Automatically Transforming and Relating Uppaal Models of Embedded Systems

Timothy Bourke
School of CSE, University of NSW
and NICTA*
Sydney 2052, Australia
tbourke@cse.unsw.edu.au

Arcot Sowmya
School of CSE, University of NSW
Sydney 2052, Australia
sowmya@cse.unsw.edu.au

ABSTRACT

Relations between models are important for effective automatic validation, for comparing implementations with specifications, and for increased understanding of embedded systems designs. Timed automata may be used to model a system at multiple levels of abstraction, and timed trace inclusion is one way to relate the models.

It is known that a deterministic and τ-free timed automaton can be transformed such that reachability analysis can decide timed trace inclusion with another timed automaton. Performing the transformation manually is tedious and error-prone. We have developed a tool that does it automatically for a large subset of Uppaal models.

Certain features of the Uppaal modeling language, namely selection bindings and channel arrays, complicate the transformation. We formalize these features and extend the validation technique to incorporate them. We find it impracticable to manipulate some forms of channel array subscripts, and some combinations of selection bindings and universal quantifiers; doing so either requires premature parameter instantiation or produces models that Uppaal rejects.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems

General Terms

Design, Verification

Keywords

Timed trace inclusion, Uppaal, Model transformation

*NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council, in part through the Australian Government's *Backing Australia's Ability* initiative.

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Models help to design, simulate, verify, and synthesize embedded systems. Multiple models may be constructed for a system, describing it from different perspectives and at various abstraction levels. Details may be added to specify an implementation, or removed to make analysis practicable.

Timing details are important for correctness and precision in many embedded systems. It is often possible to model such systems as networks of timed (safety) automata [1, 2], which can be edited, simulated, and analyzed by tools like the Uppaal model checker [4], which is described in §2.

Two timed automata models are related by timed trace inclusion if all of the sequences of observable events and delays of one are also possible in the other. Such relationships are important for comparing implementation models to abstract specifications, or when creating successive abstractions to make verification feasible [3, 7]. Employing either technique requires validation of the inclusion relation. Timed trace inclusion is similar to timed language inclusion, which is undecidable in general [1, Corollary 5.3], but PSPACE-complete if the specification automaton is deterministic [1, Theorem 6.6]. The decidable case can be verified in Uppaal by constructing a validation automaton and performing reachability analysis. An example is given in §1.2.

Constructing validation automata manually can be tedious and error prone; analysis can improve models, which then require updated constructions. We have, thus, developed a tool, §4, to do it automatically. It turns out that features of the Uppaal (Classic; version 4.0.6) language for creating succinct models are challenging to manipulate even though they are not fundamentally more expressive.

The main contribution of this paper is contained in §3. It shows how, when possible, the existing validation construction can be extended to incorporate advanced Uppaal features while preserving concision and parameterization. We develop a formalization that is more concrete than usual to explain and justify our technique.

1.1 Related Work

Trace inclusion between two models can be verified by showing refinement relations between their states. Lynch and Vaandrager give a thorough review and references [5], and also consider models with time [6].

Timed trace inclusion validation via reachability analysis is discussed by both Jensen, Larsen, and Skou [3], who consider urgent channels and shared variables, and Stoelinga [7], whose exposition we adapt to handle channel arrays, quantifiers, and selection bindings.

1.2 Example: A liberal railway controller

Uppaal is distributed with an idealized railway controller model [8]. Imagine a bridge where several lines converge temporarily to a single track. A controller receives signals as trains approach and leave the bridge. It can signal a train to stop provided it does so quickly enough; another signal is sent when it is safe to continue. Two properties must be satisfied. Safety: only one train is on the bridge at a time, and liveness: all approaching trains eventually leave.

The model is shown in Figures 1a, 1b, and 1c. The controller has been modified so that go is not urgent, because although the technique and tool can handle such channels, combining them with invariants, as we do, can give validation automata with clock guards that Uppaal rejects. Committed locations are not addressed and have been made urgent instead. The allow_approach[front()]! action was added to make observable the decision to let an approaching train pass; the Silent automaton ensures that it is never blocked.[1]

In the complete model, no train ever sends two consecutive approach signals; we make this assumption explicit, by adding an extra guard on the approach[e]? transition.[2]

The model is parameterized over the number of trains N. Each train is identified by an integer which selects an element from the channel arrays connected to the controller.

The original controller stops trains as they arrive and adds their identifiers to a queue, from which they are resumed in order. Suppose a more flexible specification is desired, one that permits controllers that give preference to moving trains, or prioritize certain tracks, or delay decisions for longer, while still never stopping trains indefinitely. A candidate solution is given in Figure 1d. It maintains a two-dimensional array with, for each train, a status—AWAY, APPR, STOP, or GO—and, if it is stopped, the number of times trains on each other track have been allowed through. It keeps a timer for each train, reset on approach, that monitors the deadline for sending a stop signal. The model uses selection bindings, the i : id_t expressions, that stand for multiple transitions, one for each i value. Guard and invariant expressions contain universal, **forall** (j : id_t), and existential, **exists** (k : id_t), quantifiers.

Suppose we want to know if the new controller $\mathcal{N}$ allows all behaviors of the original $\mathcal{O}$. Since $\mathcal{N}$ is deterministic, it can be transformed into a validation automaton $\mathcal{N}^{\mathsf{Err}}$, Figure 1e, with a new location Err and where actions are inverted, for example go[i]! becomes go[i]?, transition guards are the conjunctions of the original guards and location invariants, which are removed, and from each location a transition to Err is added for everything that cannot happen. Timed trace inclusion validation is verified by model-checking:

$$\mathcal{O} \parallel \mathcal{N}^{\mathsf{Err}} \models \mathsf{A}\Box\, \neg \mathcal{N}^{\mathsf{Err}}.\mathsf{Err}.$$

If there were a trace of $\mathcal{O}$ that was not a trace of $\mathcal{N}$, then $\mathcal{O}$ would eventually offer an action that caused $\mathcal{N}^{\mathsf{Err}}$ to enter Err; Uppaal would present it as a counter-example.

