Language Reference

This section describes the languages used when defining UPPAAL system models, and requirement specifications.

The System Description section describes the language used when defining a system model.

The Requirements Specification section describes the language used when specifying requirements on the system model.

The Expressions section describes the syntax for expressions in the two languages.

System Description

A system model in UPPAAL consists of a network of processes described as extended timed automata. The description of a model consist of three parts: its global and local declarations, the automata templates, and the system definition.

Declarations

Declarations are either global or local (to a template) and can contain declarations of clocks, bounded integers, channels (although local channels are useless), arrays, records, and types. The syntax is described by the grammar for `Declarations`:

```
Declarations  ::= (VariableDecl | TypeDecl | Function | ChanPriority)*
VariableDecl  ::= Type VariableID (',' VariableID)* ';'
VariableID    ::= ID ArrayDecl* [ '=' Initialiser ]
Initialiser   ::= Expression
                | '{' Initialiser (',' Initialiser)* '}'
TypeDecls     ::= 'typedef' Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
```

The global declarations may also contain at most one channel priority declaration.

Examples

- `const int a = 1;`
  constant `a` with value 1 of type integer.
- `bool b[8], c[4];`
  two boolean arrays `b` and c, with 8 and 4 elements respectively.
- `int[0,100] a=5;`
  an integer variable with the range [0, 100] initialised to 5.
- `int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6} };`
  a multidimensional integer array with default range and an initialiser.

- `clock x, y;`
  two clocks `x` and `y`.
- `chan d;`
  a channel.
- `urgent chan e;`
  an urgent channel.
- `struct { int a; bool b; } s1 = { 2, true };`

  an instantiation of the structure from above where the members `a` and `b` are set to `2` and `true`.
- `meta int swap;`
  `int a;`
  `int b;`
  `assign swap = a; a = b; b = swap;`
  a meta variable is used to swap the contents of two integers.

Type Declarations

The `typedef` keyword is used to name types.

Example

The following declares a record type S containing an integer `a`, a boolean `b` and a clock `c`:

```
typedef struct
{
  int a;
  bool b;
  clock c;
} S;
```

Types

There are 4 predefined types: `int`, `bool`, `clock`, and `chan`. Array and record types can be defined over these and other types.

```
Type          ::= Prefix TypeId
Prefix        ::= 'urgent' | 'broadcast' | 'meta' | 'const'
TypeId        ::= ID | 'int' | 'clock' | 'chan' | 'bool'
                |  'int' '[' Expression ',' Expression ']'
                |  'scalar' '[' Expression ']'
                |  'struct' '{' FieldDecl (FieldDecl)* '}'
FieldDecl     ::= Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
ArrayDecl     ::= '[' Expression ']'
                |  '[' Type ']'
```

The default range of an integer is [-32768, 32767]. Any assignment out of range will cause the

verification to abort.

Variables of type `bool` can have the values `false` and `true`, which are equivalent to the the integer values 0 and 1. Like in C, any non-zero integer value evalutes to true and 0 evaluates to false.

Channels can be declared as urgent and/or broadcast channels. See the section on synchronisations for information on urgent and broadcast channels.

Constants

Integers, booleans, and arrays and records over integers and booleans can be marked constant by prefixing the type with the keyword `const`.

Meta variables

Integers, booleans, and arrays and records over integers and booleans can be marked as meta variables by prefixing the type with the keyword `meta`.

Meta variables are stored in the state vector, but are sematically not considered part of the state. I.e. two states that only differ in meta variables are considered to be equal. Example:

```
const int NUM_EDGES = 42;
meta bool edgeVisited[NUM_EDGES];
```

The example uses meta variables to maintain information about the history of a path, without affecting the state-space exploration. With the declaration above and a system with 42 edges, all with an update assigning `true` to their corresponding entry in `edgeVisited`, the query below can be used to determine if there is a path where all edges are visited (and find such a path, if trace generation is enabled).

```
E<> forall (i : int[0,41]) edgeVisited[i]
```

Arrays

The size of an array is specified either as an integer or as a bounded integer type or scalar set type. In the first case the array will be 0-indexed. In the latter case, the index will be of the given type. The following declares a scalar set `s_t` of size 3 and an integer array `a` of size 3 indexed by the scalar:

```
typedef scalar[3] s_t;
int a[s_t];
```

Record Variables

Record types are specified by using the `struct` keyword, following the C notation. For example, the

record `s` below consist of the two fields `a` and `b`:

```
struct
{
  int a;
  int b;
} s;
```

Scalars

Scalars in UPPAAL are integer like elements with a limitted number of operations: Assignment and identity testing. Only scalars from the same scalar set can be compared.

The limitted number of operations means that scalars are unordered (or that all orders are equivalent in the sense that the model cannot distinguish between any of the them). UPPAAL applies *symmetry reduction* to any model using scalars. Symmetry reduction can lead to dramatic reductions of the state space of the model. resulting in faster verification and less memory being used.

