Tool support for verifying trace inclusion with Uppaal

T. Bourke
tbourke@cse.unsw.edu.au
NICTA and University of New South Wales

A. Sowmya
sowmya@cse.unsw.edu.au
University of New South Wales

Technical Report
UNSW-CSE-TR-0723
December 2007

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Trace inclusion against a deterministic Timed Automata can be verified with the Uppaal model checking tool by constructing a test automaton that traps illegal synchronisation possibilities. Constructing the automaton manually is tedious and error prone. This paper presents a tool that does it automatically for a subset of Uppaal models.

Certain features of Uppaal, namely selection bindings and channel arrays, complicate the construction. We first formalise these features, and then show how to incorporate them directly in the testing construction. To do so we limit the forms of subscript that can be used to specify synchronisations; striving for a balance between practicability and program complexity. Unfortunately, some combinations of selection bindings and universal quantifiers cannot be effectively manipulated. The tool does not yet validate the determinism requirements, nor handle committed states or broadcast channels.

1 Introduction

Timed automata are a formalism for modelling discrete-event systems where it is not only important *which* events occur, but also *when* they occur. A dense time domain is assumed, allowing natural descriptions of many *real-time* systems. Timing restrictions and behaviours are modelled using clocks. Operations and constraints on clocks are limited so as to make automatic model checking practicable.

Uppaal [LPY97] is a software tool for creating timed automata models and verifying certain of their properties by exhaustive state space exploration. Its expressive description language is one of its strengths, but also presents a challenge to tools that would automatically manipulate models—one such tool is described by this report.

Given two models expressed as timed automata, it is sometimes useful to ask whether the behaviours of one are, in some sense, permitted by the other. For example, such a relationship between models is interesting when verifying a model of an implementation against a more abstract specification, or when creating successive abstractions to make model-checking feasible [JLS00, Sto02]. Many such refinement relationships are possible; this report focuses on trace inclusion which is both adequate and natural for a large class of systems [KLSV06].

Trace inclusion is closely related to language inclusion. Whilst deciding language inclusion between two timed automata is undecidable in general [AD94, Corollary 5.3], it is PSPACE-complete if the containing automaton is deterministic [AD94, Theorem 6.6].

Checking whether the set of traces of one timed automata $\mathcal{A}_1$ is included within that of another automaton $\mathcal{A}_2$, that is whether $\mathcal{A}_1 \sqsubseteq_{\mathrm{TR}} \mathcal{A}_2$, can be performed by constructing a test automaton $\mathcal{A}_2^{\mathrm{Err}}$ and then performing a reachability analysis of the composition $\mathcal{A}_1 \parallel \mathcal{A}_2^{\mathrm{Err}}$ [JLS00, Sto02].

Manually constructing $\mathcal{A}^{\mathrm{Err}}$ can be tedious and error prone. For this reason we have developed a tool to do it automatically for a subset of Uppaal models. Automation is particularly beneficial because the insights gained by experimenting with trace inclusion can lead to improvements in specifications which in turn require updated testing constructions and further verification runs. Making this faster and more convenient increases the practical effectiveness of trace inclusion techniques.

Several features of Uppaal facilitate the creation of succinct models. While these features do not increase fundamental expressiveness, they do present a challenge to tools that would incorporate them into model transformations, such as the testing construction. We take up this challenge in §3.

By first formalising Uppaal models in a more concrete way than usual, as done in §2, we gain a means of describing and understanding the extended modelling constructs. The chief construct is termed a process, written $\mathcal{P}$, and we focus on generating $\mathcal{P}^{\mathrm{Err}}$ correctly with respect to underlying automata $\mathcal{A}$ and $\mathcal{A}^{\mathrm{Err}}$.

In §4 we briefly describe our implementation of the extended testing construction. Many parts of the program are more general in nature: model parsing and layout algorithms, for instance. They are easily adapted to serve other types of model transformation and generation, some of which have been incorporated into the tool and are accessed through a simple expression language.

2 Uppaal

Uppaal [LPY97] comprises three main components: a graphical user interface for specifying networks of timed automata using diagrams and a C-like description language, an interface for running interactive simulations and viewing traces, and, a model checking engine for deciding whether a model satisfies given formulas. The components utilise a library for parsing XML input and type-checking the description language.

In this section we formalise enough of the Uppaal modelling language to describe and justify the techniques of §3. We relate these details to those of existing semantic accounts. Uppaal models are constructed as networks of communicating processes using the description language to express guards, invariants, and actions. It is necessary to formalise these *processes* in a fairly concrete way for later constructions, and their limitations, to make sense.

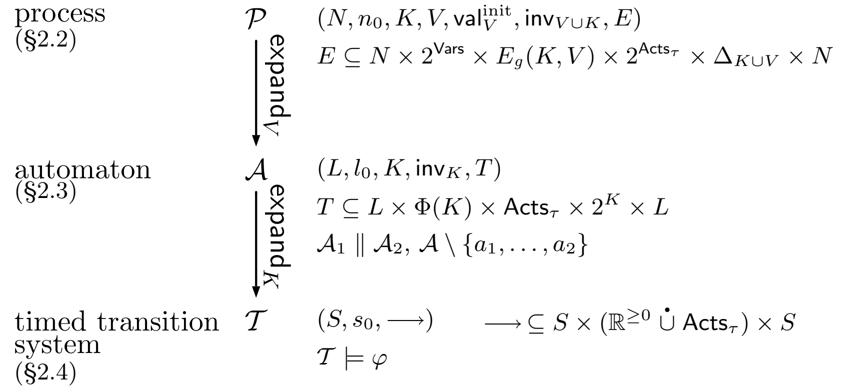

process
(§2.2)
$\mathcal{P}$
$(N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V\cup K}, E)$
$E \subseteq N \times 2^{\mathsf{Vars}} \times E_g(K,V) \times 2^{\mathsf{Acts}_\tau} \times \Delta_{K\cup V} \times N$
expand_V
automaton
(§2.3)
$\mathcal{A}$
$(L, l_0, K, \mathsf{inv}_K, T)$
$T \subseteq L \times \Phi(K) \times \mathsf{Acts}_\tau \times 2^K \times L$
$\mathcal{A}_1 \parallel \mathcal{A}_2,\ \mathcal{A} \setminus \{a_1, \ldots, a_2\}$
expand_K
timed transition system
(§2.4)
$\mathcal{T}$
$(S, s_0, \longrightarrow)$
$\longrightarrow \subseteq S \times (\mathbb{R}^{\geq 0} \dot{\cup} \mathsf{Acts}_\tau) \times S$
$\mathcal{T} \models \varphi$

Figure 2.1: Overview of the various timed transition systems and operations

Figure 2.1 shows the three major constructions that are required and some relevant interrelationships. The larger tuples toward the top make modelling more convenient, while smaller tuples at bottom are easier to understand and reason about.

Processes are the topmost structure in the figure. They formalise process models as created directly within Uppaal and transformed by our implementation. The details of template instantiation and priority specification are omitted. We are not aware of any existing formalisations of such 'macro features' in Uppaal.

Processes, and specifically their non-clock variables, are *expanded* into the *timed (safety) automata* [AD94, BY04, HNSY94], or just *automata* with which most accounts of Uppaal begin. This expansion expand_V formalises the meaning of non-clock variables, selection bindings, and channel arrays, and allows us to adopt the standard definitions for various properties and constructions.

Interpreting timed automata as *open systems* leads to a definition of traces and refinement [Sto02, §7.5.2]. We adopt *parallel composition* and *restriction* operators that are closed over timed automata, and facilitate a *closed system* interpretation of Uppaal networks [Sto02, §7.5.2].

Closed networks are represented as *timed transition systems* (TTS), which are suitable for simulation and model-checking, and also provide the ultimate semantic reference for the other structures. The function expand_K expands the constraints expressed by clocks into delay and action transitions over a more abstract state space.

Several features of Uppaal are excluded from our descriptions.

- In *urgent nodes* delay is forbidden. The effect is simulated by adding a fresh clock x, that is reset on every edge, and then augmenting the invariants of urgent nodes with the constraint $x \leq 0$. The implementation does this automatically.

- Enabled communications on *urgent channels* occur as soon as possible, taking priority over delays.

- In addition to channel synchronisations, processes may communicate with one another through *global variables.*

- The outgoing edges of a *committed node* have priority over those from non-committed nodes. This feature can be used to model atomicity of action sequences.

- An output action on a *broadcast channel* may synchronise with all enabled input actions on the same channel, or, if there are none, occur alone.

Committed nodes and broadcast channels are not further discussed. They are not supported by the current implementation. Urgent channels and global variables are not formalised, but the extended techniques apply directly and the tool implements the necessary additional constructions [JLS00], which are outlined in §3.3.

2.1 Preliminaries

Variables, expressions, and channel arrays add much to the practical utility of Uppaal. They also present challenges to tools that would automatically manipulate models. In this section we introduce the minimum of notation required in the rest of the paper. The material in §2.1 is the most critical, particularly the description of channel sets. We aim to formalise just enough of the essential intuitions to make later descriptions both precise and manageable. Familiarity with Uppaal will greatly aid understanding the definitions.

Variables

Variables increase modelling convenience. Uppaal has both data and clock variables. Data variables facilitate concise and flexible process models. They are eliminated from processes in the expansion to automata. Clock variables are used to model timing characteristics. They effectively specify which delay transitions may occur in an underlying timed transition system.

Although Uppaal variables are typed, we need only make a minimum of distinctions. Clock and channel variables are treated as distinct sets. Boolean variables, record types, and arrays of anything but channels are, without loss of generality, ignored. The set of all variables Vars encompasses the two categories of types that remain:

- Bounded sets of integers $\mathbb{Z}_{[l,u]}$ with $l \leq u$, and where $\forall j \in \mathbb{Z}.\, l \leq j \leq u$ implies $j \in \mathbb{Z}_{[l,u]}$, and,
- Finite *scalar sets* $\mathbb{S}_s$, each identified by an index s from the set Scalar.

Variables of scalar or bounded integer types may index arrays. They may be bound by quantifiers in expressions and as selection variables over edges in processes.

Elements of scalar sets may only be assigned to variables or compared for equality. These restrictions allow model-checking to be optimised in certain ways (by symmetry reduction). Each scalar set is disjoint from every other. Each declaration within Uppaal, for example `scalar[5] ids`, introduces a new set/index. Type aliases, stated using `typedef`, preserve set identity.

Expressions, valuations, and updates

Expressions in Uppaal are built from variables, constants, function calls, operators (including a conditional operator), relations, and quantifiers. We treat expressions abstractly as members of the set Exprs. We write $\mathsf{freevars}(e)$ for the set of unbound variables in $e \in \mathsf{Exprs}$.

An expression denotes either a truth value, an integer, or a value from a scalar set. In the latter case, the only possible expression is a single variable.