The channel arrays, selection bindings, and quantifiers in $\mathcal{N}$ are not directly handled by the basic construction. Whilst they could be unravelled for fixed parameter values, the result could be difficult to examine, making counter-example traces hard to follow, and more difficult to model-check. This paper shows how to treat such features directly.

[1] A broadcast channel cannot be used: §2.

[2] The results hold if a self-loop with no update is added.

2. UPPAAL

An Uppaal model comprises multiple communicating components. Each is typically presented as a (timed safety) automaton, but a more concrete formalization is required to describe and understand the extended validation technique. In this section we introduce a process model with explicit variable valuations, selection bindings, and channel arrays.

Several Uppaal features are excluded from our model.

Urgent nodes, where delays cannot occur, can be removed from models by adding the conjunct $x_u \leq 0$ to their invariants, where x_u is a new clock that is reset on every edge.[3]

Communications on *urgent channels* and through *shared variables* are not formalized, but they are treated in §3.4.

Committed nodes are used to express the atomicity of action sequences, edges leaving them have priority over those leaving normal nodes. Process and transition *priorities* also give preference to some edges over others. Neither feature is addressed by this paper.

Output actions on *broadcast channels* synchronize with all enabled input actions on the same channel, or, if there are none, may occur alone. We do not treat broadcast inputs as they become validation automata outputs that may always occur irrespective of how a model being validated behaves. Broadcast outputs, however, become inputs in the validation automaton and are treated normally. They could potentially validate whether a pattern of synchronizations between two input-enabled automata, each relying on assumptions embodied in the other, is timed trace included in a protocol model. But, unfortunately, Uppaal rejects transitions on broadcast inputs with clock variables in guards, and as clock variables are allowed on output transitions and in location invariants, the technique is of limited use.

2.1 Variables, expressions, and valuations

Some preliminary definitions and notation are required before a process can be defined.

Uppaal models may contain clock variables, which express timing constraints, and data variables, which facilitate concise and flexible process models.

Data variables are typed, but we will ignore, without loss of generality, boolean variables, record types, and arrays of anything but channels. The set of all variables Vars encompasses the two remaining categories of types: bounded sets of integers, written $[l, u]$ where $l \leq u$ and $\forall i \in \mathbb{Z}.\, l \leq i \leq u$ implies $i \in [l, u]$, and finite scalar sets, written [S]. Both integer and scalar types can index arrays, and be bound by quantifiers in expressions and by selection bindings over process edges. Scalar variables may only be directly assigned to one another and compared for equality, which permits model-checking optimizations. Each scalar declaration, for example, **scalar** [5] ids, gives a new disjoint set. Type aliases, stated with **typedef**, preserve set identity.

Expressions are built from variables, constants, function calls, operators, relations, and quantifiers. We treat them as members of a set Exprs. The set of unbound variables in $e \in \mathsf{Exprs}$ is written $\mathsf{freevars}(e)$. Each expression e denotes either a truth value, an integer, or a value from a scalar set. Its value depends on the values assigned to variables in $\mathsf{freevars}(e)$. A valuation val_V gives a value $\mathsf{val}_V(v)$ of appropriate type for each variable v in the finite set V. The set of all valuations for a given set V is written Vals_V.

[3] The implementation automatically expands urgent nodes.