Notice that symmetry reduction is **not** applied if diagnostic trace generation is enabled or when `A<>`, `E[]` or `-->` properties are verified.

Scalar sets are treated as types. New scalar sets are constructed with the `scalar[n]` type constructor, where `n` is an integer indicating the size of the scalar set. Scalars of different scalar sets are incomparable. Use `typedef` to name a scalar set such that is can be used several times, e.g.

```
typedef scalar[3] mySet;
mySet s;
int a[mySet];
```

Here `mySet` is a scalar set of size 3, `s` is a variable whos value belongs to the scalar set `mySet` and `a` is an array of integers indexed by the scalar set `mySet`. Thus `a[s] = 2` is a valid expression.

Functions

Functions can be declared alongside other declarations. The syntax for functions is defined by the grammar for `Function`:

```
Function        ::= Type ID '(' Parameters ')' Block
Block           ::= '{' Declarations Statement* '}'
Statement       ::= Block
                  | ';'
                  |  Expression ';'
                  |  ForLoop
                  |  Iteration
                  |  WhileLoop
                  |  DoWhileLoop
                  |  IfStatement
                  |  ReturnStatement
```

```
ForLoop         ::= 'for' '(' Expression ';' Expression ';' Expression ')'
Statement
Iteration       ::= 'for' '(' ID ':' Type ')' Statement
WhileLoop       ::= 'while' '(' Expression ')' Statement
DoWhile         ::= 'do' Statement 'while' '(' Expression ')' ';'
IfStatment      ::= 'if' '(' Expression ')' Statement [ 'else' Statement ]
ReturnStatement ::= 'return' [ Expression ] ';'
```

Iterators

The keyword `for` has two uses: One is a C/C++/Java like for-loop, and the other is a Java like iterator. The latter is primarily used to iterate over arrays indexed by scalars.

A statement `for (ID : Type) Statement` will execute `Statement` once for each value `ID` of the type `Type`. The scope of `ID` is the inner expression `Expr`, and `Type` must be a bounded integer or a scalar set.

Examples

add

The following function returns the sum of two integers. The arguments are call by value.

```
int add(int a, int b)
{
    return a + b;
}
```

swap

The following procedure swaps the values of two call-by-reference integer parameters.

```
void swap(int &a, int &b)
{
  int c = a;
  a = b;
  b = c;
}
```

initialize

The following procedure initializes an array such that each element contains its index in the array. Notice that the an array parameter is a call-by-value parameter unless an ampersand is used in the declaration. This is different from C++ syntax, where the parameter could be considered an array of references to integer.

```
void initialize(int& a[10])
{
  for (i : int[0,9])
  {
    a[i] = i;
  }
}
```

Templates

UPPAAL provides a rich language for defining templates in the form of extended timed automata. In contrast to classical timed automata, timed automata in UPPAAL can use a rich expression language to test and update clock, variables, record types, call user defined functions, etc.

The automaton of a template consist of Locations and edges. A template may also have local declarations and parameters. A template is instantiated by a process assignment (in the system definition).

Locations

Locations of a timed automaton are graphically represented as circles. If a timed automaton is considered as a directed graph, then locations represent the vertices of this graph. Locations are connected by edges.

Names

Locations can have an optional name. Besides serving as an identifier allowing you to refer to the location from the requirement specification language, named locations are useful when documenting the model. The name must be a valid identifier and location names share the name space with variables, types, templates, etc.

Invariants

Locations are labelled with invariants. Invariants are expressions and thus follow the abstract syntax of expressions. However, the type checker restricts the set of possible expressions allowed in invariants.

An invariant must be a conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. The bound must be given by an integer expression. Furthermore lower bounds on clocks are disallowed. It is important to understand that invariants influence the behaviour of the system -- they are distinctly different from specifying safety properties in the requirements specification language. States which violate the invariants are undefined; by definition, such states do not exist. This influences the interpretation of urgent channels and broadcast

channels. Please see the section on synchronisations for a detailed discussion of this topic.

Examples

The following are valid invariants. Here `x` and `y` are clocks and `i` is an integer array.

- `x <= 2`
  `x` is less than or equal to 2.
- `x < y`
  `x` is (strictly) less than `y`.
- `(i[0]+1) != (i[1]*10)`

Initial locations

Each template must have exactly one initial location. The initial location is marked by a double circle.

Urgent locations

Urgent locations freeze time; *i.e.* time is not allowed to pass when a process is in an urgent location.

Semantically, urgent locations are equivalent to:

- adding an extra clock, `x`, that is reset on every incomming edge, and
- adding an invariant `x <= 0` to the location.

Committed locations

Like urgent locations, committed locations freeze time. Furthermore, if any process is in a committed location, the next transition must involve an edge from one of the committed locations.