The value of an expression e depends on the values of variables in $\mathsf{freevars}(e)$, which are formulated as *valuations.*

A valuation val_V is given with respect to a finite set V of variables. For each $v \in V$, $\mathsf{val}_V(v)$ is a value of appropriate type. The set of all valuations for a given set of variables V is written Vals_V. Valuations may be composed,

$$\mathsf{val}_{V_1} \triangleright \mathsf{val}_{V_2}(v) \quad = \quad \begin{cases} \mathsf{val}_{V_2}(v) & \text{if } v \in V_2 \\ \mathsf{val}_{V_1}(v) & \text{otherwise} \end{cases}$$

For valuations over a set of clock variables K[†], we write val^0_K for the valuation that maps each $k \in K$ to 0, and val^{+d}_K for the valuation mapping each $k \in K$ to $\mathsf{val}_K(k) + d$.

The value of expression e with respect to valuation val_V is written $[\![e]\!]_{\mathsf{val}_V}$, provided $\mathsf{freevars}(E) \subseteq V$. A partial evaluation $\lceil\!\lceil e \rceil\!\rceil_{\mathsf{val}_V}$ yields another expression such that

$$[\![\, \lceil\!\lceil e \rceil\!\rceil_{\mathsf{val}_V}]\!]_{\mathsf{val}_{V'}} = [\![e]\!]_{\mathsf{val}_{V'} \triangleright \mathsf{val}_V} \quad \text{when } \mathsf{freevars}(e) \setminus V \subseteq V'$$

An *update* $\delta \in \Delta_V$ is a function mapping one valuation val_V into another val'_V. Updates model the effect of process transitions on variable states. The exact details are unimportant, except that it will be necessary to determine the set of clocks reset by an update, $\mathsf{resets}(\delta)$. Clocks may not be set to arbitrary values, only reset.

Channels and actions

Let $\mathsf{Chansets}$ be a finite collection of *channel sets.* Every $C \in \mathsf{Chansets}$ is associated with a sequence of n_C types, each written as either $[l, u]$ for $\mathbb{Z}_{[l,u]}$, or $s \in \mathsf{Scalar}$ for $\mathbb{S}_s$. A single element $C_{\langle i_1,\ldots,i_{n_C}\rangle}$ of the set C is designated by a sequence of values $\langle i_1, \ldots, i_{n_C}\rangle$ of the respective types.

An empty sequence of types ϵ denotes a singleton channel set $C = \{c\}$. We write c as shorthand for C_ϵ.

Two *actions* are associated with each channel. Given a set of channels C,

$$\begin{aligned} C? &= \{\, C_{\langle i_1 \ldots i_{n_C}\rangle}? \mid \text{for all } \langle i_1, \ldots, i_{n_C}\rangle \,\}, \text{ and,} \\ C! &= \{\, C_{\langle i_1 \ldots i_{n_C}\rangle}! \mid \text{for all } \langle i_1, \ldots, i_{n_C}\rangle \,\} \end{aligned}$$

are associated sets of input and output actions, respectively. We will use a suffix of ?/! when a statement applies to an action regardless of direction. For any $\mathcal{C} \subseteq \mathsf{Chansets}$, let $\mathcal{C}? = \bigcup_{C \in \mathcal{C}} C?$ and $\mathcal{C}! = \bigcup_{C \in \mathcal{C}} C!$. Letting τ represent the distinguished silent action, sets of actions are given by

$$\mathsf{Acts} = \mathsf{Chansets}? \,\dot{\cup}\, \mathsf{Chansets}!, \qquad \mathsf{Acts}_\tau = \mathsf{Acts} \,\dot{\cup}\, \{\tau\},$$

[†] we write K for a set of clocks because C will be used for a set of channels

where we write $\dot{\cup}$ to mean union with an implicit assumption of disjointness. Given an action a, where $a \neq \tau$, its complement is written $\overline{a}$, for example, if $a = c?$ then $\overline{a} = c!$, and vice versa.

Subsets of a channel set C may be designated by a sequence of expressions $\langle e_1, \ldots, e_{n_C} \rangle$; written as $C[e_1, \ldots, e_{n_C}]$. The set of free variables is

$$\mathsf{freevars}(C[e_1, \ldots, e_{n_C}]) = \bigcup_{0<j\leq n_C} \mathsf{freevars}(e_j).$$

The definition of an evaluation can be lifted to designations of channel subsets to specify a single channel from the set,

$$[\![\, C[e_1, \ldots, e_{n_C}] \,]\!]_{\mathsf{val}_V} = C_{\langle [\![e_1]\!]_{\mathsf{val}_V}, \ldots, [\![e_{n_C}]\!]_{\mathsf{val}_V} \rangle}.$$

2.2 Processes

An Uppaal model comprises one or more communicating processes. Features like variables, selection bindings, and channel arrays assist users to express timed behaviours.

Definition 2.1 *A process $\mathcal{P} = (N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V\cup K}, E)$ over Acts_τ comprises finite sets of nodes N, clocks K, variables V, and edges E; an initial node n_0 and valuation $\mathsf{val}_V^{\mathrm{init}}$; and a function $\mathsf{inv}_{V\cup K}$ that maps each node to an invariant expression over V and K. The edges connects pairs of nodes with a structured label,*

$$E \subseteq N \times S \times E_g(K, V \cup S) \times 2^{\mathsf{Acts}} \times \Delta_{V\cup K}(V, S) \times N \qquad \text{where } S \subseteq \mathsf{Vars}.$$

For $(n, S, e, C[e_1, \ldots, e_{n_C}]?/!, \delta(V, S), n') \in E$ we write $n \xrightarrow[C[e_1,\ldots,e_{n_C}]?/!]{S \;\, e \;\, \delta(V,S)}_E n'$, and similarly for τ-transitions, where

- *n and n' are, respectively, source and destination nodes.*
- *S is a finite set of* selection bindings*, where $S \cap V = \emptyset$.*
- $\mathsf{freevars}(e) \subseteq V \cup S$
- *the action set is either $\{\tau\}$, or it must be consistent in direction, all inputs or all outputs, and name, that is, it must be expressible in the form $C[e_1, \ldots, e_{n_C}]?/!$ where $\mathsf{freevars}(C[e_1, \ldots, e_{n_C}]) \subseteq V \cup S$,*
- *$\delta(V, S)$ is an update for valuations over V and K that depend on a valuation of V and S.*

Expressions of the form $E_g(K, V)$ are restricted so that they can be represented by the symbolic zones used by the Uppaal model checking algorithm. To define these expressions we first restrict the form of terms, particularly those involving clocks.

Definition 2.2 *The set of* clock terms $T_{clk}(K,V)$ *is the smallest set containing*

1. p_{nclk} *where* $\mathsf{freevars}(p_{nclk}) \subseteq V$
2. $k\ R\ e$ *where* $k \in K, R \in \{<, \leq, =, \geq, >\}, \mathsf{freevars}(e) \subseteq V$
3. $k_1 - k_2\ R\ e$ *where* $k_1, k_2 \in K, R \in \{<, \leq, =, \geq, >\}, \mathsf{freevars}(e) \subseteq V$

Form (1) represents boolean-valued, clock-free expressions. In form (2) a single clock variable is compared to an integer-valued expression that contains no clocks, using one of five relations. Form (3) is similar but compares the difference of two clock variables.

Definition 2.3 *The set of* guard expressions *over K and V,* $E_g(K,V)$*, where K and V are sets of clock and non-clock variables respectively, is the smallest set that can be built using the rules:*

$$\frac{p \in T_{clk}(K,V)}{p\ \in E_g(K,V)}\,{\scriptstyle 1} \qquad \frac{p, q \in E_g(K,V)}{p \wedge q\ \in E_g(K,V)}\,{\scriptstyle 2} \qquad \frac{p \in E_g(K,V) \quad v \in \mathsf{Vars}}{(\forall v.\, p)\ \in E_g(K,V)}\,{\scriptstyle 3}$$

Guard expressions can be represented compactly by convex polygons because of the restriction to conjunctive forms. Disjunction and existential quantification effectively allow such polygons to be pasted together into more complicated shapes that cannot be represented as efficiently. Uninterpreted clock-free expressions p_{nclk} may contain existential quantifiers and disjunctive sub-terms.

2.3 Automata

Automata are generated from processes by expanding references to non-clock variables.

Definition 2.4 *An automaton* $\mathcal{A}$*, over* Acts_τ*, is a tuple* $(L, l_0, K, \mathsf{inv}_K, T)$ *comprising finites sets of locations L and clocks K, an initial location* l_0*, an invariant function* inv_K *from L to expressions over K, and a transition relation* $T \subseteq L \times \Phi(K) \times \mathsf{Acts}_\tau \times 2^K \times L$*. Transition guards are taken from* $\Phi(K)$ *and depend only on clock variables. Each transition resets a subset of clocks from K.*

A transition $(l, \phi(K), a, R, l') \in T$ *is written* $l \xrightarrow[a]{\phi(K)\ \ R}_T l'$.

The definition of expand_V effectively defines what the process modelling notation means. Process manipulations will be justified by their effect on the underlying automata, which are easier to reason about.

Definition 2.5
Let $(L, l_0, K, \mathsf{inv}_K, T) = \mathsf{expand}_V(N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V \cup K}, E)$*, where*

- $L = N \times \mathsf{Vals}_V$
- $l_0 = \big(n_0, \mathsf{val}_V^{\mathrm{init}}\big)$
- $\mathsf{inv}_K\,((n, \mathsf{val}_V)) = \lceil\!\!\lceil\, \mathsf{inv}_{V \cup K}(n)\, \rceil\!\!\rceil_{\mathsf{val}_V}$
- *T is the smallest relation satisfying:*

$$\dfrac{n \xrightarrow[C[e_1,\ldots,e_2]?]{S\ \ e\ \ \delta(V,S)}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[\llbracket C[e_1,\ldots,e_{n_C}\,]?\rrbracket_{\mathsf{val}_V \triangleright \mathsf{val}_S}]{\lceil\!\lceil e \rceil\!\rceil_{\mathsf{val}_V \triangleright \mathsf{val}_S}\ \ \mathsf{resets}(\delta(V,S))}_T \big(n', \delta_{(V,S)}(\mathsf{val}_V)\big)}\ \mathsf{expand}_\mathsf{V}?$$

$$\dfrac{n \xrightarrow[C[e_1,\ldots,e_2]!]{S\ \ e\ \ \delta(V,S)}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[\llbracket C[e_1,\ldots,e_{n_C}\,]!\rrbracket_{\mathsf{val}_V \triangleright \mathsf{val}_S}]{\lceil\!\lceil e \rceil\!\rceil_{\mathsf{val}_V \triangleright \mathsf{val}_S}\ \ \mathsf{resets}(\delta(V,S))}_T \big(n', \delta_{(V,S)}(\mathsf{val}_V)\big)}\ \mathsf{expand}_\mathsf{V}!$$

$$\dfrac{n \xrightarrow[\tau]{S\ \ e\ \ \delta(V,S)}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[\tau]{\lceil\!\lceil e \rceil\!\rceil_{\mathsf{val}_V \triangleright \mathsf{val}_S}\ \ \mathsf{resets}(\delta(V,S))}_T \big(n', \delta_{(V,S)}(\mathsf{val}_V)\big)}\ \mathsf{expand}_\mathsf{V}\tau$$

The three rules differ only in the type of action addressed. Every selection binding in S is assigned a value via val_S. The states in the conclusion are formed by pairing the control nodes n and n' of the premise with a valuation val_V of variables in V. The destination valuation is formed by applying the update δ, whose effect may depend on variables and selection bindings, to the original valuation. The clock variables reset by δ are distilled for inclusion on the resulting transition. Guard expressions in the process e are partially evaluated against both valuations; in the result they depend only on the values of the clock variables in K. For the rules $\mathsf{expand}_V?$ and $\mathsf{expand}_V!$, each valuation effectively selects a single channel, and hence action, from the array C. The rule $\mathsf{expand}_V\tau$ is simpler in this regard.

