    ::=
              | TemplateParam (',' TemplateParam)*
TemplateParam ::= ID '=' TemplateExpr
TemplateExpr  ::= ID '(' Nat ')'
XTALines     ::= XTALine (XTALine)*
```

A function template consists of several lines of text. The first line begins with the keyword `'template'`, with or without a following `'inline'`, then the identifier or function name, followed by a list of template parameters enclosed by a pair of angle brackets. A template parameter consists of three pieces of information: a template parameter name which can be any identifier; a template parameter expression name—one of the six expressions the translator so far supports; and the order of the function parameter which you want the translator to look at. The order is essentially a number starting from zero. Starting from the second line until the end of file or the next `'template'` keyword comes the XTA code for this function. The syntax for XTA functions is C-like. The user can find help in the UPPAAL documentation.

The mechanism of function template is basically just text rewriting. When the translator is invoked it will load built-in templates and also the external template file if it exists. During the translation, when it sees a function call that matches a name in the template list, it will do the following:

1. Get the template parameter lists.
2. For each template parameter, get the template parameter information, which is always a string, and assign it to the template parameter. E.g. “`ty=type(1)`” means user wants to get the type name of the second function parameter. Assume the second function parameter is of TIOA `'Int'` type, then its corresponding UPPAAL type “`int`”, in form of string, will be bound to template parameter `'ty'`.
3. For each template parameter and its value, replace all places in the XTA lines where the template parameter is embraced by a pair of brace brackets with the string value associated with the template parameter. E.g. “`{ty}`” is replaced with “`int`”.

4. The resulted XTA lines are considered the instantiated XTA function code. The translator will add the function declaration to XTA global scope.
5. It is possible that the instantiated function has different function name as the template name. This is considered a name mangling. The original TIOA function call therefore will be translated to a call with the new name, with its actuals remain intact.

Here is a example:

```
// arr_min_index(array)
template arr_min_index<len=len(0)>
int arr_min_index_{len}(bool arr[{len}]) {
    int i = 0;
    while (!arr[i] && i<{len}) i++;
    return i;
}
```

Lines led by double slash `/// are comments lines which will be ignored. The second line template arr_min_index<len=len(0)> indicates this function definition is for TIOA operator arr_min_index. To instantiate this function, the template needs one additional piece of information, which, in this case, is the length of the array parameter. Here len=len(0) means that the translator will get the array length of the first parameter of operator arr_min_index (assuming the first parameter is an array), convert it to a string, and assign it to the template parameter len. To instantiate this template, every place of {len} will be replaced with value of template parameter len. For example, If the translator reads a call to this operator, say arr_min_index(L), where L is an array of length 10, a UPPAAL function`

```
int arr_min_index_10(bool arr[10]) {
    int i = 0;
    while (!arr[i] && i<10) i++;
    return i;
}
```

will be added to UPPAAL global scope and every appearance of `arr_min_index(L)` will be rewritten to `arr_min_index_10(L)`.

There is also so-called *inline* function template. E.g.,