Committed locations are useful for creating atomic sequences and for encoding synchronization between more than two components. Notice that if several processes are in a committed location at the same time, then they will interleave.

Edges

Locations are connected by *edges*. Edges are annotated with *selections*, *guards*, *synchronisations* and *updates*.

Selections
: Selections non-deterministically bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding.

Guards

An edge is enabled in a state if and only if the guard evaluates to true.

Synchronisation
: Processes can synchronize over channels. Edges labelled with complementary actions over a common channel synchronise.

Updates
: When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.

Selections

```
SelectList ::= ID ':' Type
             | SelectList ',' ID ':' Type
```

For each `ID` in `SelectList`, bind `ID` non-deterministically to a value of type `Type`. The identifiers are available as variables within the other labels of this edge (guard, synchronization, or update). The supported types are bounded integers and scalar sets.
Note: The identifiers will shadow any variables with the same names.

Example

select: `i : int[0,3]`
synchronization: `a[i]?`
update expression: `receive_a(i)`

This edge will non-deterministically bind `i` to an integer in the range 0 to 3, inclusive. The value `i` is then used both as an array index when deciding what channel to synchronize on, and as an argument in the subsequent call to the function `receive_a`.

Guards

Guards follow the abstract syntax of expressions. However, the type checker restricts the set of possible expressions allowed in guards: A guard must be a conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. The bound must be given by an integer expression.

Examples

- `x >= 1 && x <= 2`
  `x` is in the interval [1,2].
- `x < y`
  `x` is (strictly) less than `y`.
- `(i[0]+1) != (i[1]*10)`

Value at position 0 in an integer array `i` plus one is not equal to value at position 1 times 10 (`i` must be an integer array since we use arithmetic operations on its elements).

Synchronisations

Channels are used to synchronise processes. This is done by annotating edges in the model with *synchronisation labels*. Synchronisation labels are syntactically very simple. They are of the form `e?` or `e!`, where `e` is a side effect free expression evaluating to a channel.

The intuition is that two processes can synchronise on enabled edges annotated with complementary synchronisation labels, *i.e.* two edges in different processes can synchronise if the guards of both edges are satisfied, and they have synchronisation labels `e1?` and `e2!` respectively, where `e1` and `e2` evaluate to the same channel.

When two processes synchronise, both edges are fired at the same time, *i.e.* the current location of both processes is changed. The update expression on an edge synchronizing on `e1!` is executed before the update expression on an edge synchronizing on `e2?`. This is similar to the kind of synchronisation used in CCS or to rendezvous synchronisation in SPIN.

Urgent channels are similar to regular channels, except that it is not possible to delay in the source state if it is possible to trigger a synchronisation over an urgent channel. Notice that clock guards are not allowed on edges synchronising over urgent channels.

Broadcast channels allow 1-to-many synchronisations. The intuition is that an edge with synchronisation label `e!` emits a broadcast on the channel `e` and that any enabled edge with synchronisation label `e?` will synchronise with the emitting process. *I.e.* an edge with an emit-synchronisation on a broadcast channel can always fire (provided that the guard is satisfied), no matter if any receiving edges are enabled. But those receiving edges, which are enabled *will* synchronise. Notice that clock guards are not allowed on edges receiving on a broadcast channel. The update on the emitting edge is executed first. The update on the receiving edges are executed left-to-right in the order the processes are given in the system definition.

Notice that for both urgent and broadcast channels it is important to understand when an edge is enabled. An edge is enabled if the guard is satisfied. **Depending on the invariants, the target state might be undefined.** This does not change the fact that the edges are enabled! *E.g.* when two edges in two different processes synchronise via a broadcast channel, and the invariant of the target location of the receiving edge is violated, then this state is not defined. It is **not** the case that the emitting edge can be fired by itself since the receiving edge is enabled and thus must synchronise. Please see the section about the semantics for further details.

Updates

An *update* is a comma separated list of expressions. These expressions will typically have side effects.

Assignments to clocks are limited to the regular = assignment operator and only integer expressions are allowed on the right hand side of such assignments. The syntax of updates is defined by the grammar for `Update`:

```
Update ::= [Expression (',' Expression)*]
```

Note: Assignments are evaluated sequentially (not concurrently). On synchronizing edges, the assignments on the !-side (the emitting side) are evaluated before the ?-side (the receiving side).

The regular assignment operator, =, can be used for assigning values to integer, boolean, record and clock variables. The other assignment operators are limitted to integer and boolean variables and work as in C, *e.g.* `i += 2` is equivalent to `i = i + 2` except that any side effect of evaluating `i` is only executed once in the first case whereas it is executed twice in the latter case.

Please remember that any integers are bounded. Any attempt to assign a value outside the declared range to an integer, will cause an error and the verification will be aborted.