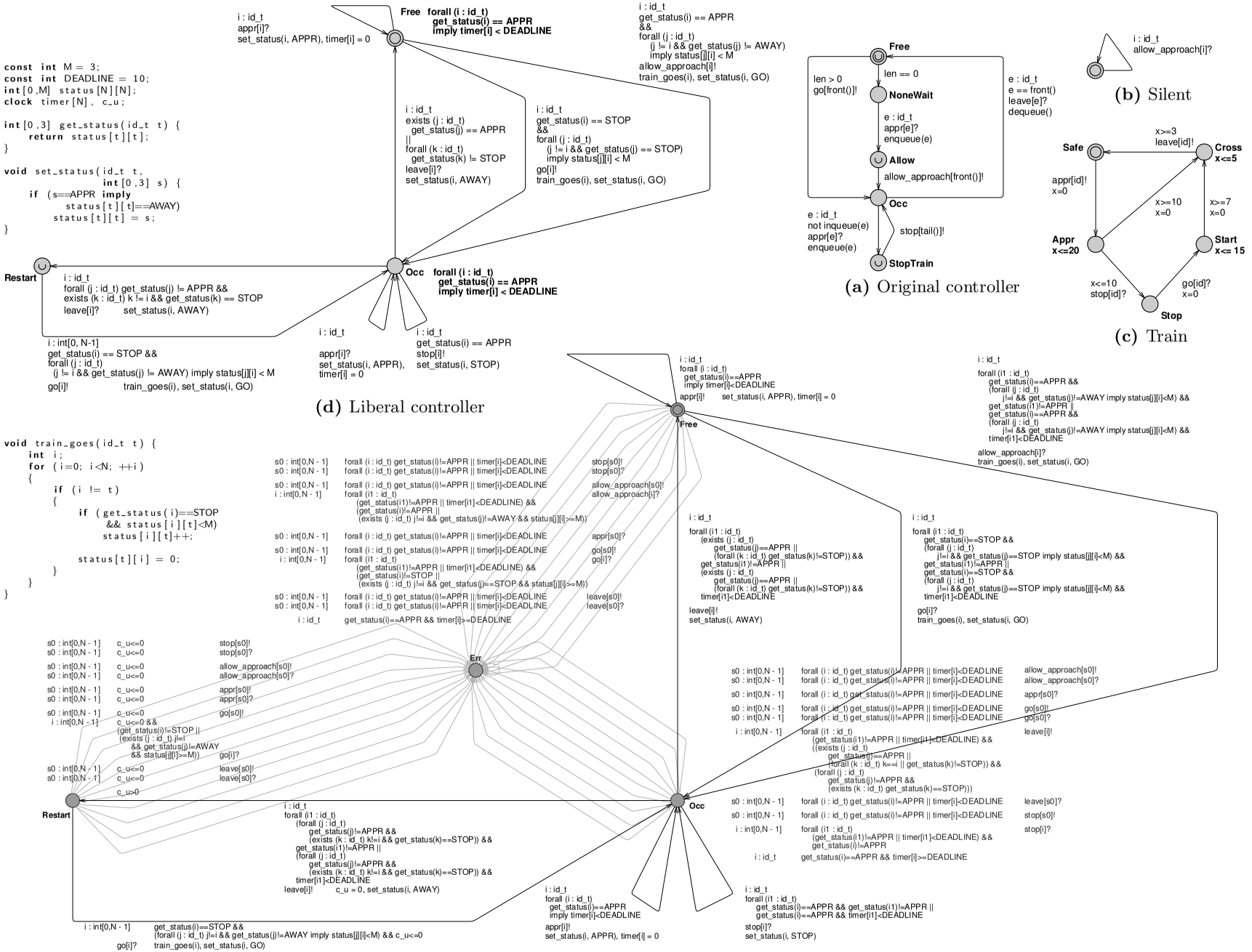

Figure 1: Railway controller example

Valuations may be composed:

$$\mathsf{val}_{V_1} \triangleright \mathsf{val}_{V_2}(v) = \mathsf{val}_{V_2}(v) \text{ if } v \in V_2, \text{ otherwise } \mathsf{val}_{V_1}(v).$$

The value of expression e with respect to valuation val_V is written $[\![e]\!]_{\mathsf{val}_V}$, provided $\mathsf{freevars}(e) \subseteq V$.

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

The set of update functions is written Δ_V, an element δ_V maps one valuation val_V to another val'_V. Clock variables may only become 0. The set of clock variables reset by an update δ_V is written $\mathsf{resets}(\delta_V)$.

2.2 Channels and actions

Channel arrays are convenient for specifying a parameterizable number of channels; a sequence of expressions can select a channel for synchronization. The controller of Figure 1a uses channel arrays to communicate with N trains. A precise notation is necessary to define their transformation.

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 either of the form $[l, u]$, or [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 appropriate types. Non-array, *elementary*, channels are denoted by singleton channel sets, for example C_ϵ. Two *actions* are associated with each channel. To each channel set C are associated sets of input and output actions,

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

We write $^?/_!$ for either direction, used consistently in a rule, and $^!/_?$ for its inverse.

For $\mathcal{C} \subseteq \mathsf{Chansets}$, let $\mathcal{C}? = \bigcup_{C\in\mathcal{C}} C?$ and $\mathcal{C}! = \bigcup_{C\in\mathcal{C}} C!$. We define action sets $\mathsf{Acts} = \mathsf{Chansets}? \dot\cup \mathsf{Chansets}!$ and $\mathsf{Acts}_\tau = \mathsf{Acts} \dot\cup \{\tau\}$ with disjoint union and the silent action τ. The inverse of an action $a \neq \tau$, is written $\overline{a}$; if $a = c?$ then $\overline{a} = c!$ and vice versa; similarly for action sets.

Subsets of a channel set C may be designated by an expression sequence $\langle e_1, \ldots, e_{n_C}\rangle$. Evaluation is lifted to such designations to specify a single channel from the subset,

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

2.3 Processes

Transition guard expressions in Uppaal are restricted in form for efficient manipulation as symbolic zones. They are built on clock terms.

Def. 2.1. *The set of* clock terms $T_{clk}(K, V)$, $K \cap V = \emptyset$, *is the smallest containing* **1.** p_{nclk} *where* $\mathsf{freevars}(p_{nclk}) \subseteq V$; **2.** $k\ R\ e$ *where* $k \in K$, $R \in \{<, \leq, =, \geq, >\}$, $\mathsf{freevars}(e) \subseteq V$; *and* **3.** $k_1 - k_2\ R\ e$ *where* $k_1, k_2 \in K$, *and* $\mathsf{freevars}(e) \subseteq V$.

Form 1 admits boolean-valued, clock-free expressions. In form 2 a clock variable is compared with an integer-valued, clock-free expression, in form 3 the difference of two clock variables is compared.

Def. 2.2. *The* guard expressions *on K and V, $E_g(K,V)$, where K and V are disjoint sets of clock and non-clock variables respectively, is the smallest set built by the rules:*

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

Uninterpreted clock-free expressions p_{nclk} may contain existential quantifiers and disjunctive sub-terms, but guard expressions may not. Invariant expressions $E_{\mathsf{inv}}(K, V)$ are the subset of guard expressions where comparisons in clock terms are restricted to $R \in \{<, \leq\}$.

There is now sufficient background to formalize processes as modeled directly within Uppaal.

Def. 2.3. *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* $\mathsf{inv}_{V\cup K}$*, mapping from N to expressions in $E_{\mathsf{inv}}(K, V)$. The labeled edges connect pairs of nodes, where* $S \subseteq \mathsf{Vars}$*,*

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

For $(n, S, e, C[e_1, \ldots, e_{n_C}]^?/_!, \delta_{V\cup K}^{(V,S)}, n') \in E$ *we write*

$$n \xrightarrow[C[e_1,\ldots,e_{n_C}]^?/_!]{S\ \ e\ \ \delta_{V\cup K}^{(V,S)}}_E n',$$

similarly for τ-transitions, where n and n' are, respectively, source and destination nodes; S is a finite set of selection bindings*, $S \cap V = \emptyset$; $\mathsf{freevars}(e) \subseteq V \cup S$; $\delta_{V\cup K}^{(V,S)}$ is an update for valuations over V and K that depends on a valuation of V and S; and the action set is either $\{\tau\}$, or consistent in name and direction, that is, it has the form $C[e_1, \ldots, e_{n_C}]^?/_!$ where $\mathsf{freevars}(C[e_1, \ldots, e_{n_C}]) \subseteq V \cup S$.*

2.4 Automata

Uppaal models are normally formalized as automata.