Every edge corresponds to a set of transitions determined by the state, that is the combination of control node and data valuation, and all possible assignments to S. This notion forms the basis of later manipulations.

Two operators over automata are necessary to understand the closed system semantics given to Uppaal models for model checking and simulation. Their definitions provide background to subsequent developments, though the details are not critical. We adopt CCS-like definitions [Sto02, Definition 7.5.6]: given two automata, each may either act alone, or transitions labelled with complementary actions may synchronise resulting in a silent action and thus precluding further synchronisation (and ruling broadcast channels out of consideration).

Definition 2.6 *The* parallel composition *of two automata* $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ *is defined* $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ *where*

$$\begin{aligned} L &= L_1 \times L_2 \\ l_0 &= (l_{01}, l_{02}) \\ K &= K_1 \mathbin{\dot\cup} K_2 \\ \mathsf{inv}_K(l_1, l_2) &= \mathsf{inv}_K(l_1) \wedge \mathsf{inv}_K(l_2) \end{aligned}$$

T is the smallest relation such that

$$\frac{l_1 \xrightarrow[a_1]{\phi_1\ \ R_1}_T l_1'}{(l_1, l_2) \xrightarrow[a_1]{\phi_1\ \ R_1}_T (l_1', l_2)}\ \textsf{left} \qquad \frac{l_2 \xrightarrow[a_2]{\phi_2\ \ R_2}_T l_2'}{(l_1, l_2) \xrightarrow[a_2]{\phi_2\ \ R_2}_T (l_1, l_2')}\ \textsf{right}$$

$$\frac{l_1 \xrightarrow[a]{\phi_1\ \ R_1}_T l_1' \quad l_2 \xrightarrow[\bar{a}]{\phi_2\ \ R_2}_T l_2'}{(l_1, l_2) \xrightarrow[\tau]{\phi_1 \wedge \phi_2\ \ R_1 \cup R_2}_T (l_1', l_2')}\ \textsf{both}$$

and $a_1, a_2 \in \mathsf{Acts}_\tau$*,* $a \in \mathsf{Acts}$.

The *restriction* operator prunes all transitions labelled with actions from a given set.

Definition 2.7 *The* restriction *of an automaton* $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ *to actions over channels not in a set* P *is written* $\mathcal{A} \setminus P = (L, l_0, K, \mathsf{inv}_K, T')$*, where* T' *is the smallest relation such that*

$$\frac{l \xrightarrow[a]{\phi\ \ R}_T l' \quad a \notin (P? \cup P!\,)}{l \xrightarrow[a]{\phi\ \ R}_{T'} l'}\ \textsf{restrict}$$

An Uppaal model of n processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ communicating over sets of channels Chans, perhaps encompassing several channel sets, and variables V can be mapped to an automaton

$$\mathcal{A} = (\mathsf{expand}_V(\mathcal{P}_1) \parallel \ldots \parallel \mathsf{expand}_V(\mathcal{P}_n)) \setminus \mathsf{Chans}$$

This is a closed system semantics [Sto02]; the resulting automaton contains only the silent τ transitions made by individual processes and those resulting from shared communications of pairs of processes. No further interactions with the environment are possible.

2.4 Timed Transition Systems (TTS)

Timed Transition Systems model the bare essentials of sequential behaviour in real-time.

Definition 2.8 *A* timed transition system*, or TTS, is a triple* $(S, s_0, \longrightarrow)$ *where* $s_0 \in S$ *and* $\longrightarrow \subseteq S \times (\mathbb{R}^{\geq 0} \mathbin{\dot\cup} \mathsf{Acts}_\tau) \times S$*. The transition relation comprises* delay transitions $s \xrightarrow{d} s'$*, where* $d \in \mathbb{R}^{\geq 0}$*, and* action transitions $s \xrightarrow{a} s'$*, where* $a \in \mathsf{Acts}_\tau$*.*

Uppaal simulates and verifies closed *timed transition systems*: those where action labels are limited to the subset $\mathbb{R}^{\geq 0} \cup \{\tau\}$. A less restrictive definition, as presented above, is useful for defining the open system semantics needed to formalise traces and trace inclusion [Sto02, §7.5.2].

The behaviour over time of automata is specified with clocks. The precise meaning of operations and guards on clocks in automata is given by translation to TTS.

Definition 2.9 *Let* $(S, s_0, \longrightarrow) = \mathsf{expand}_K(L, l_0, K, \mathsf{inv}_K, T)$ *be defined*

- $S = \Big\{ (l, \mathsf{val}_K) \mid l \in L \text{ and } [\![\mathsf{inv}_K(l)]\!]_{\mathsf{val}_K} \Big\}$,
- $s_0 = \big(l_0, \mathsf{val}^0_K\big)$*, assuming* $[\![\mathsf{inv}_K(l_0)]\!]_{\mathsf{val}^0_K}$

and $\longrightarrow$ *is the smallest relation satisfying*

$$\dfrac{l \xrightarrow[a]{\Phi\ \ R}_T l' \quad \mathsf{val}_K \in \mathsf{Vals}_K \quad [\![\Phi]\!]_{\mathsf{val}_K} \quad [\![\mathsf{inv}_K(l')]\!]_{\mathsf{val}_K \rhd \mathsf{val}^0_R}}{(l, \mathsf{val}_K) \xrightarrow{a} \big(l', \mathsf{val}_K \rhd \mathsf{val}^0_R\big)}\ \mathsf{action}$$

$$\dfrac{d \in \mathbb{R}^{\geq 0} \quad \mathsf{val}_K \in \mathsf{Vals}_K \quad \forall 0 \leq t \leq d.\ [\![\mathsf{val}^{+t}_K]\!]_{\mathsf{inv}_K(l)}}{(l, \mathsf{val}_K) \xrightarrow{d} \Big(l, \mathsf{val}^{+d}_K\Big)}\ \mathsf{delay}$$

where $a \in \mathsf{Acts}_\tau$*.*

Each state of the TTS pairs an automaton location l with a valuation of the set of clocks val_K. States where the location invariant is not satisfied by the valuation are excluded, we assume that this is not the case for the initial state where all clocks have value zero. An action transition is possible between two states when there exists an automaton transition between the location components whose guard is satisfied by the source clock valuation, and where clocks reset on the transition have zero value in the destination clock valuation, the latter satisfying the invariant at the destination location. A delay transition of value d exists between states with the same location component, where clock values at the destination are d greater than those at the source, and where the location invariant is satisfied for all delay transitions of lesser value.

Given a closed TTS $\mathcal{T}$ and a property φ, Uppaal decides whether the latter is satisfied by the former, that is, whether $\mathcal{T} \models \varphi$. Properties are evaluated over paths of alternating delay and action transitions beginning at the initial state s_0. They are restricted in form,

$\mathsf{E}\Diamond\, p$ asserts that it is *possible* to reach a state satisfying p on a path from s_0.

$\mathsf{E}\Box\, p$ asserts that there is a path from s_0 along which states *always* satisfy p. Either the path is infinite, or ends in a state s with no outgoing transitions, or where $s \xrightarrow{d} s'$ is possible for arbitrary d.

$\mathsf{A}\Diamond\, p$ is equivalent to $\neg\mathsf{E}\Box\, \neg p$

$\mathsf{A}\Box\, p$ is equivalent to $\neg\mathsf{E}\Diamond\, \neg p$

$p \rightsquigarrow q$ is equivalent to $\mathsf{A}\Box\, (p \text{ } \mathsf{implies} \text{ } \mathsf{A}\Diamond\, q)$

In practice, state properties, written here as p and q, are formulated over the control nodes and valuations of the component processes that were expanded to give the TTS. The deadlock property asserts that no action transitions leave a state or its time successors (states reachable via a delay transition). Uppaal can provide witness traces for properties asserting existence, that is true properties of form $\mathsf{E}\Diamond\, p$ or $\mathsf{E}\Box\, p$, or counter-examples for failed properties of form $\mathsf{A}\Diamond\, p$ or $\mathsf{A}\Box\, p$.

3 Checking trace inclusion

Given two processes, $\mathcal{P}_{\mathcal{I}}$ and $\mathcal{P}_{\mathcal{S}}$, over a set of channels Chans, we would like to verify whether the possible, open system traces of $\mathcal{P}_{\mathcal{I}}$ are a subset of those of $\mathcal{P}_{\mathcal{S}}$, that is whether $\mathcal{P}_{\mathcal{I}} \sqsubseteq_{\mathrm{TR}} \mathcal{P}_{\mathcal{S}}$. If $\mathcal{P}_{\mathcal{S}}$ is deterministic and free of τ-transitions it is possible to construct a test process $\mathcal{P}_B^{\mathrm{Err}}$, containing a distinguished error state, such that

$$\mathsf{expand}_K\left(\left(\mathsf{expand}_V(\mathcal{P}_{\mathcal{I}}) \parallel \mathsf{expand}_V(\mathcal{P}_B^{\mathrm{Err}})\right) \setminus \mathsf{Chans}\right) \models \neg\mathsf{E}\Diamond\, \mathsf{error}$$
$$\text{implies } \mathcal{P}_{\mathcal{I}} \sqsubseteq_{\mathrm{TR}} \mathcal{P}_{\mathcal{S}}.$$

The problem is usually phrased in terms of automata. The definition of a test automaton [Sto02, §A.1.5] is repeated below with a minor modification to the rule error.

Definition 3.1 *Given an automaton $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ that is deterministic, where no transitions are labelled τ, define,*

$$\mathcal{A}^{\mathrm{Err}} = (L \mathbin{\dot\cup} \{\mathsf{error}\}, l_0, K, \mathsf{inv}_K^{\mathrm{Err}}, T^{\mathrm{Err}})$$

where $\mathsf{inv}^{\mathrm{Err}}_K(l \in L) = \mathsf{true}$ *and* T^{Err} *is the smallest relation such that*

$$\dfrac{l \xrightarrow[a]{g\ \ R}_T l'}{l \xrightarrow[\overline{a}]{(g \wedge \mathsf{inv}_K(l))\ \ R}_{T^{\mathrm{Err}}} l'}\ \mathsf{legal} \qquad \dfrac{g_{(a,l)} = \neg \bigvee \{\, g \mid l \xrightarrow[a]{g\ \ R}_T l' \,\}}{l \xrightarrow[\overline{a}]{(g_{(a,l)} \wedge \mathsf{inv}_K(l))\ \ \emptyset}_{T^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{illegal}$$

$$\dfrac{}{l \xrightarrow[\tau]{\neg\mathsf{inv}_K(l)\ \ \emptyset}_{T^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{notinv} \qquad \dfrac{}{\mathsf{error} \xrightarrow[\tau]{\mathsf{true}\ \ \emptyset}_{T^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{error}$$

and $a \in \mathsf{Acts}$*.*

If there are no transitions for a certain pairing of action and location (a, l) the upper part of the *illegal* rule becomes $\neg \bigvee \emptyset = \mathsf{true}$, giving a transition directly to the error state.