Examples

- `x = 0`
  clock (or integer variable) `x` is reset.
- `j = ( i[1]>i[2] ? i[1] : i[2] )`
  integer `j` is assigned the maximum value of array elements `i[1]` and `i[2]`. This is equivalent to `j = i[1] >? i[2]`, except that one of the sub-expressions is evaluated twice in the example (once in the condition, and again in either the true case or the false case).
- `x = 1, y=2*x`
  integer variable `x` is set to `1` and `y` to `2` (as assignments are interpreted sequentially).

Parameters

Templates and functions are parameterised. The syntax for parameters is defined by the grammar for `Parameters`:

```
Parameters ::= [ Parameter (',' Parameter)* ]
Parameter  ::= Type [ '&' ] ID ArrayDecl*
```

In contrast to global and local declarations, the parameter list should not be terminated by a semicolon.

Call by Reference and Call by Value

Parameters can be declared to have either call-by-value or call-by-reference semantics. The syntax is taken from C++, where the identifier of a call-by-reference parameter is prefixed with an ampersand in the parameter declaration. Call-by-value parameters are not prefixed with an ampersand.

Clocks and channels must always be reference parameters.

Note: Array parameters must be prefixed with an ampersand to be passed by reference, this does not follow the C semantics.

Examples

- `P(clock &x, bool bit)`
  process template `P` has two parameters: the clock `x` and the boolean variable `bit`.
- `Q(clock &x, clock &y, int i1, int &i2, chan &a, chan &b)`
  process template `Q` has six parameters: two clocks, two integer variables (with default range), and two channels. All parameters except `i1` are reference parameters.

System Definition

In the *system definition*, a system model is defined. Such a model consists of one or more concurrent processes, local and global variables, and channels.

Global variables, channels and functions can be defined in the system definition using the grammar for declarations. Such declarations have a global scope. However, they are not directly accessible by any template, as they are declared after the templates. They are most useful when giving actual arguments to the formal parameters of templates. The declarations in the system definition and in the top-level *declarations* section are part of the system model.

The processes of the system model are defined in the form of a system declaration line, using the grammer for `System` given below. The system line contains a list of templates to be instantiated into processes. Processes can be prioritised as described in the section on priorities.

```
System ::= 'system' ID ((',' | '<') ID)* ';'
```

Templates without parameters are instantiated into exactly one process with the same name as the template. Parameterised templates give rise to one process per combination of arguments, i.e., UPPAAL automatically binds any free template parameters. Any such parameter must be either a call-by-value bounded integer and or a call-by-value scalar. Individual processes can be referenced in expressions using the grammar for `Process` given below. Notice that this is already covered by the grammar for expressions.

```
Process ::= ID '(' Arguments ')'
```

It is often desirable to manually bind some or all formal parameters of a template to actual arguments. This can be done by partial instantiation of templates.

Any progress measures for the model are defined after the system line.

Example

In this example we use the textual syntax for template declaration as used in the XTA format. In the GUI, these templates would be defined graphically.

```
process P()
{
  state s...;
  ...
}

process Q(int[0,3] a)
{
  state t...;
  ...
}

system P, Q;
```

This defines a system consisting of five processes named `P, Q(0), Q(1), Q(2)` and `Q(3)`. Automatic binding of template parameters is very useful in models in which a large number of almost identical processes must be defined, e.g., the nodes of a network in a model of a communication protocol. In order to express that, e.g., all Q processes must be in location s, an expression like `forall (i : int[0,3]) Q(i).s` suffices.

Template Instantiation

New templates can be defined from existing templates using the grammar for `Instantiation`. The new template has the same automaton structure and the same local variables as the template it is defined from. However, arguments are provided for any formal parameters of the template, thus changing the interface of the template.

```
Instantiation ::= ID [ '(' Parameters  ')' ] '=' ID '(' Arguments ')' ';'
```

Template instantiation is most often used to bind formal parameters to actual arguments. The resulting template is later instantiated into a process by listing it in the system line.

The new template can itself be parameterised. This provides the opportunity to make a partial instantiation of a template, where some formal parameters are bound while others remain free. Examples of typical uses are listed below.

For more examples, see the example systems included in the UPPAAL distribution.

Examples

Renaming

```
P1 = Q();
P2 = Q();
system P1, P2;
```

`Q` is a template without any formal parameters. `P1` and `P2` become templates identical to `Q`. This is used to make several instances of `Q` with different names. Notice that `P1=Q()` is a shorthand of `P1()=Q()`.

Binding parameters

In this example we use the textual syntax for template declaration as used in the XTA format. In the GUI, these templates would be defined graphically.

```
process R(int &i, const int j)
{
...
}
int x;
S = R(x, 1);
system S;
```

Here we bind the formal parameters of `R`, `i` and `j`, to `x` and `1` respectively. `S` becomes a template without any parameters. When listed in the system line, `S` is instantiated into a process with the same name.