Def. 2.4. *An automaton* $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ *on* Acts_τ *comprises finites sets of locations L and clocks K, an initial location l_0, a function inv_K from L to $E_{\mathsf{inv}}(K, \emptyset)$, and a set of transitions $T \subseteq L \times E_g(K, \emptyset) \times \mathsf{Acts}_\tau \times 2^K \times L$. A transition $(l, e, a, R, l') \in T$ is written $l \xrightarrow[a]{e\ \ R}_T l'$.*

We define a function Γ_V to map processes to automata by expanding references to non-clock variables. It gives the semantics of the process modeling notation. We write $[\![e]\!]_{\mathsf{val}_V}$ for a partial evaluation that maps one expression to another such that $[\![\, [\![e]\!]_{\mathsf{val}_V} \,]\!]_{\mathsf{val}_{V'}} = [\![e]\!]_{\mathsf{val}_{V'} \triangleright \mathsf{val}_V}$ if $\mathsf{freevars}(e) \backslash V \subseteq V'$.

Def. 2.5. *Given a* $\mathcal{P} = (N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V\cup K}, E)$*, let* $\Gamma_V(\mathcal{P}) = (L, l_0, K, \mathsf{inv}_K, T)$*, where* $L = N \times \mathsf{Vals}_V$*,* $l_0 = \left(n_0, \mathsf{val}_V^{\mathrm{init}}\right)$*,* $\mathsf{inv}_K\left((n, \mathsf{val}_V)\right) = [\![\, \mathsf{inv}_{V\cup K}(n) \,]\!]_{\mathsf{val}_V}$*, and T is the smallest relation satisfying:*

$$\frac{n \xrightarrow[C[e_1,\ldots,e_2]^?/_!]{S\ \ e\ \ \delta_{V\cup K}^{(V,S)}}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[[\![C[e_1,\ldots,e_{n_C}]^?/_!]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S}]{[\![e]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S}\ \ \mathsf{resets}\left(\delta_{V\cup K}^{(V,S)}\right)}_T \left(n', \delta_{V\cup K}^{(V,S)}(\mathsf{val}_V)\right)}\ ?, !$$

$$\frac{n \xrightarrow[\tau]{S\ \ e\ \ \delta_{V\cup K}^{(V,S)}}_E n' \quad \mathsf{val}_V \in \mathsf{Vals}_V \quad \mathsf{val}_S \in \mathsf{Vals}_S}{(n, \mathsf{val}_V) \xrightarrow[\tau]{[\![e]\!]_{\mathsf{val}_V \triangleright \mathsf{val}_S}\ \ \mathsf{resets}\left(\delta_{V\cup K}^{(V,S)}\right)}_T \left(n', \delta_{V\cup K}^{(V,S)}(\mathsf{val}_V)\right)}\ \tau$$

The three rules above differ only in the action type. Automata locations are formed of process nodes paired with non-clock variable valuations. After fixing a variable valuation, each process edge may still expand to multiple transitions, one for every valuation val_S of the selection bindings S. The original guard is partially evaluated—the result

may depend on clock variables—against binding and variable values. The combined binding and variable values affect the update of the destination valuation, and, in the rules for ? and !, select an element from the channel set. Updates to clocks remain on the result transitions.

Later manipulations work from the observation that every edge corresponds to a set of transitions determined by the combination of control node and data valuation, and all possible assignments to S.

An Uppaal model of n processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ over Acts_τ and variables V can be mapped to a *closed system* automaton

$$\mathcal{A}_\tau = (\Gamma_V(\mathcal{P}_1) \parallel \ldots \parallel \Gamma_V(\mathcal{P}_n)) \setminus \mathsf{Acts}$$

We will not repeat definitions for parallel composition and restriction. Parallelism follows CCS, complementary actions synchronize to τ-transitions; rules are added for urgent and broadcast channels, and shared variables. Restriction prunes transitions *open* to synchronization on a set of actions.

2.5 Validation automata

We repeat the validation construction definition [7, §A.1.5] for automata. It will be lifted to processes in §3.

DEF. 2.6. *For a deterministic* $\mathcal{A} = (L, l_0, K, \mathsf{inv}_K, T)$ *over* Acts *let* $\mathcal{A}^{\mathsf{Err}} = (L \dot\cup \{\mathsf{Err}\}, l_0, K, \mathsf{inv}_K^{\mathsf{Err}}, T^{\mathsf{Err}})$ *where* $\mathsf{inv}_K^{\mathsf{Err}}(l \in L) = \mathsf{true}$, T^{Err} *is the smallest relation such that*

$$\frac{}{l \xrightarrow[\tau]{\neg \mathsf{inv}_K(l) \;\; \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\,1 \qquad \frac{a \in \mathsf{Acts}}{\mathsf{Err} \xrightarrow[a]{\mathsf{true} \;\; \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\,2$$

$$\frac{l \xrightarrow[a]{g \;\; R}_T l'}{l \xrightarrow[\overline{a}]{(g \wedge \mathsf{inv}_K(l)) \;\; R}_{T^{\mathsf{Err}}} l'}\,3 \qquad \frac{g_{(a,l)} = \neg \bigvee \{\, g \mid l \xrightarrow[a]{g \;\; R}_T l' \,\}}{l \xrightarrow[\overline{a}]{(g_{(a,l)} \wedge \mathsf{inv}_K(l)) \;\; \emptyset}_{T^{\mathsf{Err}}} \mathsf{Err}}\,4$$

If there are no transitions for a pairing of action and location (a, l) the upper part of rule 4 becomes $\neg \bigvee \emptyset = \mathsf{true}$, giving a transition directly to the Err state.