Given $\mathcal{P}$ we would like to construct $\mathcal{P}^{\mathrm{Err}}$ such that:

$$\begin{tikzcd}
\mathcal{P} \arrow[r] \arrow[d, "\mathsf{expand}_V"'] & \mathcal{P}^{\mathrm{Err}} \arrow[d, "\mathsf{expand}_V"] \\
\mathcal{A} \arrow[r] & \mathcal{A}^{\mathrm{Err}}
\end{tikzcd} \qquad \mathsf{expand}_V(\mathcal{P})^{\mathrm{Err}} = \mathsf{expand}_V(\mathcal{P}^{\mathrm{Err}})$$

In pragmatic terms, the aim is to correctly extend the original construction to address more Uppaal features, thereby allowing trace refinement verification for a larger subset of models directly from Uppaal. The formalisation also serves as a foundation for understanding the algorithm.

Definition 3.2 *Let* $\mathcal{P}$ *be a process* $(N, n_0, K, V, \mathsf{val}^{\mathrm{init}}_V, \mathsf{inv}_{V\cup K}, E)$ *where the underlying automaton* $\mathsf{expand}_V(\mathcal{P})$ *is deterministic and free of* τ*-transitions, then*

$$\mathcal{P}^{\mathrm{Err}} = (N \mathbin{\dot\cup} \{\mathsf{error}\}, n_0, K, \mathsf{val}^{\mathrm{init}}_V, \mathsf{inv}^{\mathrm{Err}}_{V\cup K}, E^{\mathrm{Err}})$$

where $\mathsf{inv}^{\mathrm{Err}}_{V\cup K}(l \in L) = \mathsf{true}$ *and* E^{Err} *is the smallest relation such that*

$$\dfrac{n \xrightarrow[C[e_1,\ldots,e_{n_C}]?]{S\ \ g\ \ \lambda}_E n'}{n \xrightarrow[C[e_1,\ldots,e_{n_C}]!]{S\ \ (g \wedge \mathsf{inv}_{V\cup K}(n))\ \ \lambda}_{E^{\mathrm{Err}}} n'}\ \mathsf{plegal?} \qquad \dfrac{n \xrightarrow[C[e_1,\ldots,e_{n_C}]!]{S\ \ g\ \ \lambda}_E n'}{n \xrightarrow[C[e_1,\ldots,e_{n_C}]?]{S\ \ (g \wedge \mathsf{inv}_{V\cup K}(n))\ \ \lambda}_{E^{\mathrm{Err}}} n'}\ \mathsf{plegal!}$$

$$\dfrac{}{n \xrightarrow[\tau]{\emptyset\ \ \neg\mathsf{inv}_{V\cup K}(n)\ \ \cdot}_{E^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{pnotinv} \qquad \dfrac{}{\mathsf{error} \xrightarrow[\tau]{\emptyset\ \ \mathsf{true}\ \ \cdot}_{E^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{perror}$$

$$\dfrac{\begin{array}{c} n \in N \qquad C \in \mathsf{Chansets} \\ (S', g', \langle e'_1, \ldots, e'_{n_C}\rangle) \in \mathsf{flip}\Big(C, \Big\{(S, g, \langle e_1, \ldots, e_{n_C}\rangle) \mid n \xrightarrow[C[e_1,\ldots,e_{n_C}]!]{S\ \ g\ \ \cdot}_E \cdot\Big\}\Big) \end{array}}{n \xrightarrow[C[e'_1,\ldots,e'_{n_C}]?]{S'\ \ (g' \wedge \mathsf{inv}_{V\cup K}(n))\ \ \cdot}_{E^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{pillegal?}$$

$$\frac{n \in N \qquad C \in \mathsf{Chansets} \qquad (S', g', \langle e'_1, \ldots, e'_{n_C} \rangle) \in \mathsf{flip}\Big(C, \Big\{(S, g, \langle e_1, \ldots, e_{n_C} \rangle) \mid n \xrightarrow[C[e_1,\ldots,e_{n_C}]?]{S\ \ g\ \ \cdot}_E \cdot \Big\}\Big)}{n \xrightarrow[C[e'_1,\ldots,e'_{n_C}]!]{S'\ \ (g' \wedge \mathsf{inv}_{V \cup K}(n))\ \ \cdot}_{E^{\mathrm{Err}}} \mathsf{error}}\ \mathsf{pillegal!}$$

where flip *maps a set of triples—of selection bindings, guards, and subscript expressions—of edges for a fixed location l and channel set C, to another set of triples such that one of the triples will be enabled for a single action at a location of the underlying automaton* $\mathsf{expand}_V(\mathcal{P}^{\mathrm{Err}})$ *iff no transition from the same location with the same action is enabled in the corresponding location of* $\mathsf{expand}_V(\mathcal{P})$*. Edges to the* error *state must serve for all actions on the labelling channel set.*

The following subsections describe, constructively, the flip function for a large class of, but not all, edge sets. In doing so, we give a means of automating a significant component of trace inclusion testing for a larger class of Uppaal models than directly addressed by Definition 3.1 alone. The techniques apply to Uppaal templates, that is processes where some constant values are unknown. Although Uppaal insists that such values are determinable at compile time, we prefer to treat their values as arbitrary. This does preclude handling some models, but doing so would likely involve explicit edge expansions, which is less elegant and rather inefficient.

The flip function is described by gradually increasing its domain. We begin with singleton channel sets, making adjustments progressively to incorporate selection bindings and universal quantifiers. Then a broader class of channels sets is considered: at first limited to expressions over state variables and then extended to include selection bindings in a limited way.

3.1 Basic channels

This section describes the flip function for edges that are labelled with singleton channel sets. We focus on handling selection bindings over guards only and producing transition resultant guards that are acceptable to Uppaal. These techniques are extended, in the next section, to more general channel sets.

The function and its implementation is described progressively over the next three subsections. We begin with simple transition guards, then admit selection bindings, and finally explain the additional problems introduced by universal quantifiers.

No selection bindings or quantifiers

In the absence of selection bindings and quantifiers over expressions containing clocks, the challenge is to ensure that an implementation produces guard expressions that meet the syntactic restrictions of Uppaal. Ideally, expressions are simplified whenever possible.

We define a set of clock expressions to represent the intermediate results of manipulations. They must be converted back into guard expressions to be acceptable to Uppaal.

Definition 3.3 *The set of* clock expressions $E_{clk}(K, V)$ *over sets of clock K and non-clock variables V is the smallest set that can be built using the rules:*

$$\frac{p \in T_{clk}(K, V)}{p \ \in E_{clk}(K, V)}\ \textsf{clkterm}$$

$$\frac{p, q \in E_{clk}(K, V)}{p \wedge q \ \in E_{clk}(K, V)}\ \textsf{clkand} \qquad \frac{p, q \in E_{clk}(K, V)}{p \vee q \ \in E_{clk}(K, V)}\ \textsf{clkor}$$

$$\frac{p \in E_{clk}(K, V) \quad v \in \textsf{Vars}}{\forall v.\, p \ \in E_{clk}(K, V)}\ \textsf{clkall} \qquad \frac{p \in E_{clk}(K, V) \quad v \in \textsf{Vars}}{\exists v.\, p \ \in E_{clk}(K, V)}\ \textsf{clkexists}$$

Proposition 1 *For any sets of clocks K and variables V,*

$$E_g(K, V) \subseteq E_{clk}(K, V)^*.$$

For the present, we limit ourselves to clock expressions constructed without using either of the clkall or clkexists rules. Quantifier bindings within clock terms, like $p \in T_{clk}(K, V)$, are not considered significant because they do not enclose clock variables.

Definition 3.4 *Given a clock expression e constructed without* clkall *and* clkexists*,* $\textsf{neg}(e)$ *is a function that returns a clock expression* e'*.*

$$\begin{array}{rcl}
\textsf{neg}(p_{nclk}) & = & \textsf{neg}_{nclk}(p_{nclk}) \\
\textsf{neg}(c < e) & = & c \geq e \\
\textsf{neg}(c \leq e) & = & c > e \\
\textsf{neg}(c = e) & = & c < e \vee c > e \\
\textsf{neg}(c \geq e) & = & c < e \\
\textsf{neg}(c > e) & = & c \leq e \\
\textsf{neg}(c_1 - c_2 < e) & = & c_1 - c_2 \geq e \\
\textsf{neg}(c_1 - c_2 \leq e) & = & c_1 - c_2 > e \\
\textsf{neg}(c_1 - c_2 = e) & = & c_1 - c_2 < e \vee c_1 - c_2 > e \\
\textsf{neg}(c_1 - c_2 \geq e) & = & c_1 - c_2 < e \\
\textsf{neg}(c_1 - c_2 > e) & = & c_1 - c_2 \leq e \\
\textsf{neg}(p \wedge q) & = & \textsf{neg}(p) \vee \textsf{neg}(q) \\
\textsf{neg}(p \vee q) & = & \textsf{neg}(p) \wedge \textsf{neg}(q)
\end{array}$$

[*] clock terms and guard expressions are defined in Definitions 2.2 and 2.3 respectively.

where $\mathsf{neg}_{nclk}(p_{nclk})$ *gives the logical negation of* p_{nclk} *in an appropriate subexpression language.*

Proposition 2 *The function* neg *is closed over the set of clock expressions constructed without using* clkall *and* clkexists*.*

Proposition 3 *For any clock expression* e*, constructed without using* clkall *and* clkexists*, and any valuation* $\mathsf{val}_V \in \mathsf{Vals}_V$*,* $\neg [\![e]\!]_{\mathsf{val}_V} = [\![\mathsf{neg}\, e]\!]_{\mathsf{val}_V}$*.*

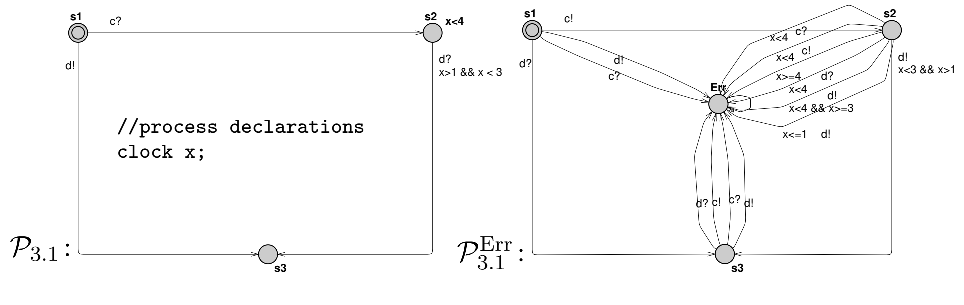
The set of m edges to be flipped can be written $E = \{g_1, \ldots, g_m\}$, because each edge has the same source node n, none have selection bindings, each performs the same action a, and neither updates nor destination nodes are relevant.