Partial instantiation

In this example we use the textual syntax for template declaration as used in the XTA format. In the GUI, these templates would be defined graphically.

```
process P(int &x, int y, const int n, const int m)
{
   ...
}

int v, u;
const struct { int a, b, c; } data[2] = { { 1, 2, 3 }, { 4, 5, 6 } };

Q(int &x, const int i) = P(x, data[i].a, data[i].b, 2 * data[i].c);
Q1 = Q(v, 0);
Q2 = Q(u, 1);

system Q1, Q2;
```

Here `P` is a template with four formal parameters integer parameters. The first must be passed by reference, the remaining by value. `Q` is a template with two formal integer parameters. The first must be passed by reference, the second by value. `Q1` is equivalent to `P(v, data[0].a, data[0].b, 2 * data[0].c`.

This is very convenient when defining many instances of the same template with almost the same arguments. It is also useful to bind some formal parameters and leave others free. When the resulting template is listed in the system line, UPPAAL will create a process for each possible combination of arguments to the free parameters.

Progress Measures

A progress measure is an expression that defines progress in the model. It should be weakly monotonically increasing, although occasional decreasses are acceptable. E.g. sequence numbers used in communication protocols might be used to define a progress measure, provided that the sequence number does not overflow to often.

If progress measures are defined, UPPAAL uses the generalized sweepline method to reduce the memory usage. However to be efficient, the domain of a progress measure should not be too large - otherwise performance might degrade significantly.

Progress measures are placed after the system definition. The syntax is defined by the grammar for `ProgressDecl`:

```
ProgressDecl  ::= 'progress' '{' ( Expression ';' )* '}'
```

Examples

```
int i, j, k;

...

progress
{
  i;
  j + k;
}
```

For the above to be a useful progress measure, `i` and `j + k` should increase weakly monotonically.

Priorities

Given some priority order on the transitions, the intuition is that, at a given time-point, a transition is enabled only if no higher priority transition is enabled (see also Semantics.) We say that the higher

priority transition *blocks* the lower priority transition.

Priorities can be assigned to the channels and processes of a system. The priority orders defined in the system are translated into a priority order on tau-transitions and synchronizing transitions. *Delay transitions are still non-deterministic* (unless urgent channels are used.)

Priorities on Channels

```
ChanPriority ::= 'chan' 'priority' (ChanExpr | 'default') ((',' | '<') (ChanExpr |
'default'))* ';'
ChanExpr ::= ID
           | ChanExpr '[' Expression ']'
```

A channel priority declaration can be inserted anywhere in the global declarations section of a system (only one per system). The priority declaration consist of a list of channels, where the '<' separator defines a higher priority level for channels listed on its right side. The `default` priority level is used for all channels that are not mentioned, including tau transitions.

Note: the channels listed in the priority declaration must be declared earlier.

Example

```
chan a,b,c,d[2],e[2];
chan priority a,d[0] < default < b,e;
```

The example assigns the lowest priority to channels `a` and `d[0]`, and the highest priority to channels `b`, `e[0]` and `e[1]`. The default priority level is assigned to channels `c` and `d[1]`.

Priorities on Processes

Process priorities are specified on the system line, using the separator '<' to define a higher priority for processes to its right. If an instance of a template set is listed, all processes in the set will have the same priority.

Example

```
system A < B,C < D;
```

Resolving Synchronization

In a synchronisation the process priorities are ambigous, because more than one process is involved in

such a transition.

When two processes A and B synchronize the priority of the transition is the pair (B,A), where B is the higher priority process. When there are two potential transitions with priorities (B,A) and (D,C), the priorities of B and D are compared. If B > D then (B,A) blocks (D,C), and if D > B then (D,C) blocks (B,A). When B=D we compare the priorities of A and C and resolve the priority of the transitions accordingly.

For a broadcast synchronization the priority is represented as an ordered tuple of descending process priorities, in a similar manner to the pairs used for binary synchronisations.

Priorities on both Channels and Processes

In a system with priorities on both processes and channels, priorities are resolved by comparing priorities on channels first. If they are the same, the process priorities are compared.

Scope Rules

The scope rules determine which element a name refers to in a given context. The context is either local (to a process template), or global (in a system description).

In a local context, the names are always referring to local declarations or formal parameters (if the name is locally defined), otherwise to a globally declared name.

In the global context, a name is always referring to a global declaration.

Note: There is only one name space in each context. This means that in each context all declared clocks, integer variables, constants, locations, and formal paramters must have unique names. However, local names may shadow globally declared names.