The validation construction can only be applied to τ-free, deterministic automata, as timed trace inclusion is not decidable for nondeterministic specifications. It is sometimes possible to remove nondeterminism by relabeling [7, §7.5.2, Appendix A], τ-transitions may be similarly dispatched.

The railway controller, for example, was relabeled, replacing τ with $\mathsf{allow\_approach}[\mathsf{i}]!$ actions. Without observable actions, the validation automaton could remain in Free, when the gate being validated silently entered Occ, whence it could perform actions leading the former to Err. Were a single $\mathsf{allow\_approach}!$ action used, the models would be nondeterministic, and although initially synchronized, each could change its variables differently and later behave differently.

3. TRANSFORMING UPPAAL MODELS

The validation construction for automata was given in Definition 2.6. Given a process $\mathcal{P}$ our aim is to analogously define how to construct another process $\mathcal{P}^{\mathsf{Err}}$ such that:

$$\Gamma_V(\mathcal{P}^{\mathsf{Err}}) \approx \Gamma_V(\mathcal{P})^{\mathsf{Err}}, \qquad \begin{array}{ccc} \mathcal{P} & \xrightarrow{\text{Definition 3.1}} & \mathcal{P}^{\mathsf{Err}} \\ \big\downarrow \Gamma_V & & \big\downarrow \Gamma_V \\ \mathcal{A} & \xrightarrow{\text{Definition 2.6}} & \big[\mathcal{A}^{\mathsf{Err}}\big] \end{array}$$

where $\mathcal{A}_1 \approx \mathcal{A}_2$ iff 1. $\mathcal{A}_1$ has a location Err, 2. as does $\mathcal{A}_2$, and 3. $\forall \mathcal{A}\,.\,(\mathcal{A}_1 \parallel \mathcal{A} \models \mathsf{A}\Box\, \neg \mathcal{A}_1.\mathsf{Err}) \Leftrightarrow (\mathcal{A}_2 \parallel \mathcal{A} \models \mathsf{A}\Box\, \neg \mathcal{A}_2.\mathsf{Err})$.

The equivalence class is introduced as the validation construction for processes may add additional states and transitions to handle unexpanded features.

In pragmatic terms, we want to correctly extend the original construction to allow direct timed trace inclusion validation of models, leaving explicit expansion, Γ_V, to Uppaal.

DEF. 3.1. *Let* $\mathcal{P} = (N, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V \cup K}, E)$ *where the underlying automaton* $\Gamma_V(\mathcal{P})$ *is deterministic and* τ*-free, then* $\mathcal{P}^{\mathsf{Err}} = (N \dot\cup \{\mathsf{Err}\}, n_0, K, V, \mathsf{val}_V^{\mathrm{init}}, \mathsf{inv}_{V \cup K}^{\mathsf{Err}}, E^{\mathsf{Err}})$, *where* $\mathsf{inv}_{V \cup K}^{\mathsf{Err}}(l \in L) = \mathsf{true}$ *and* E^{Err} *is the least relation such that,*

$$\frac{}{n \xrightarrow[\tau]{\emptyset \;\; \neg \mathsf{inv}_{V \cup K}(n) \;\; \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\,1 \qquad \frac{a \in \mathsf{Acts}}{\mathsf{Err} \xrightarrow[a]{\emptyset \;\; \mathsf{true} \;\; \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\,2$$

$$\frac{n \xrightarrow[C[e_1,\ldots,e_{n_C}]^?/_!]{S \;\; g \;\; \lambda}_E n'}{n \xrightarrow[C[e_1,\ldots,e_{n_C}]\,!/_?]{S \;\; (g \wedge \mathsf{inv}_{V \cup K}(n)) \;\; \lambda}_{E^{\mathsf{Err}}} n'}\,3$$

$$\frac{\begin{array}{l} n \in N \quad C \in \mathsf{Chansets} \quad (S', g', \langle e_1', \ldots, e_{n_C}' \rangle) \in \mathsf{flip}(C, T) \\ \textit{where } T = \left\{ (S, g, \langle e_1, \ldots, e_{n_C} \rangle) \mid n \xrightarrow[C[e_1,\ldots,e_{n_C}]^?/_!]{S \;\; g \;\; \cdot}_E \cdot \right\} \end{array}}{n \xrightarrow[C[e_1',\ldots,e_{n_C}']\,!/_?]{S' \;\; (g' \wedge \mathsf{inv}_{V \cup K}(n)) \;\; \cdot}_{E^{\mathsf{Err}}} \mathsf{Err}}\,4$$

The function flip *maps a set of triples—selection bindings, guards, and subscript expressions—of edges for a fixed location* l *and channel set* C*, to another set of triples so as to satisfy* $\Gamma_V(\mathcal{P}^{\mathsf{Err}}) \approx \Gamma_V(\mathcal{P})^{\mathsf{Err}}$.

Rule 4 applies to node/action set pairs, mapping, via the flip function, all associated edges to another set of edges to the Err state. The flip function encapsulates the treatment of selection bindings and channel arrays. It is presented incrementally over the following subsections. The presentation includes details that, while not required for the abstract definition, are important for implementation.

3.1 Elementary channels

In this section we consider edges labeled with singleton channel sets; selection bindings are thus limited to guards.

3.1.1 No selection binding or quantifiers

In the absence of selection bindings and quantifiers over expressions containing clocks, the challenge is to produce guard expressions that meet the syntactic restrictions of Uppaal. Ideally, expressions are simplified when possible.

Clock expressions represent the intermediate results of manipulations. They must be converted to guard expressions, $E_g(K, V) \subseteq E_{clk}(K, V)$, for acceptance by Uppaal.