The result of flip should contain guards such that at least one is true exactly when all of the guards in E are not. We directly mimic the premise of rule $\mathsf{illegal}$ in Definition 3.1 by forming $\mathsf{neg}(g_1 \vee \cdots \vee g_m)$. To ensure that the resultant expression conforms to the syntactic restrictions of Uppaal, it must first be converted into *disjunctive normal form* (DNF) $\overline{g}_1 \vee \cdots \vee \overline{g}_{m'}$, before separating the clauses to give $\overline{E} = \{\overline{g}_1, \ldots, \overline{g}_{m'}\}$. Each component of which will guard a transition in $\mathcal{P}^{\mathrm{Err}}$:

$$n \xrightarrow[\overline{a}]{\emptyset \;\; (\overline{g}_i \wedge \mathsf{inv}_{V \cup K}(n)) \;\; \cdot}{}_{E^{\mathrm{Err}}} \mathsf{error}$$

due to either the rule $\mathsf{pillegal?}$ or $\mathsf{pillegal!}$ of Definition 3.2.

In practice, it is often possible to simplify the resulting guard terms. For example, $(c > 2) \wedge (c \leq 2)$ may be omitted completely, and $(c < 2) \wedge (c < 4)$ may be replaced with $(c < 4)$. Whilst not strictly necessary, such simplifications improve the readability of the results which, in turn, increases confidence in their correctness, and makes traces within Uppaal easier to follow. The current version of the tool uses simple syntactic criteria to assess, for a pair of terms in a conjunctive clause, whether one implies or contradicts the other. One possible improvement would be to exploit a heavy duty simplifier, as used in theorem provers like HOL or Isabelle.



```
//process declarations
clock x;
```

Figure 3.1: Example for guards without selection bindings or quantifiers

The left side of Figure 3.1 shows a process $\mathcal{P}_{3.1}$ that has one clock x and synchronises over channels c and d. The right side shows $\mathcal{P}_{3.1}^{\mathrm{Err}}$. The two transitions from s_1 of $\mathcal{P}_{3.1}^{\text{Err}}$ to error are labelled with complements of the actions not leaving s_1 of $\mathcal{P}_{3.1}$. The guards of transitions leaving s_2 of $\mathcal{P}_{3.1}^{\text{Err}}$ are more involved

$c?$, $c!$, and $d?$	$x < 4$	node invariant becomes guard
τ	$x \geq 4$	negated invariant, $\mathsf{neg}(\mathsf{inv}_{V \cup K}(n))$
$d!$	$x < 3 \wedge x > 1$	original guard
$d!$	$x < 4 \wedge x \geq 3$	invariant and half of negated guard
$d!$	$x \leq 1$	other half of negated guard

In the third and fourth cases, simplification has removed the unnecessary invariant term from the conjunct. The negated guard of $d!$ is split across two transitions because it would otherwise be a disjunction of two clock terms.

Handling selection bindings

Selection bindings are a relatively new feature of Uppaal. A selection binding pairs a variable name with either a bounded integer or scalar type. There may be multiple bindings on an edge, the names are bound over the guard expression, update statement, and action array indices. The latter possibility is not considered until §3.2.

An edge with selection bindings represents multiple transitions in the corresponding unwound automaton, even after fixing the values of state variables. This is apparent from the transition rules of Definition 2.5. Given a node and variable valuation, any choice of values for the selection bindings that satisfies the guard represents a possible transition. The simulator will present each such valuation as a distinct transition, while the model checker will explore all possibilities.

A set of m edges to be flipped, must now be written

$$E = \{(S_1, g_1), \ldots, (S_m, g_m)\},$$

where each S_i is a set of selection variables, bound over g_i. Again, we assume that the g_i are constructed without using the *clkall* and *clkexists* rules, and that the selection sets are pairwise disjoint—elements can be renamed if necessary.

A transition for a given location and action is enabled whenever

$$(\exists s_{11}, \ldots, s_{1n_1} . \, g_1) \vee \cdots \vee (\exists s_{m1}, \ldots, s_{mn_m} . \, g_m)$$

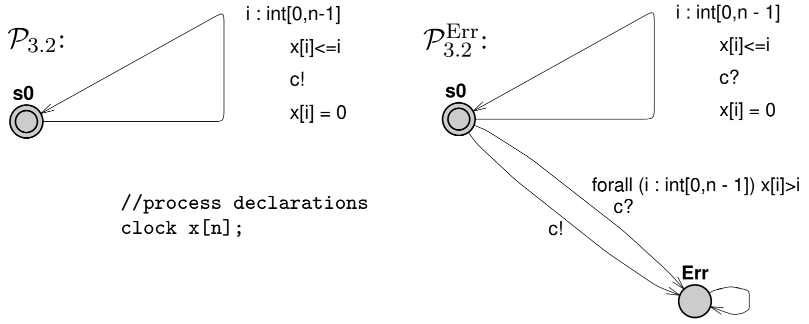
which can be rewritten

$$\exists s_{11}, \ldots, s_{1n_1}, \ldots, s_{m1}, \ldots, s_{mn_m} . \, g_1 \vee \cdots \vee g_m.$$

Thus, no transitions are enabled when

$$\forall s_{11}, \ldots, s_{1n_1}, \ldots, s_{m1}, \ldots, s_{mn_m} . \, \mathsf{neg}(g_1 \vee \cdots \vee g_m),$$

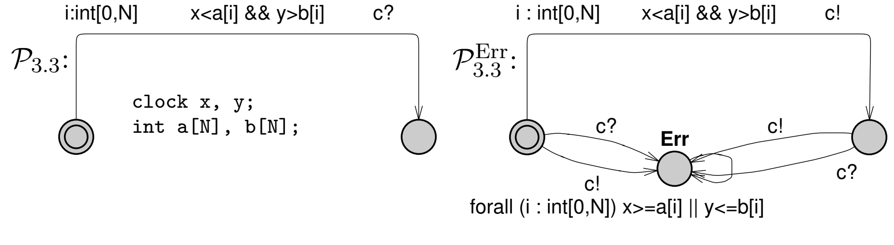
that is, when, for all valuations of the selection bindings, no guard is satisfied. The result of $\mathsf{neg}(g_0 \vee \cdots \vee g_m)$ can be converted to DNF $\overline{g}_0 \vee \cdots \vee \overline{g}_m$, but it is only possible to assign each clause to a separate transition if the scope of each universally bound variable can be reduced to a single disjunct. This is disappointing because it limits the processes for which the construction can be automated. An alternative would be to eliminate the quantified variables that apply over more than one disjunct, by generating a separate transition for each possible value, and for every disjunct within scope. Such manipulations would increase, possibly greatly, the number of edges. They are not possible when the variable takes values from a scalar set, or from a bounded integer where either of the bounds is an expression that cannot be reduced to a concrete value, for instance, expressions involving template arguments. The current tool displays a warning message when a negated expression cannot be split into separate transitions and hence will likely be rejected by Uppaal.



```
//process declarations
clock x[n];
```

Figure 3.2: Example for guards with selection bindings but no quantifiers

In $\mathcal{P}^{\mathrm{Err}}_{3.2}$ of Figure 3.2 a transition from s_0 to error occurs on $c?$ only when the negated guard is true for all possible values of i.



```
clock x, y;
int a[N], b[N];
```

Figure 3.3: Example where selection bindings clash with a negated guard

Another process and corresponding test process are depicted in Figure 3.3. The disjunct guard clauses in the latter cannot be split into separate transitions due to the `forall` binding. The error process $\mathcal{P}^{\mathrm{Err}}_{3.3}$ is rejected Uppaal.

Handling quantifiers

The previous section showed how universal quantifiers are introduced when negating transitions with selection bindings. We now consider the case where edge guards additionally already contain universal quantifiers, that is the clkall rule may be used to construct expressions.

A quantifier binding, like a selection binding, pairs a name and finite type. Names are bound over subexpressions. A universally quantified expression $\forall i \in \mathbb{Z}_{[l,u]} .\, e(i)$ effectively expands to a conjunctive sequence $e(l) \wedge \cdots \wedge e(u)$, and the existential variety $\exists i \in \mathbb{Z}_{[l,u]} .\, e(i)$ to a disjunctive sequence $e(l) \vee \cdots \vee e(u)$. As existential bindings may split clock zones, they may not enclose subexpressions containing clocks, and thus need not be addressed by the flip function.

Since it is possible to convert a guard expression into prenex normal form where all the quantifiers are universal, and thus order is irrelevant, a set of m edges may now be written

$$E = \{(S_1, A_1, g_1), \ldots, (S_m, A_m, g_m)\},$$

where each A_i is a set of universally quantified variables binding over g_i. We assume that all selection and quantifier sets are pairwise disjoint, and further that quantified variables only occur in corresponding guard expressions, that is,

$$\begin{array}{rl} \forall\, 1 \le i, j \le m. & S_i \cap A_j = \emptyset \\ \forall\, 1 \le i, j \le m, i \neq j. & S_i \cap S_j = \emptyset,\;\; A_i \cap A_j = \emptyset,\;\; A_i \cap \mathsf{freevars}(g_j) = \emptyset, \\ & S_i \cap \mathsf{freevars}(g_j) = \emptyset \end{array}$$

These assumptions can be met by renaming as required.

For a fixed valuation, a transition for the given action is enabled whenever

$$\exists s_{11}, \ldots, s_{mn_m} .\, \forall a_{11}, \ldots, a_{mn'_m} .\, g_1 \vee \cdots \vee g_m.$$

Thus, there are no transitions enabled for the action when

$$\forall s_{11}, \ldots, s_{mn_m} .\, \exists a_{11}, \ldots, a_{mn'_m} .\, \mathsf{neg}(g_1 \vee \cdots \vee g_m), \qquad (\psi_1)$$

which is problematic because if any of the guards contain clock variables, Uppaal will reject the expression. There would seem to be hope only for those cases where the expression can be manipulated into the form

$$\exists a_{11}, \ldots, a_{mn'_m} .\, \forall s_{11}, \ldots, s_{mn_m} .\, \mathsf{neg}(g_1 \vee \cdots \vee g_m), \qquad (\psi_2)$$

The existential bindings in the prefix could then be converted into selection bindings, with the remainder of the expression having the form discussed in §3.1 and subject to the same limitations and treatment.

We require some $\mathsf{condition}(\psi_1)$ such that

$$\mathsf{condition}(\psi_1) \implies \forall\, \mathsf{val}_V \in \mathsf{Vals}_V.\ [\![\psi_1]\!]_{\mathsf{val}_V} = [\![\psi_2]\!]_{\mathsf{val}_V}\ ,$$

which soundly increases the range of the technique. The most minimal condition is false which simply rejects all processes with guards that mix selection bindings and universal quantifiers over clock variables. The condition that determines whether ψ_1 is logically equivalent to ψ_2 is the most precise. It could be implemented, for example, by emitting constraints and proof obligations for treatment in a theorem prover or, if the values of all constants are known, a model-checker. Instead we have implemented a simpler and approximate condition.