Semantics

In the following we give a pseudo-formal semantics for UPPAAL. The semantics defines a timed transition system (*S, s0,* ->) describing the behaviour of a network of extended timed automata. The set of states *S* is defined as {(*L, v*) | *v satisfies Inv*(*L*)}, where *L* is a location vector, *v* is a function (called a *valuation*) mapping integer variables and clocks to their values, and *Inv* is a function mapping locations and location vectors to invariants. The initial state *s0* is the state where all processes are in the initial location, all variables have their initial value, and all clocks are zero. The transition relation, ->, contains two kinds of transitions: delay transitions and action transitions. We will describe each type below.

Given a valuation *v* and an expression *e*, we say that *v* satifies *e* if *e* evaluates to non-zero for the given valuation *v*.

Invalid Evaluations

If during a successor computation any expression evaluation is invalid (consult the section on expressions for further details about invalid evaluations), the verification is aborted.

Delay Transitions

Delay transitions model the passing of time without changing the current location. We have a delay transition (L, v) --(d)--> (L, v'), where d is a non-negative real, if and only if:

- $v' = v + d$, where $v+d$ is obtained by incrementing all clocks with d.
- for all $0 <= d' <= d$: $v + d'$ *satisfies* $Inv(L)$
- L contains neither committed nor urgent locations
- for all locations l in L and for all locations l' (not necessarily in L), if there is an edge from l to l' then either:
    - this edge does not synchronise over an urgent channel, or
- this edge does synchronise over an urgent channel, but for all $0 <= d' <= d$ we have that $v + d'$ does not satisfy the guard of the edge.

Action Transitions

For action transtions, the synchronisation label of edges is important. Since UPPAAL supports arrays of channels, we have that the label contains an expression evaluating to a channel. The concrete channel depends on the current valuation. To avoid cluttering the semantics we make the simplifying assumption that each synchronisation label refers to a channel directly.

Priorities increase the determinism of a system by letting a high priority action transition block a lower priority action transition. Note that delay transitions can never be blocked, and no action transition can be blocked by a delay transition.

For action transitions, there are three cases: Internal transitions, binary synchronisations and broadcast synchronisations. Each will be described in the following.

Internal Transitions

We have a transition (L, v) --*--> (L', v') if there is an edge $e=(l,l')$ such that:

- there is no synchronisation label on e
- v satisfies the guard of e
- $L' = L[l'/l]$
- v' is obtained from v by executing the update label given on e
- v' satisfies $Inv(L')$
- Either l is committed or no other location in L is committed.

- There is no action transition from (L, v) with a strictly higher priority.

Binary Synchronisations

We have a transition (L, v) --*--> (L', v') if there are two edges $e1$=($l1$,$l1'$) and $e2$=($l2$,$l2'$) in two different processes such that:

- $e1$ has a synchronisation label $c!$ and $e2$ has a synchronisation label $c?$, where c is a binary channel.
- v satisfies the guards of $e1$ and $e2$.
- L' = L[$l1'$/$l1$, $l2'$/$l2$]
- v' is obtained from v by first executing the update label given on $e1$ and then the update label given on $e2$.
- v' satisfies Inv(L')
- Either
    - $l1$ or $l2$ or both locations are committed, or
- no other location in L is committed.
- There is no action transition from (L, v) with a strictly higher priority.

Broadcast Synchronisations

Assume an order $p1$, $p2$, ... pn of processes given by the order of the processes in the system declaration statement. We have a transition (L, v) --*--> (L', v') if there is an edge e=(l,l') and m edges ei=(li,li') for 1<=i<=m such that:

- Edges e, $e1$, $e2$, ..., em are in different processes.
- $e1$, $e2$, ... em are ordered according to the process ordering $p1$, $p2$,... pn.
- e has a synchronisation label $c!$ and $e1$, $e2$, ... em have synchronisation labels $c?$, where c is a broadcast channel.
- v satisfies the guards of e, $e1$, $e2$, ... em.
- For all locations l in L not a source of one of the edges e, $e1$, $e2$, ... em, all edges from l either do not have a synchronisation label $c?$ or v does not satisfy the guard on the edge.
- L' = L[l'/l, $l1'$/$l1$, $l2'$/$l2$, ... lm'/lm]
- v' is obtained from v by first executing the update label given on e and then the update labels given on ei for increasing order of i.
- v' satisfies Inv(L')
- Either
    - one or more of the locations l, $l1$, $l2$, ... lm are committed, or
- no other location in L is committed.
- There is no action transition from (L, v) with a strictly higher priority.

Requirements

In this help section we give a BNF-grammar for the requirement specification language used in the verifier of UPPAAL.

```
Prop ::= 'A[]' Expression | 'E<>' Expression | 'E[]' Expression
      | A<> Expression | Expression --> Expression
```

All expressions must be side effect free. It is possible to test whether a certain process is in a given location using expressions on the form `process.location`.
See also: Semantics of the Requirement Specification Language

Examples

- `A[] 1<2`
  invariantly 1<2.
- `E<> p1.cs and p2.cs`
  true if the system can reach a state where both process `p1` and p2 are in their locations `cs`.
- `A[] p1.cs imply not p2.cs`
  invariantly process `p1` in location `cs` implies that process `p2` is **not** in location `cs`.
- `A[] not deadlock`
  invariantly the process is not deadlocked.