DEF. 3.2. *The set of* clock expressions $E_{clk}(K, V)$ *over clock* K *and non-clock variables* V *is the smallest such that*

$$\frac{p \in T_{clk}(K, V)}{p \;\in E_{clk}(K, V)}\,1$$

$$\frac{p, q \in E_{clk}(K, V)}{p \wedge q \;\in E_{clk}(K, V)}\,2 \qquad \frac{p, q \in E_{clk}(K, V)}{p \vee q \;\in E_{clk}(K, V)}\,3$$

$$\frac{p \in E_{clk}(K, V) \quad v \in \mathsf{Vars}}{\forall v.\, p \;\in E_{clk}(K, V)}\,4 \qquad \frac{p \in E_{clk}(K, V) \quad v \in \mathsf{Vars}}{\exists v.\, p \;\in E_{clk}(K, V)}\,5$$

For now, we limit our attention to *quantifier-free clock expressions*, that is those formed without using rules 4 and

5. Quantifiers may nevertheless appear in clock terms $p \in T_{clk}(K, V)$ where they do not encompass clock variables.

Quantifier-free clock expressions are closed under negation. The function neg is defined over the structure of expressions, after the pattern:

$$\mathsf{neg}(c < e) = c \geq e, \qquad \mathsf{neg}(c = e) = c < e \vee c > e,$$
$$\mathsf{neg}(p_{nclk}) = \mathsf{neg}_{nclk}(p_{nclk}), \qquad \mathsf{neg}(p \vee q) = \mathsf{neg}(p) \wedge \mathsf{neg}(q).$$

where $\mathsf{neg}_{nclk}(p_{nclk})$ negates clock-free expressions p_{nclk}.

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

The set of edges $\overline{E} = \mathsf{flip}(E)$ should contain guards such that at least one is true when all of the guards in E are false. We directly mimic the premise of rule 4 of Definition 2.6 by forming $\mathsf{neg}(g_1 \vee \cdots \vee g_m)$. To ensure that the resulting expression conforms to Uppaal's syntactic restrictions, 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 $\overline{E} = \{\overline{g}_1, \ldots, \overline{g}_{m'}\}$.

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 $c < 4$. Simplification is not strictly necessary, but it improves readability which, in turn, increases confidence in the results, and makes counter-example traces easier to follow. Our implementation uses simple syntactic criteria to assess, for a pair of terms in a conjunctive clause, whether one implies or contradicts the other. A possible improvement would be to exploit a heavy-duty simplifier, as in theorem proving tools.

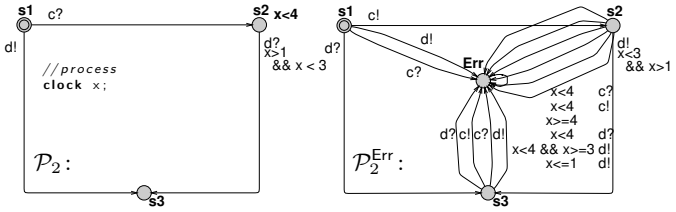

Figure 2: No selection bindings or quantifiers

Figure 2 shows a process $\mathcal{P}_2$ that has one clock x and synchronizes on channels c and d, and the corresponding validation process $\mathcal{P}_2^{\mathsf{Err}}$. The two transitions from s_1 of $\mathcal{P}_2^{\mathsf{Err}}$ to Err are labeled with inverses of the actions not leaving that state. Transitions from s_2 of $\mathcal{P}_2^{\mathsf{Err}}$ have more involved guards

$c?, c!, d?$	$x < 4$	guard is node invariant
τ	$x \geq 4$	negated invariant
$d!$	$x < 3 \wedge x > 1$	original guard
$d!$	$x < 4 \wedge x \geq 3$	invariant, half negated guard
$d!$	$x \leq 1$	other half of negated guard

In the third and fifth lines above, simplification has removed the redundant invariant conjunct. The negated guard of $d!$ is split over two transitions to avoid disjunction.

3.1.2 With selection bindings

A selection binding pairs a variable name with a bounded integer or scalar type. Multiple names may be bound over the expression, update statement, and action array indices of an edge. The latter are not considered until §3.2.2.

An edge with selection bindings represents multiple transitions in the unwound automaton, even after fixing the values of state variables. This is apparent in Definition 2.5. Any choice of values for the selection bindings that satisfies the guard represents a possible transition.

The 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. We assume that the selection sets are pairwise disjoint, without loss of generality since elements may be renamed if necessary. For now, we only consider quantifier-free g_i.

A transition for fixed location and action is enabled when

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

where $S_i = \{s_{i1}, \ldots, s_{in_i}\}$, 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 no guard is satisfied for any valuation of the selection bindings. 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. One solution is to eliminate problematic quantified variables by creating a new edge for each of their possible values and every disjunct in scope. This is not possible for variables that take values from a scalar set, or from bounded integers where either of the bounds is an expression that cannot be reduced to a concrete value, for instance those involving template arguments. We will look at another construction in the next section that can also be applied to non-scalar types. Our current implementation warns when a negated expression cannot be split into separate transitions and is thus likely to be rejected by Uppaal.

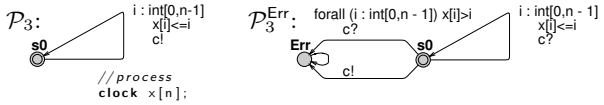

Figure 3: Selection bindings but no quantifiers

In $\mathcal{P}_3^{\mathsf{Err}}$ of Figure 3 a transition from s_0 to Err occurs on $c?$ when the negated guard is true for all possible values of i.