Definition 3.5 *We define the* $\mathsf{canswap}$ *predicate on formulas of the form*

$$\forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \varphi_1 \vee \cdots \vee \varphi_l$$

where each $\varphi_i = p_i^1 \wedge \cdots \wedge p_i^{n_i}$. *Let*

$$\begin{aligned} A &= \{a_1, \ldots, a_n\}, & E &= \{e_1, \ldots, e_m\}, \\ A_i &= \mathsf{freevars}(\varphi_i) \cap A & E_i &= \mathsf{freevars}(\varphi_i) \cap E \end{aligned}$$

$\mathsf{canswap}$ *is true iff for each* $1 \leq i \leq l$ *either*

1. $A_i = \emptyset$ *or* $E_i = \emptyset$, *or*
2. *For all* $1 < j < n$ *where* $j \neq i$, $A_i \cap A_j = \emptyset$, $E_i \cap E_j = \emptyset$, *and for all* $1 < k < n_i$ *either* $\mathsf{freevars}(p_i^k) \cap A_i = \emptyset$ *or* $\mathsf{freevars}(p_i^k) \cap E_i = \emptyset$.

Proposition 4 *Given a quantifier free formula* ψ *in disjunctive normal form,*

$$\begin{aligned} &\mathsf{canswap}(\forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \psi) \implies \\ &\qquad\qquad \forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \psi \equiv \exists e_1, \ldots, e_m.\, \forall a_1, \ldots, a_n.\, \psi \end{aligned}$$

Proof Assume that ψ is in disjunctive normal form. It is possible, through associativity and commutativity of disjunction, to juxtapose clauses that do not contain any universally bound variables. The scope of each existential quantifier can be reduced to either the juxtaposed group or a single clause; the clauses of $\mathsf{canswap}$ guarantee the side condition of $\exists x.\, (\phi_1 \vee \phi_2) = (\exists x.\, \phi_1) \vee \phi_2 \quad (x \notin \mathsf{freevars}(\phi_2))$. The scope of the universal quantifiers can be similarly reduced.

The scope of existential quantifiers in clauses of the form $\forall A_i.\, \exists E_i.\, p_i^1 \wedge \cdots \wedge p_i^{n_i}$ can be reduced to a subset of the terms. Likewise for the scopes of universal quantifiers. The second clause of $\mathsf{canswap}$ guarantees that no existential quantifier overlaps with any universal quantifier on a term.

The scope of existential bindings can be widened to cover all clauses. Similarly for universal bindings. □

The canswap predicate is no panacea, but it does address several useful cases, for example, sets of transitions where no single transition employs both selection bindings and universal quantifiers and each guard is a single term.

3.2 Channel arrays

This section considers edges labelled with actions on elements of channel arrays, thus generalizing the techniques of the previous section. The central difficulty is in deciding when channels can be identified. For basic channels, that is singleton channel sets, two channels are equal when they share a name. An array name, however, can stand for multiple channels. Individual elements are selected by sequences of index expressions over state variables and selection bindings.

We first develop techniques for handling array index expressions where the only variables are state variables, then extend these, in a limited way, to incorporate variables from selection bindings. More general solutions are possible, but more difficult to implement, and it remains to be determined whether they are proportionately useful.

In Uppaal, channels and arrays of channels can be passed by reference as template parameters. Reference values cannot be compared for equality and thus aliasing is not easily detectable. The implementation prints a warning for templates where channels are passed by reference.

For simplicity of presentation, we assume in this section that arrays are indexed from one rather than zero.

No bindings for channel selection

Rather than collecting edges based on a single action, they must now be grouped by channel set and direction. Recall, from §2.1, that for every $C \in \mathsf{Chansets}$, there is an associated sequence of n_C types, and that a single channel in the set can be specified by a sequence of expressions $\langle e_1, \ldots, e_{n_C} \rangle$.

The set of m edges to be flipped is now written

$$E = \{\left(S_1, A_1, g_1, \langle e_1^1, \ldots, e_{n_C}^1 \rangle\right), \ldots, \left(S_m, A_m, g_m, \langle e_1^m, \ldots, e_{n_C}^m \rangle\right)\},$$

We make the previous assumptions of disjointness, and additionally require that quantifier bindings are restricted to guards, and likewise, until the next section, for selection bindings, that is,

$$\forall 1 \leq i, j \leq m, 1 \leq k \leq n_C.\, (A_i \cup S_i) \cap \mathsf{freevars}(e_k^j) = \emptyset$$

The edges within E must now be grouped by channel, and allowance made for channels in C that are not represented by edges.

For example, given $C = \{c[1], c[2]\}$, and,

$$E = \{(S_1, A_1, g_1, e^1), (S_2, A_2, g_2, e^2)\},$$

Given a valuation val_V there are two possibilities, if $[\![e^1]\!]_{\mathsf{val}_V} = [\![e^2]\!]_{\mathsf{val}_V}$ then the previous techniques can be applied to the edge $(S_1 \cup S_2, A_1 \cup A_2, g_1 \vee g_2)$ on action $c[\,[\![e^1]\!]_{\mathsf{val}_V}]$, and the edge $(\emptyset, \emptyset, \mathsf{false})$ on action $c[i]$ where $i \neq [\![e^1]\!]_{\mathsf{val}_V}$. Otherwise, if $[\![e^1]\!]_{\mathsf{val}_V} \neq [\![e^2]\!]_{\mathsf{val}_V}$ there is one edge (S_1, A_1, g_1) on $c[\,[\![e^1]\!]_{\mathsf{val}_V}]$, and another (S_2, A_2, g_2) on $c[\,[\![e^2]\!]_{\mathsf{val}_V}]$.

In general, every possible partitioning of the m edges must be considered. Unfortunately, this (Bell) number grows exponentially:

$$B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, B_6 = 203, B_7 = 877, \ldots$$

This effectively limits the models that can be addressed, based on the maximum number of edges leaving a single node, on the same channel array, in the same direction, and with unique index expressions. Before performing partitioning, the implementation merges edges that have sequences of syntactically identical index expressions by forming unions of their selection binding sets and disjunctions of their guards.

Each partition is represented by a predicate p_b that equates index expressions in the same block, and distinguishes representatives between blocks. For example, given three edge labels on a channel array of two dimensions: $\langle e_1^1, e_2^1 \rangle$, $\langle e_1^2, e_2^2 \rangle$, and $\langle e_1^3, e_2^3 \rangle$, there are five partitions and corresponding predicates:

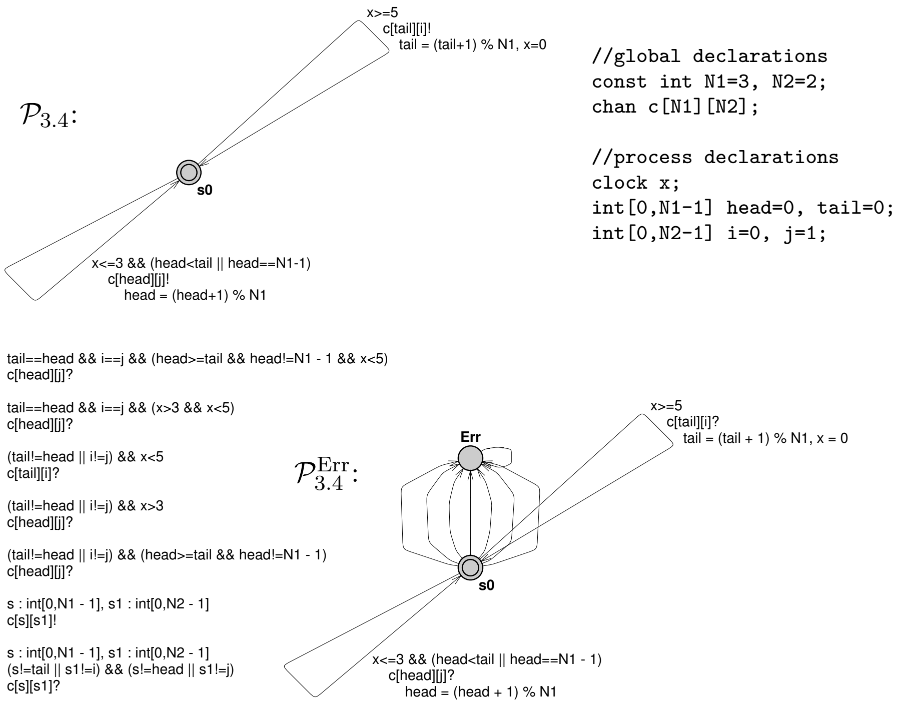
$$\begin{array}{cc}
[1,2,3] & (e_1^1 = e_1^2 \wedge e_2^1 = e_2^2 \wedge e_1^1 = e_1^3 \wedge e_2^1 = e_2^3) \\
[1,2]\ [3] & (e_1^1 = e_1^2 \wedge e_1^2 = e_2^2) \wedge (e_1^1 \neq e_1^3 \vee e_2^1 \neq e_2^3) \\
[1]\ [2,3] & (e_1^2 = e_1^3 \wedge e_2^2 = e_2^3) \wedge (e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2) \\
[1,3]\ [2] & (e_1^1 = e_1^3 \wedge e_2^1 = e_2^3) \wedge (e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2) \\
[1]\ [2]\ [3] & (e_1^1 \neq e_1^2 \vee e_2^1 \neq e_2^2) \wedge (e_1^1 \neq e_1^3 \vee e_2^1 \neq e_2^3) \wedge (e_1^2 \neq e_1^3 \vee e_2^2 \neq e_2^3)
\end{array}$$

A given valuation will only satisfy one of the predicates. In practice, partition predicates may be simplified, as per guard terms. Some subexpressions can be replaced with true, for instance comparing a variable with itself $x = x$, others with false, as when comparing two different constants $1 = 2$, which may then lead to further simplifications.

Edges within the same partition can be grouped and negated using the techniques of previous sections, provided the partition predicate is added after negating the guard: $\forall \ldots \exists \ldots p_b \wedge \mathsf{neg}(g \vee \ldots \vee g)$. This technique makes the channel subscripts irrelevant: while the exact action groupings may depend on variable values, the predicates guarantee that all possibilities are taken into account.