Semantics of the Requirement Specification Language

In the following we give a pseudo-formal semantics for the requirement specification language of UPPAAL. We assume the existens of a timed transition system (*S, s0,* ->) as defined in the semantics of UPPAAL. In the following, `p` and `q` are state properties for which we define the following temporal properties:

Possibly

The property `E<> p` evaluates to true for a timed transition system if and only if there is a sequence of alternating delay transitions and action transitions *s0*-->*s1*-->...-->*sn*, where *s0* is the initial state and *sn* satisfies *p*.

Invariantly

The property `A[] p` evaluates to true if (and only if) every reachable state satisfy `p`.

An *invariantly* property `A[] p` can be expressed as the *possibly* property `not E<> not p`.

Potentially always

The property `E[] p` evaluates to true for a timed transition system if and only if there is a sequence of alternating delay or action transitions *s0*-->*s1*-->...-->*si*-->... for which *p* holds in all states *si* and which either:

- is infinite, or
- ends in a state (*Ln, vn*) such that either
    - for all *d*: (*Ln, vn* + *d*) satisfies *p* and *Inv*(*Ln*), or
- there is no outgoing transition from (*Ln, vn*)

Eventually

The property `A<> p` evaluates to true if (and only if) all possible transition sequences eventually reaches a state satisfying `p`.

An *eventually* property `A<> p` can be expressed as the *potentially* property `not E[] not p`.

Leads To

The syntax `p --> q` denotes a leads to property meaning that whenever `p` holds eventually `q` will hold as well. Since UPPAAL uses timed automata as the input model, this has to be interpreted not only over action transitions but also over delay transitions.

A *leads to* property `p --> q` can be expressed as the property `A[] (p imply A<> q)`.

State Properties

Any side-effect free expression is a valid state property. In addition it is possible to test whether a process is in a particular location and whether a state is a deadlock. State proprerties are evaluated for the initial state and after each transition. This means for example that a property `A[] i != 1` might be satisfied even if the value of `i` becomes 1 momentarily during the evaluation of initializers or update-expressions on edges.

Locations

Expressions on the form `P.l`, where `P` is a process and `l` is a location, evaluate to true in a state (*L, v*) if and only if *P.l* is in *L*.

Deadlocks

The state property `deadlock` evaluates to true for a state (*L, v*) if and only if for all *d* >= *0* there is no

action successor of $(L, v + d)$.

Property Equivalences

The UPPAAL requirement specification language supports five types of properties, which can be reduced to two types as illustrated by the following table.

Name	Property	Equivalent to
Possibly	`E<> p`	
Invariantly	`A[] p`	`not E<> not p`
Potentially always	`E[] p`	
Eventually	`A<> p`	`not E[] not p`
Leads to	`p --> q`	`A[] (p imply A<> q)`

Expressions

Most of the expression syntax of UPPAAL coincides with that of C, C++ and Java. E.g. assignments are done using the '=' operator (the older ':=' still works, but '=' is preferred). Notice that assignments are them self expressions.

The syntax of expressions is defined by the grammar for `Expression`.

```
Expression ::= ID
             |  NAT
             |  Expression '[' Expression ']'
             |  '(' Expression ')'
             |  Expression '++' | '++' Expression
             |  Expression '--' | '--' Expression
             |  Expression Assign Expression
             |  Unary Expression
             |  Expression Binary Expression
             |  Expression '?' Expression ':' Expression
             |  Expression '.' ID
             |  Expression '(' Arguments ')'
             |  'forall' '(' ID ':' Type ')' Expression
             |  'exists' '(' ID ':' Type ')' Expression
             |  'deadlock' | 'true' | 'false'

Arguments  ::= [ Expression ( ',' Expression )* ]

Assign     ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
             | '|=' | '&=' | '^=' | '<<=' | '>>='
Unary      ::= '+' | '-' | '!' | 'not'
Binary     ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
             |  '+' | '-' | '*' | '/' | '%' | '&'
             |  '|' | '^' | '<<' | '>>' | '&&' | '||'
```

```
|  '<?' | '>?' | 'or' | 'and' | 'imply'
```

Like in C++, assignment, preincrement and predecrement expressions evaluate to references to the first operand. The inline-if operator does in some cases (*e.g.* when both the true and false operands evaluate to compatible references) also evaluate to a reference, *i.e.*, it is possible to use an inline-if on the left hand side of an assignment.

The use of the `deadlock` keyword is restricted to the requirement specification language.