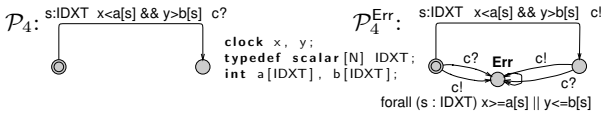

Figure 4: Selection bindings/negated guard clash

Another process and corresponding validation process are shown in Figure 4. The guard disjuncts of $\mathcal{P}_4^{\mathsf{Err}}$ cannot be split into separate transitions, because of the **forall**; this process is rejected by Uppaal.

3.1.3 With universal quantifiers

The previous section showed how universal quantifiers are introduced when negating transitions with selection bindings. We now consider guards that already contain universal quantifiers (rule 5 of Definition 3.2). Guards with existential quantifiers are rejected by Uppaal, and hence not considered.

Quantifier bindings, like selection bindings, bind a name and finite type over a subexpression. A universally quantified expression $\forall i \in [l,u].\, e(i)$ is equivalent to a sequence of conjunctions $e(l) \wedge \cdots \wedge e(u)$, and similarly for scalar types.

As it is possible to convert guard expressions into prenex normal form where all the quantifiers are universal, and thus order is irrelevant, a set of m edges is now 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, renaming if necessary, and further that quantified variables only occur in corresponding guard expressions, that is, for all $1 \le i, j \le m$, $S_i \cap A_j = \emptyset$, and if $i \neq j$ then $S_i \cap S_j = \emptyset$, $A_i \cap A_j = \emptyset$, $A_i \cap \mathsf{freevars}(g_j) = \emptyset$, and $S_i \cap \mathsf{freevars}(g_j) = \emptyset$. These assumptions can be met by renaming as required.

A transition for a given action is enabled whenever

$$\exists s_{1_1}, \ldots, s_{m_{n_m}}.\, \forall a_{1_1}, \ldots, a_{m_{n'_m}}.\, g_1 \vee \cdots \vee g_m.$$

Thus, there are no transitions enabled for the action when

$$\forall s_{1_1}, \ldots, s_{m_{n_m}}.\, \exists a_{1_1}, \ldots, a_{m_{n'_m}}.\, \mathsf{neg}(g_1 \vee \cdots \vee g_m), \quad (\psi_1)$$

which is problematic because Uppaal will reject it if any of the guards contain clock variables. One solution is to rearrange the expression, if possible, into the form

$$\exists a_{1_1}, \ldots, a_{m_{n'_m}}.\, \forall s_{1_1}, \ldots, s_{m_{n_m}}.\, \mathsf{neg}(g_1 \vee \cdots \vee g_m), \quad (\psi_2)$$

where existential bindings in the prefix could then be converted into selection bindings, the remainder of the expression having the form discussed in §3.1.2 and subject to the same limitations and treatment.

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

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

The minimal condition is false which rejects all processes with guards that mix selections and quantifiers over clock variables. Logical equivalence of ψ_1 and ψ_2 is the most accepting. We could, for example, emit constraints and proof obligations for treatment in a theorem prover or a model checker. Currently we implement an approximate condition.

DEF. 3.3. *The* canswap *predicate is defined on formulas of form* $\forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \varphi_1 \vee \cdots \vee \varphi_l$*, where each* φ_i *is a conjunction of clock terms,* $p_i^1 \wedge \cdots \wedge p_i^{n_i}$*. Let*

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

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

PROPOSITION 1. *For a quantifier-free formula* ψ *in DNF,* $\mathsf{canswap}(\forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \psi)$ *implies*

$$\forall a_1, \ldots, a_n.\, \exists e_1, \ldots, e_m.\, \psi \equiv \exists e_1, \ldots, e_m.\, \forall a_1, \ldots, a_n.\, \psi$$

The $\mathsf{canswap}$ 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.

An alternative to rearranging formulas, when none of the universal quantifiers are of scalar type, is shown in Figure 5. In the committed location, meta variables $s_{1_1} \ldots s_{m_{n_m}}$, shadowing universal quantifiers, are iterated over all valuations provided assignments to selection bindings $a_{1_1} \ldots a_{m_{n'_m}}$ can be found to satisfy one of the disjunctive clauses $\varphi_1 \ldots \varphi_{m'}$. If this is not possible, the construction returns to the original location, otherwise it offers a synchronization into Err.

3.2 Channel arrays

We now generalize the techniques of the previous subsection by considering edges labeled with actions on elements of channel arrays. The central challenge is to decide when two index sequences specify the same channel. For singleton channel sets, channels may be identified by the set name alone.[4] An array name, however, is usually shared by multiple channels. Individual elements are selected by sequences of 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 to address selection bindings in a limited way.

The implementation warns if channel arrays are passed by reference as template parameters. Exact detection of aliasing is not possible because reference equality is not testable.

3.2.1 State expressions only

Rather than collecting edges on a single action, they must now be grouped by channel set and direction.

The set of m edges to be flipped is now written

$$\begin{aligned} E \;=\; & \big\{\big(S_1, A_1, g_1, \langle e_1^1, \ldots, e_{n_C}^1\rangle\big), \\ & \quad \ldots, \big(S_m, A_m, g_m, \langle e_1^m, \ldots, e_{n_C}^m\rangle\big)\big\}, \end{aligned}$$

where the added expression sequences specify an element of the channel set. We make the same disjointness assumptions as in the previous section, and additionally require that quantifier bindings are restricted to guards, and likewise, until the next section, for selection bindings, that is,

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

The edges within E must be grouped by channel and allowance made for channels not represented by any edge. For example, given a set of two channels $C = \{c[1], c[2]\}$, and,

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

there are two possibilities, for a fixed valuation val_V: if $[\![e^1]\!]_{\mathsf{val}_V} = [\![e^2]\!]_{\mathsf{val}_V}$ 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 $c[\,[\![e^2]\!]_{\mathsf{val}_V}]$.

We originally formed explicit edge groupings in this way, but the number of possible partitions grows very quickly, and the generalization to index expressions containing selection bindings is intricate. It turns out that both problems can be avoided by introducing additional selection bindings.