There is the possibility that some channel set elements are never enabled at a node. These synchronisations must also be directed to the error state by creating an edge with a selection binding for each dimension and a guard that is true whenever at least one of the selection bindings differs from all other expressions in the same dimension. For the previous three transition, two dimension example, this *cover all* edge would be

$$\big(\{i_1, i_2\}, \emptyset, \big((i_1 \neq e_1^1 \vee i_2 \neq e_2^1) \wedge (i_1 \neq e_1^2 \vee i_2 \neq e_2^2) \wedge (i_1 \neq e_1^3 \vee i_2 \neq e_2^3)\big)\big)\,.$$


$\mathcal{P}_{3.4}$:
x>=5
c[tail][i]!
tail = (tail+1) % N1, x=0
s0
x<=3 && (head<tail || head==N1-1)
c[head][j]!
head = (head+1) % N1
tail==head && i==j && (head>=tail && head!=N1 - 1 && x<5)
c[head][j]?
tail==head && i==j && (x>3 && x<5)
c[head][j]?
(tail!=head || i!=j) && x<5
c[tail][i]?
(tail!=head || i!=j) && x>3
c[head][j]?
(tail!=head || i!=j) && (head>=tail && head!=N1 - 1)
c[head][j]?
s : int[0,N1 - 1], s1 : int[0,N2 - 1]
c[s][s1]!
s : int[0,N1 - 1], s1 : int[0,N2 - 1]
(s!=tail || s1!=i) && (s!=head || s1!=j)
c[s][s1]?
$\mathcal{P}_{3.4}^{\mathrm{Err}}$:
Err
x>=5
c[tail][i]?
tail = (tail + 1) % N1, x = 0
s0
x<=3 && (head<tail || head==N1 - 1)
c[head][j]?
head = (head + 1) % N1

```
//global declarations
const int N1=3, N2=2;
chan c[N1][N2];

//process declarations
clock x;
int[0,N1-1] head=0, tail=0;
int[0,N2-1] i=0, j=1;
```

Figure 3.4: Example channel selections without bindings

In the $\mathcal{P}_{3.4}^{\mathrm{Err}}$ process of Figure 3.4 seven transitions connect the original state to the error state, from top to bottom: two for state valuations where both original transitions have the same input action, three for when both transitions have different input actions, one to cover all outputs on channels in c, and another to cover any inputs on c not present on the other edges. The constants $N1$ and $N2$ are declared globally, but they could also be template parameters. The testing process is correct regardless of their exact values.

Restricted channel selection

In the previous section we assumed the absence of selection bindings in channel subscript expressions. In Uppaal, however, channel array expressions are within the scope of selection bindings on the same edge. These further complications are partially addressed in this section and the current implementation.

The set of m edges to be flipped is again written

$$E = \{\left(S_1, A_1, g_1, \langle e_1^1, \ldots, e_{n_C}^1\rangle\right), \ldots, \left(S_m, A_m, g_m, \langle e_1^m, \ldots, e_{n_C}^m\rangle\right)\},$$

The disjointness assumptions from §3.1 still hold, and, additionally, we require, as does Uppaal, that quantifier bindings are restricted to guards,

$$\forall 1 \leq i, j \leq m, 1 \leq k \leq n_C .\, A_i \cap \mathsf{freevars}(e_k^j) = \emptyset.$$

We admit selection bindings into channel expressions in a limited way, by partitioning the subscript indexes into two classes I_{free} and I_{bound}. For each e_k, either $k \in I_{\text{free}}$ or $k \in I_{\text{bound}}$. An index expression is either completely free of selection variables, $\forall 1 \leq i \leq m, k \in I_{\text{free}}.\, S_i \cap \mathsf{freevars}(e^i_k) = \emptyset$; or it is a selection variable, $\forall 1 \leq i \leq m, k \in I_{\text{bound}}.\, \exists s \in S_i.\, e^i_k = s$, that spans the whole array dimension. No single selection variable may occur in two different dimensions, $\forall 1 \leq i \leq m \;\; k, l \in I_{\text{bound}}.\, e^i_k = e^i_l \implies k = l$. Where a selection variable indexes an array dimension, we assume, without loss of generality, that the same variable s_k is used across all edges in T, $\forall 1 \leq i, j \leq m \;\; k \in I_{\text{bound}}.\, e^i_k = e^j_k = s_k$, and write $\mathsf{selsub}(i \in I_{\text{bound}})$ for the corresponding variable. Let $\forall k \in I_{\text{bound}}.\, s_k \in S_{\text{sub}}$, be the set of all such variables.

As an example, consider $I_{\text{free}} = \{1, 3\}$, $I_{\text{bound}} = \{2, 4\}$, and,

$$\begin{aligned} E \;=\{ \;\; & (\{s, t\},\ \emptyset,\ l > 0,\ \langle 2, s, 2l - 1, t\rangle)\,, \\ & (\{s, t, u\},\ \emptyset,\ u \neq s \wedge o_u > 0,\ \langle 1, s, 2l, t\rangle) \quad \}\,, \end{aligned}$$

where s and t range over their entire respective dimensions. The guard expressions contain a free variable l, which is not bound by the selection set. The variables s and t appear in the same index position in all transitions. One of the transitions has a third selection binding, but it is only present in the guard.

The advantage of these restrictions is that no two different assignments to variables in S_{sub} specify the same channel. Thus, a limited form of action selection is permitted with only a minor extension to the basic techniques. The set of transitions is altered,

$$\begin{aligned} E' \;=\{ \;\; & \left(S_1 \setminus S_{\text{sub}},\ A_1,\ g_1,\ \langle e^1_1, \ldots, e^1_{n_C}\rangle \restriction_{I_{\text{free}}}\right), \ldots, \\ & \left(S_m \setminus S_{\text{sub}},\ A_m,\ g_m,\ \langle e^m_1, \ldots, e^m_{n_C}\rangle \restriction_{I_{\text{free}}}\right) \quad \}\,. \end{aligned}$$

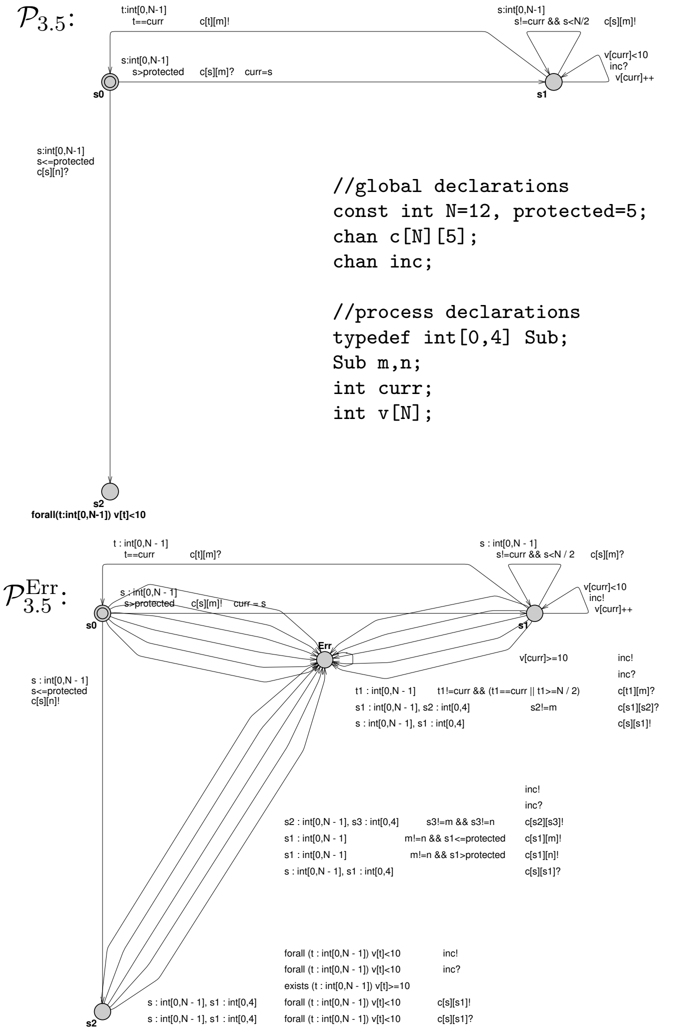
where we write $\langle i_1, \ldots, i_n\rangle \restriction_{I_{\text{free}}}$ to indicate a new sequence containing, in the same order, only those elements occurring at indices in I_{free} in the original sequence $\langle i_1, \ldots, i_n\rangle$. Since E' meets the stronger disjointedness assumptions, the techniques of previous sections may be applied to yield m' transitions,

$$\overline{E'} = \left\{ \left(\overline{S_1}, \overline{A_1}, \overline{g_1}, \langle e^1_1, \ldots, e^1_{|I_{\text{free}}|}\rangle\right), \ldots, \left(\overline{S_{m'}}, \overline{A_{m'}}, \overline{g_{m'}}, \langle e^{m'}_1, \ldots, e^{m'}_{|I_{\text{free}}|}\rangle\right) \right\},$$

to which the elements of S_{sub} may be returned,

$$\begin{aligned} \overline{E} \;=\{ \;\; & \left(\overline{S_1} \cup S_{\text{sub}}, \overline{A_1}, \overline{g_1}, \mathsf{inject}_{\mathsf{selsub}}\left(\langle e^1_1, \ldots, e^1_{|I_{\text{free}}|}\rangle\right)\right), \ldots, \\ & \left(\overline{S_{m'}} \cup S_{\text{sub}}, \overline{A_{m'}}, \overline{g_{m'}}, \mathsf{inject}_{\mathsf{selsub}}\left(\langle e^{m'}_1, \ldots, e^{m'}_{|I_{\text{free}}|}\rangle\right)\right) \quad \}\,. \end{aligned}$$

The inject function inserts variables from S_{sub} back into their original positions within the sequence.



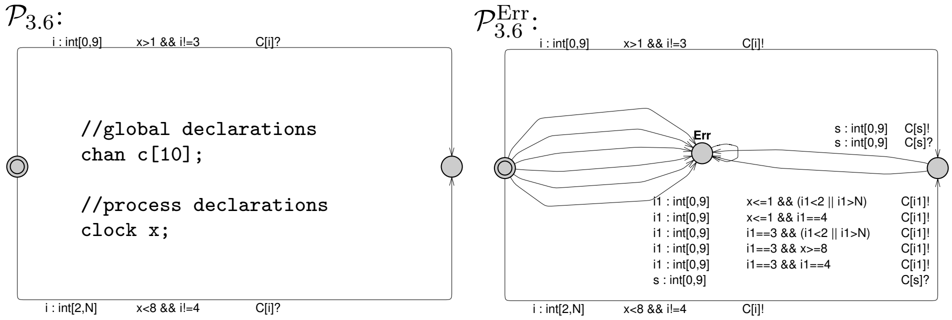
```
//global declarations
const int N=12, protected=5;
chan c[N][5];
chan inc;

//process declarations
typedef int[0,4] Sub;
Sub m,n;
int curr;
int v[N];
```

Figure 3.5: Example channel selections with limited bindings

In the example of Figure 3.5 selection bindings are used in channel selection. Note the transition from s_1 to s_0 in $\mathcal{P}_{3.5}$: it combines a selection binding with a guard to meet the restrictions outlined above, rather than the simpler equivalent $(\emptyset, \emptyset, \mathsf{true}, \langle curr, m\rangle)$ (where $curr$, a state variable, would otherwise clash with s, a selection variable, on the other transition leaving s_1); and the selection variable t differs from that, s, used on the other transition—both have been renamed in the edges to the error state of $\mathcal{P}_{3.5}^{\mathrm{Err}}$, to t_1, to satisfy the required assumptions.

The restriction of selection variable occurrences to at most one subscript dimension, excluding forms like $(\{s\}, A, g, \langle s, s\rangle)$, is easily circumvented by introducing additional selection bindings and appropriate constraints. For example, the previous transition becomes $(\{s, s'\}, A, s = s' \wedge g, \langle s, s' \rangle)$, then meeting the required assumptions. The implementation performs such manipulations automatically.