Boolean Values

Boolean values are type compatible with integers. An integer value of 0 (zero) is evaluated to false and any other integer value is evaluated to true. The boolean value `true` evaluates to the integer value 1 and the boolean value `false` evaluates to the integer value 0. **Notice:** A comparison like `5 == true` evaluates to false, since `true` evaluates to the integer value 1. This is consistent with C++.

Precedence

UPPAAL operators have the following associativity and precedence, listed from the highest to lowest. Operators borrowed from C keep the same precedence relationship with each other.

left	() [] .
right	! ++ -- unary -
left	* / %
left	- +
left	<< >>
left	<? >?
left	< <= >= >
left	== !=
left	&
left	^
left	|
left	&&
left	||
right	?:
right	= := += -= *= /= %= &= |= <<= >>= ^=
right	not
left	and
left	or imply

left	forall exists

Operators

Anybody familiar with the operators in C, C++, Java or Perl should immediately feel comfortable with the operators in UPPAAL. Here we summarise the meaning of each operator.

()	Parenthesis alter the evaluation order
[]	Array lookup
.	Infix lookup operator to access process scope
!	Logical negation
++	Increment (can be used as both prefix and postfix operator)
--	Decrement (can be used as both prefix and --> --postfix operator)
-	Integer subtraction (can also be used as unary negation)
+	Integer addition
*	Integer multiplication
/	Integer division
%	Modulo
<<	Left bitshift
>>	Right bitshift
<?	Minimum
>?	Maximum
<	Less than
<=	Less than or equal to
==	Equality operator
!=	Inequality operator
>=	Greater than or equal to
>	Greater than
&	Bitwise and
^	Bitwise xor
|	Bitwise or
&&	Logical and
||	Logical or
?:	If-then-else operator
not	Logical negation
and	Logical and

or	Logical or
imply	Logical implication
forall	Forall quantifier
exists	Exists quantifier

Notice that the keywords `not`, `and` and `or` behave the same as the `!`, `&&`, and `||` operators, except that the former have lower precedence.

Expressions Involving Clocks

When involving clocks, the actual expression syntax is restricted by the type checker. Expressions involving clocks are divided into three categories: *Invariants*, *guards*, and *constraints*:

- An invariant is a conjunction of upper bounds on clocks and differences between clocks, where the bound is given by an integer expression.
- A guard is a conjunction of bounds (both upper and lower) on clocks and differences between clocks, where the bound is given by an integer expression.
- A constraint is any boolean combination (involving negation, conjunction, disjunction and implication) of bounds on clocks and differences between clocks, where the bound is given by an integer expression.

In addition, any of the three expressions can contain expressions (including disjunctions) over integers, as long as invariants and guards are still conjunctions at the top-level. The full constraint language is only allowed in the requirement specification language.

Out of Range Errors and Invalid Evaluations

An evaluation of an expression is *invalid* if out of range errors occur during evalution. This happens in the following situations:

- Division by zero.
- Shift operation with negative count.
- Out of range assignment.
- Out of range array index.
- Assignment of a negative value to a clock.
- Function calls with out of range arguments.
- Function calls with out of range return values.

In case an invalid evaluation occurs during the computation of a successor, *i.e.*, in the evaluation of a guard, synchronisation, assignment, or invariant, then the verification is aborted.

Quantifiers

An expression `forall (ID : Type) Expr` evaluates to true if `Expr` evaluates to true for all values `ID` of the type `Type`. An expression `exists (ID : Type) Expr` evaluates to true if `Expr` evaluates to true for some value `ID` of the type `Type`. In both cases, the scope of `ID` is the inner expression `Expr`, and `Type` must be a bounded integer or a scalar set.

Example

The following function can be used to check if all elements of the boolean array `a` have the value `true`.

```
bool alltrue(bool a[5])
{
  return forall (i : int[0,4]) a[i];
}
```

Identifiers

The valid identifier names are described by the following regular expression: `[a-zA-Z_]([a-zA-Z0-9_])*`

Examples

- `a, B, c2, d2`
  valid identifier names.
- `1, 2a, 3B, 4c5`
  invalid identifier names.

Reserved Keywords

The reserved keywords that should not be used as identifier names when defining systems are: `chan`, `clock`, `bool`, `int`, `commit`, `const`, `urgent`, `broadcast`, `init`, `process`, `state`, `guard`, `sync`, `assign`, `system`, `trans`, `deadlock`, `and`, `or`, `not`, `imply`, `true`, `false`, `for`, `forall`, `exists`, `while`, `do`, `if`, `else`, `return`, `typedef`, `struct`, `rate`, `before_update`, `after_update`, `meta`, `priority`, `progress`, `scalar`, `select`, `void`, `default`.

The following keywords are reserved for future use: `switch`, `case`, `continue`, `break`.