```
// queue_empty(queue)
template inline queue_empty<queue=name(0)>
(queue.end==queue.front)
```

The keyword `inline` indicates the call to operator `queue_empty(queue)` will be translated to a one line UPPAAL expression, which is no more a function call again. E.g., `queue_empty(q)` will be translated to UPPAAL expression `(q.end==q.front)`. Here `queue=name(0)` means template parameter `queue` is the literal name of the first parameter of `queue_empty`. Translating inline functions will not introduce function declarations to UPPAAL model.

One thing notable here is that the translation from TIOA operators to UPPAAL functions actually extends TIOA semantics, since TIOA operators do not allow side effect to their parameters but UPPAAL functions support call-by-reference.

Remember that the translator will not check the syntax of XTA code. It is the user's responsibility to do so.

So far there are six template parameter expressions needed in practice:

- *basetype(i)* : Basetype of *i*-th function parameter, counting from zero. By using this expression one shall expect that this function parameter is an array; if it is not, then just the type of it.
- *type(i)* : Type of *i*-th function parameter, counting from zero. In fact it is same as *basetype(i)*.
- *elementype(i)* : Assuming the *i*-th function parameter is a struct translated from TIOA *Seq*, *elementype* returns the field type of this TIOA sequence.

- *elemLen(i)* : Assuming the *i*-th function parameter is a struct translated from TIOA *Seq*, *elemLen* returns the length of the array field in the struct, namely, the sequence bound that the user set by “-seqbound” option, or 10 by default.
- *len(i)* : Array length of *i*-th function parameter, counting from zero. Obviously the parameter shall be an array.
- *name(i)* : Just the literal name of the *i*-th function parameter, counting from zero. This is usually used in an inline template.

Some widely used Tempo operators are built-in. The user does not need to define them any more. They are:

- *arr_init(L, val)* : Array initialization, all elements in *L* are assigned by value *val*. Note: both these array operators are for one dimensional arrays only.
- *arr_assign(L1, L2)* : Array assignment, copy array *L2* to array *L1*.
- *set_clear(S)* : Set initialization, make it empty. Note: all these set operators are for sets of non-tuple elements. Sets of tuples will result in multi-dimensional arrays. The user shall write her own version of set operators for tuple elements.
- *set_empty(S)* : Return true if *S* is empty, false otherwise.
- *set_size(S)* : Return the size of *S*.
- *set_insert(S, e)* : Insert element *e* into *S*.
- *set_delete(S, e)* : Remove element *e* from *S*.
- *set_union(S1, S2, S3)* : Set union, make *S1* be the result of $S2 \cup S3$.
- *set_intersection(S1, S2, S3)* : Set intersection, make *S1* the result of $S2 \cap S3$.
- *set_diff(S1, S2, S3)* : Set difference, make *S1* the result of $S2 - S3$.
- *set_subset(S1, S2)* : Return true if *S1* is a subset of *S2*, false otherwise.
- *set_supset(S1, S2)* : Return true if *S1* is a super set of *S2*, false otherwise.
- *queue_push_front(Q, e)* : Push *e* to the front of *Q*. Set *Q.invalid* if *Q.array* bound is exceeded.
- *queue_push_end(Q, e)* : Push *e* to the end of *Q*. Set *Q.invalid* if *Q.array* bound is exceeded.
- *queue_pop_front(Q)* : Pop the first element of *Q* if it is not empty. Set *Q.invalid* otherwise.
- *queue_pop_end(Q)* : Pop the last element of *Q* if it is not empty. Set *Q.invalid* otherwise.
- *queue_clear(Q)* : Queue initialization, make it empty.
- *queue_empty(Q)* : Return true if *Q* is empty, false otherwise.
- *queue_length(Q)* : Return the number of elements in *Q*.
- *nil()* : A function that returns constant number -1.
- *embed(i)* : Conversion from a field type to its extension, simply *i*.
- *div(a, b)* : Integer division: a/b .
- *mod(a, b)* : Modulo: $a\%b$.
- *min(a, b)* : Smaller one of two values: $(a<?b)$.
- *max(a, b)* : Greater one of two values: $(a>?b)$.

There is also a special “copy & paste” template which is declared with the special name *__global__*. If the translator imports any template with this special name, it will simply copy the XTA lines of this template to the global scope of the target XTA program. These *__global__* templates are useful for specifying (or instantiating) UPPAAL system parameters. For example,

```

template __global__<>
const WeightedGraph W = {
  {{false, true, true, false},
   {false, false, false, true},
   {false, false, false, true},
   {false, false, false, false}}
};

template __global__<>
const int u = 2;
const int b = 1;
const int e = 3;

```

is used for a system declared as follows:

```

automaton system(W:WeightedGraph, u,b,e:Int) where u>0 /\ b>=0 /\ e>0
components
  BRoot: BellmanFordRoot(W, u, n0);
  BNR1: BellmanFordNonRoot(W u, b, e, n1);
  BNR2: BellmanFordNonRoot(W, u, b, e, n2);
  BNR3: BellmanFordNonRoot(W, u, b, e, n3);
  TC01: TimedChannel2(b, n0, n1);
  TC02: TimedChannel2(b, n0, n2);
  TC13: TimedChannel2(b, n1, n3);
  TC23: TimedChannel2(b, n2, n3);

```

In this way, the user can change the values of system parameters in the template file instead of altering the TIOA file. More importantly, it gives users a way to instantiate generic type parameters such as W . This is not possible in TIOA.

7 UPPAAL installation

The Tempo model checker can invoke the UPPAAL toolset (version 4.0.6 or later) and load in the generated UPPAAL code automatically provided that the UPPAAL toolset is installed on the same logical disk as the Tempo toolset. You can visit <http://www.uppaal.com> to download the UPPAAL toolset, and to obtain installation instructions you might want to follow. Additionally, please remember to add to the PATH system variable the directory where the UPPAAL toolset is extracted.

Installation note: After you unzip UPPAAL on your machine, please ensure that all files in directories `bin-Linux` and `bin-SunOS` have their execution bit set.

8 User Guide

The T2X translator can be invoked from the Tempo toolset using the GUI or via command-line interaction. To run it in command-line mode, go the directory in which `tempo.zip` has been extracted and enter the command:

```
java -jar bin/tempo.jar -states=stages -plugin=uppaal <path_to_tioa.file>
```

There are a number of additional T2X options, available only in command-line mode:

- dumpxta** : Screen-print the XTA file.
- fundef=<path>** : Specify the path to import function templates.
- nouppaal** : Do not invoke the UPPAAL toolset.

- seqbound=<num>** : Set sequence bound.
- arraybound=<num>** : Set array bound.
- bcastintnl** : Broadcast TIOA internal actions as output events. The translator will build an `InternalEventSink` process to watch the broadcasted events. This feature is designed to make simulation in UPPAAL more convenient: the `InternalEventSink` process catches all broadcasted internal actions, giving the user the opportunity to notice them during simulation.
- dumpbuiltin** : Screen-print XTA code of built-in templates.

The following command will screen-print this usage help:

```
java -jar bin/tempo.jar -help -stages=stages -plugin=uppaal
```

To run in GUI mode, load a TIOA file into the current window and then click the "Run uppaal" button in the Tempo toolbar.

You will see that the UPPAAL toolset is invoked and the generated UPPAAL model is input to UPPAAL. Click the *Verifier* tab in the *Query* area, enter a property, which, for example, could be "A [] not deadlock" (meaning the system never deadlocks), then click the *Check* button. You will see "Property is satisfied" in the color green, or "Property is not satisfied" in the color red, in the *Status* text area.

If a property is not satisfied and you would like to see the counter-example, click *Options* in the UPPAAL menu bar, choose *Diagnostic Trace -> Shortest*, and then click *Check*. A window will pop up, saying "Storing the new trace in the simulator will destroy the old one. Continue?" Choose *Yes*. Then click the *Simulator* tab; in the bottom-right area is the message sequence chart (*MSC*) of the shortest counter-example. You can analyze it to determine the scenario that violates the property.

9 Examples

The current release of the Tempo model checker includes the following examples. (The `_r` suffix in the following `tioa` file names indicates that the file has been revised to be made XTA-compliant.)

- **Train**: This example models a train approaching a crossing. Each of the actions takes place with a certain urgency.
- **Door**: This example represents an impatient person ringing a door bell. There are four actions: ringing the bell, door opening for the person to go in, the person leaving, and the person getting impatient while waiting.
- **NodeChannel**: This example consists of a pair of nodes, communicating through two one-way channels.
- **Fischer protocol (buggy version)**: This example models an incorrect version of the Fischer protocol which does not guarantee mutual exclusion.
- **Fischer protocol (correct version)**: This example models the correct Fischer protocol which does guarantee mutual exclusion.
- **TrainCrossing**: This example models trains crossing a gate without collision and starvation.
- **2bitsadder**: This example models a 2-bits adder.
- **leader_r.tioa**: An XTA-compliant version of `leader.tioa` from the book *Tempo Language User Guide and Reference Manual*. This example, which illustrates the usage of sets, models a classic distributed leader-election algorithm, in which several processes attempt to coordinate so that one of them is distinguished as the "leader".
- **leader_arr_r.tioa**: Another rewriting of `leader.tioa` but using arrays as the main data structure.

- `fischerme_r.tioa`: Classic Fischer’s timed mutual-exclusion algorithm. Adapted from `fischerme.tioa` in the new user guide book with little change.
- `timeout_r.tioa`: Rewritten from `timeout.tioa`. This is a simple failure-detection system consisting of three automata: a sender process that sends periodic messages, a detector process that receives these messages and notes when the sender appears to have failed, and a timed channel from the sender to the receiver. This example illustrates the use of the **Seq** data structure.
- `twoTaskRace_r.tioa`: Rewritten from `twoTaskRace.tioa` in the new user guide book. It involves two tasks. One simply sets a boolean flag. The other task periodically increments a counter until the flag is set, then decrements the counter until it reaches zero; at this point, it reports that it is done.
- `bellmanFord_r.tioa`: The timing-based Bellman-Ford shortest-path algorithm. The goal of this system is that each vertex within the weighted directed graph will eventually gain information of a minimum-weight (shortest) path from the root to itself. This example illustrates nearly all features supported by the T2X translator. The generated UPPAAL model is large, normally costing more than three hours to check an invariant property of it.

10 Conclusions

This document has served as a reference manual and user guide for TMC, the Tempo model checker. TMC allows one to perform automatic verification, in the form of temporal logic model checking, on Tempo specifications of distributed systems with (or without) timing constraints. TMC is based on the idea of first translating system specifications written in TAL, the most general UPPAAL-compliant, finite-state sublanguage of Tempo, into XTA, the input language of the UPPAAL model checker for Timed Automata. UPPAAL is then used to perform the actual model checking on the resulting XTA program. The T2X TAL-to-XTA translator can be invoked using the GUI environment or via command-line interaction. Command-line mode provides the user with a number of translation options, ranging from placing bounds on infinite Tempo data structures to screen-printing the XTA code for built-in function templates.

The TMC distribution comes complete with a number of Temp (TIOA) examples on which we have performed T2X-based translation for the purpose of UPPAAL-based model checking. These include the Fischer timed mutual exclusion algorithm and the timing-based Bellman-Ford shortest path algorithm. We are also applying TMC to DHCP Failover (DHCPF), a fault-tolerant version of the original DHCP protocol that uses multiple servers to robustly manage a pool of IP addresses. Doing so requires the use of predicate abstraction and the introduction of “location variables”, which have a natural mapping to UPPAAL (graphical) states. Preliminary results on the DHCPF case study can be found in [4].

References

- [1] R. Alur and D. L. Dill. The theory of timed automata. *TCS*, 126(2), 1994.
- [2] M. Archer, H. Lim, N. Lynch, S. Mitra, and S. Umeno. Specifying and proving properties of Timed I/O Automata using Tempo. *Design Automation for Embedded Systems*, 12(1-2):139–170, June 2008.
- [3] R. Fan, R. Droms, N. Griffeth, and N. A. Lynch. The DHCP failover protocol: A formal perspective, 2007.
- [4] R. Grosu, E. Golub, X. Huang, N. Lynch, and S. A. Smolka. Uppaal-based model checking for TIOA via predicate abstraction. Technical report, Stony Brook University, Department of Computer Science, 2010.
- [5] D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science, Morgan Claypool Publishers, 2006.

- [6] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1:134–152, 1997.
- [7] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, New Jersey, July 1996. Springer-Verlag.
- [8] VeroModo. Tempo toolset v0.2.3, May 2009. <http://www.veromodo.com/tempo>.