$\mathcal{P}_{3.6}$:

```
//global declarations
chan c[10];

//process declarations
clock x;
```

$\mathcal{P}_{3.6}^{\mathrm{Err}}$:

i : int[0,9]
x>1 && i!=3
C[i]?
i : int[2,N]
x<8 && i!=4
C[i]?
i : int[0,9]
x>1 && i!=3
C[i]!
Err
s : int[0,9] C[s]!
s : int[0,9] C[s]?
i1 : int[0,9] x<=1 && (i1<2 || i1>N) C[i1]!
i1 : int[0,9] x<=1 && i1==4 C[i1]!
i1 : int[0,9] i1==3 && (i1<2 || i1>N) C[i1]!
i1 : int[0,9] i1==3 && x>=8 C[i1]!
i1 : int[0,9] i1==3 && i1==4 C[i1]!
s : int[0,9] C[s]?
i : int[2,N]
x<8 && i!=4
C[i]!

Figure 3.6: Example of channel selections with differing ranges

The requirement for selection bindings used in channel selection to span the entire subscript range is not limiting. Given an integer binding s over the range $[l, u]$, where either or both bounds differ from those of the array dimension, the binding can be replaced with one over the full range provided the additional constraint $l \leq s \leq u$ is added to the transition guard. This technique is implemented in the current version. Figure 3.6 gives an example (note that the built-in simplifier does not eliminate the clause $i_1 = 3 \wedge i_1 = 4$).

The strict separation of selection variables from expressions over state variables is not so easily eliminated. When selection variables are allowed in non-elementary expressions, it becomes more difficult to determine which assignments resolve to the same channel, and thus to group expressions before negation. Certain forms of expressions preserve the property that each selection variable assignment specifies a different channel, for instance addition and multiplication simply shift the assignments, and correspondingly selection bounds, within the array. Whereas the modulo operator, shift operators, integer division, and function calls (for example, $f(x) = 1$) do not necessarily.

Allowing a mix of selection variables and expressions in a single channel array dimension across transitions also makes grouping transitions more difficult. The distinction is irrelevant if selection variables are allowed in compound expressions (not just single variables).

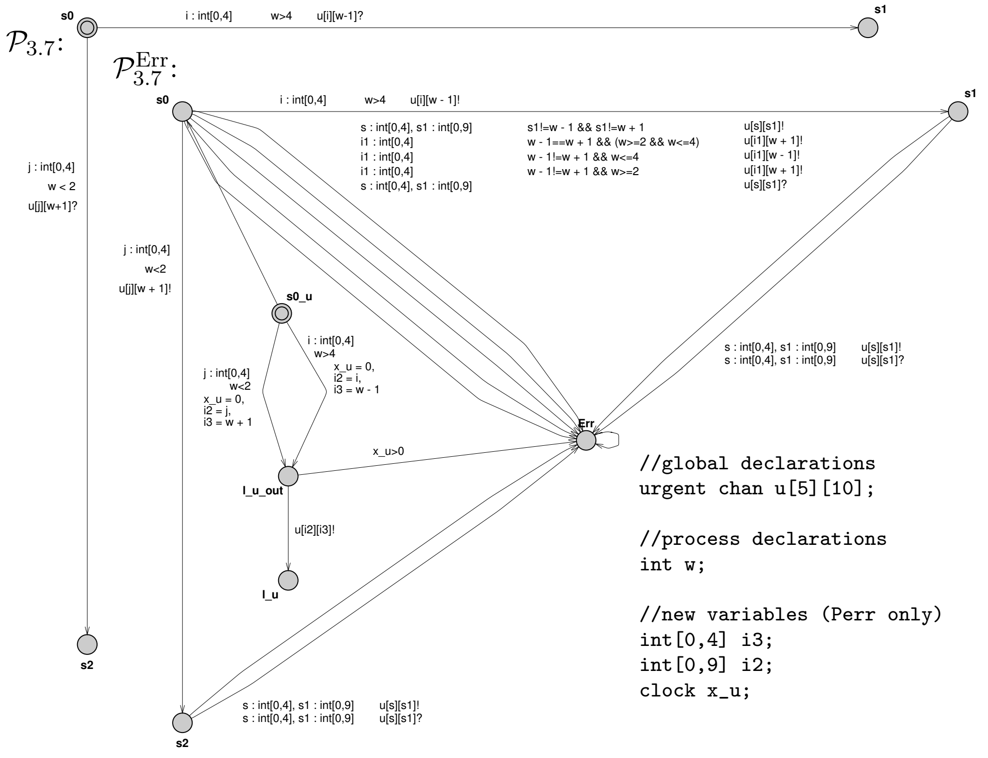
Augmenting the tool with features to overcome these limitations would increase its utility, but also its complexity; we currently favour relative simplicity.

3.3 Urgent channels and shared variables

A process is not usually obliged to synchronise on an enabled channel if further delay is permitted by the active location invariant. But Uppaal also allows channels to be marked *urgent*. Synchronisation on such channels, when they are enabled, must occur in preference to delay.

The basic notion of trace inclusion testing is insufficient when models contain urgent channels. When testing whether $\mathcal{P}_{\mathcal{I}} \sqsubseteq_{\mathrm{TR}} \mathcal{P}_{\mathcal{S}}$, we must ensure that $\mathcal{P}_{\mathcal{I}}$ can synchronise on a given urgent channel whenever $\mathcal{P}_{\mathcal{S}}$ can [JLS00]. The testing construction is extended by splitting states with outgoing urgent actions, joining the new state to the original with a τ-transition, and using an additional clock to detect illegal delays that are then directed to the error state [JLS00].

In Uppaal, processes may communicate by reading and writing shared variables. The testing construction is extended to check this behaviour by creating duplicate global variables in a $\mathcal{P}^{\mathrm{Err}}$ that shadow those in the original $\mathcal{P}$. At each state of $\mathcal{P}^{\mathrm{Err}}$ the real variables are compared to the copies [JLS00].



```
//global declarations
urgent chan u[5][10];

//process declarations
int w;

//new variables (Perr only)
int[0,4] i3;
int[0,9] i2;
clock x_u;
```

Figure 3.7: Testing for an array of urgent channels

Our implementation supports both extensions. Only a minor adjustment is necessary to support arrays of urgent channels: state variables are added to record index values in the two transition check for immediate synchronisation. Figure 3.7 shows an example. Since Uppaal provides direct array comparison, arrays of shared variable present no special challenges.

4 An implementation

We have developed a tool called *urpal* that is able to parse a saved Uppaal model and automatically generate the testing construction for a specified template. It implements the standard techniques [JLS00, Sto02] and the extensions described by this report.

The tool can only handle models

- that are deterministic,
- without committed nodes,
- where all edges are labelled (no τ transitions),
- and no broadcast channels are used, and,
- template parameters are not passed by reference.

Additionally, there are limitations on combinations of selection bindings and conjunction, as described in §3.1, and also with `forall` bindings, as described in §3.1. Channel array subscripts must be of the form stated in §3.2.

The tool does not provide any support for validating the assumption of determinism.

Whilst the tool currently focuses on the testing construction for trace inclusion, many of its subsystems are generic and could form the basis of a more general system for transforming Uppaal models. As an example, we have implemented an input enabling construction [Sto02, §A.1.4], features to prune transitions, and merge, drop, and rename nodes. Model transformations may be specified by a simple expression language over templates, sets and maps.

The tool produces models for Uppaal 4.0.6 and runs under Windows with Cygwin and Unix.

4.1 Overview

Urpal is written in Standard ML, a strongly-typed, garbage-collected, mostly functional language with formally defined semantics. Standard ML is ideal for applications involving complex symbol manipulation. Similar advantages are offered by the Haskell and Ocaml languages, as well as a greater number of libraries. Haskell has additional conveniences like type classes and list

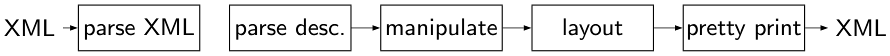


Figure 4.1: High-level structure of Urpal

comprehensions. In its favour, Standard ML has an excellent module system and is very stable, the language and core library will not change.

Figure 4.1 shows the five major subsystems of the tool: a parser for the XML-based format in which Uppaal saves models; a parser for the *description language* used for declarations, expressions and actions; algorithms for transforming models; an interface to Graphviz [GN00] for placing nodes and routing transitions; and a pretty printer back to the XML format.

The Uppaal developers distribute a separate C++ parsing library (libutap). It parses both the XML format and the description language, and performs type checking. While using the library potentially saves implementation effort and provides some insulation from changes to the XML file format, we decided that integrating the object-based API into Standard ML would involve as much work as writing a custom parser. Any such integration would depend on updates and binary releases from the Uppaal developers, and would complicate compilation and installation of our tool. Thus we wrote a custom parser that allows rudimentary access to type information, but otherwise assumes that input files have already passed validation by Uppaal.

Uppaal XML files are first parsed by FXP [Neu99], giving unparsed declarations and templates which are then processed by an ml-lex/ml-yacc parser for the description language. The result is an Uppaal model expressed in Standard ML data types. Extensive use is made of the SML Basis and SML/NJ libraries.

The manipulations performed by the tool can introduce many new transitions. A custom interface to the GraphViz `fdp` and `neato` tools makes an attempt at untangling introduced states and transitions while preserving any elements of the original structure. This processing make it easier to inspect and validate output from the tool, and increases confidence in its results. When trace inclusion verification fails, good formatting of the testing construction is almost essential to understand why. It is for similar reasons that we use the SML/NJ pretty printing library, based on Weis' Caml version, to output updated declarations and expressions.

Most of the figures in this report were produced directly by the tool, although transition label positions were sometimes adjusted slightly to improve readability. Some of the nodes and transitions in Figure 3.7 were placed manually.

5 Summary

This report has described a tool that supports trace inclusion verification for a subset of Uppaal models. Uppaal contains several features that make modelling convenient, but that complicate construction of test automata. We have addressed these by providing a more concrete formalisation than usual, and showing how successively more complicated edge forms can be addressed.

Uppaal restricts the form of expression guards to avoid splitting the symbolic clock zones used for verification. Unfortunately, these restrictions, in combination with certain modelling features and expression forms can prevent the effective construction of transitions needed to build test automata.

We believe our tool is still quite useful despite these limitations. It might be further improved by incorporating a more sophisticated term rewriting engine for simplification and equivalence testing, and by adding automatic or semi-automatic validation of the assumption of determinism.

Many of the implemented subsystems could potentially be reused for other transformations of Uppaal models, though more experience is needed to choose the most appropriate data structures and library functions.

6 Acknowledgements

This work was inspired by a pleasant afternoon with Frits Vaandrager, who recommended two of the key references, at Radboud University in Nijmegen.

This research was fully funded by National ICT Australia. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

Bibliography

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

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2004.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualisation system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):192–244, June 1994.

[JLS00] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In Mathai Joseph, editor, *Proc. 6th International Symposium on Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT '00)*, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30, Pune, India, September 2000. Springer-Verlag.

[KLSV06] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The theory of Timed I/O Automata.* Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2006.

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

[Neu99] Andreas Neumann. *Parsing and Querying XML Documents in SML.* PhD thesis, Universität Trier, Germany, December 1999.

[Sto02] Mariëlle I.A. Stoelinga. *Alea Jacta est: Verification of probabilistic, real-time and parametric systems.* PhD thesis, Katholieke Universiteit Nijmegen, The Netherlands, April 2002.