We introduce a new selection binding, ensuring disjoint names, for each channel array dimension, with a type that spans the entire dimension. Let $S_{\mathrm{w}} = \{z_1, \ldots, z_{n_C}\}$ be this set of n_C *sweep bindings*. We assume that they are disjoint from one another and from other selection and quantifier bindings. Each valuation of the bindings specifies a different element of the channel set and all elements are considered.

[4] Unless they are template parameters passed by reference.

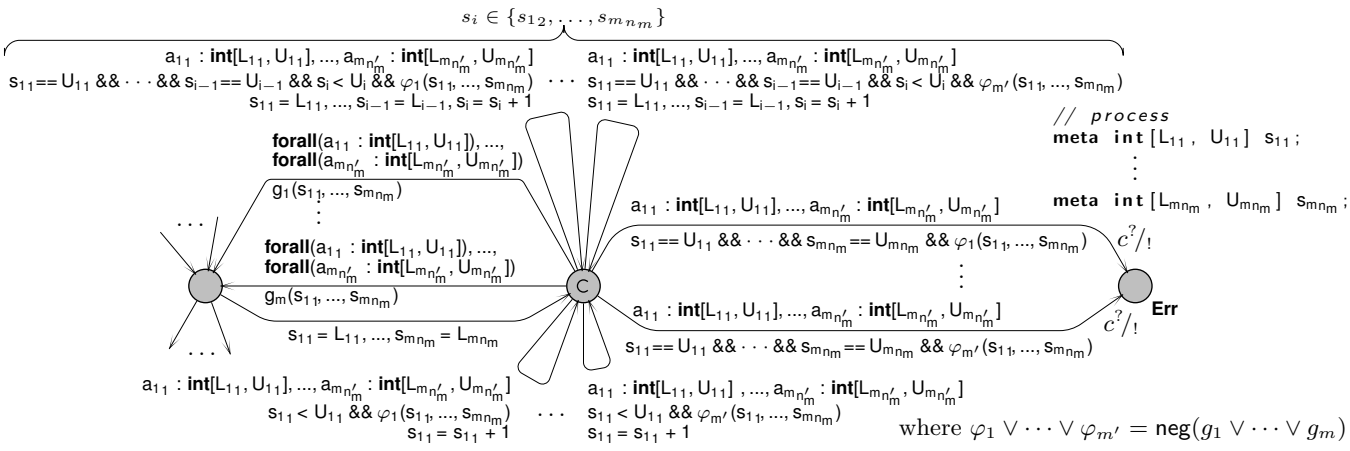

Figure 5: Construction for $\forall s_{11}, \ldots, s_{m n_m}. \exists a_{11}, \ldots, a_{m n'_m}. \mathsf{neg}(g_1 \vee \cdots \vee g_m)$

For any valuation, the negated disjunction of all applicable guards must be formed, as for an elementary channel. Thus to each sequence of index expressions we associate a clause satisfied only when the sweep binding valuation specifies the same element as do the evaluated expressions. The conjunctions of each clause and guard may then be combined in disjunction, and the entirety negated as required. The clauses ensure appropriate grouping of guards as the model checker evaluates the entire expression through all valuations.

The m edges can thus be considered as a single edge

$$\begin{aligned}\epsilon = \Big(&S_\mathrm{w} \cup S_1 \cup \cdots \cup S_m, A_1 \cup \cdots \cup A_m,\\ &(z_1 = e_1^1 \ \wedge \cdots \wedge \ z_{n_C} = e_{n_C}^1 \ \wedge \ g_1)\\ \vee\ &(z_1 = e_1^2 \ \wedge \cdots \wedge \ z_{n_C} = e_{n_C}^2 \ \wedge \ g_2)\\ &\vdots\\ \vee\ &(z_1 = e_1^m \ \wedge \cdots \wedge \ z_{n_C} = e_{n_C}^m \ \wedge \ g_m),\\ &\qquad\qquad \langle z_1, \ldots, z_{n_C}\rangle\Big).\end{aligned}$$

For a valuation of the sweep bindings $\mathsf{val}_{S_\mathrm{w}}$ the conjunct $(z_1 = e_1 \wedge \cdots \wedge z_{n_C} = e_{n_C} \wedge g)$ is either equivalent to g, and remains in the disjunction, or to false, and drops from the disjunction, depending on whether the guard applies to an edge on a channel set element identified by $\langle z_1, \ldots, z_{n_C}\rangle$. The guards of channels not referred to by any index sequence are, appropriately, equivalent to false. The sets of universal bindings A_i may be grouped into the union because the scope of each applies only to a single guard expression g_i and is disjoint from the set of free variables in the array index expressions $e_1^i, \ldots, e_{n_C}^i$. Similarly for the selection bindings S_i. We also note that the $(z_1 = e_1 \wedge \cdots \wedge z_{n_C} = e_{n_C})$ subexpressions do not contain clocks since they are formed by equating selection bindings and array index expressions, and that every assignment to S_w selects a different channel.

Edge ϵ represents multiple transitions, one for each valuation of S_w. No transitions are enabled for an action when

$$\begin{aligned}&\forall s_{11}, \ldots, s_{m n_m}. \exists a_{11}, \ldots, a_{m n'_m}.\\ &\qquad (z_1 \neq e_1^1 \ \vee \cdots \vee \ z_{n_C} \neq e_{n_C}^1 \ \vee \ \mathsf{neg}(g_1)) \qquad (\psi_3)\\ &\wedge\ (z_1 \neq e_1^2 \ \vee \cdots \vee \ z_{n_C} \neq e_{n_C}^2 \ \vee \ \mathsf{neg}(g_2))\\ &\quad\vdots\\ &\wedge\ (z_1 \neq e_1^m \ \vee \cdots \vee \ z_{n_C} \neq e_{n_C}^m \ \vee \ \mathsf{neg}(g_m)).\end{aligned}$$

The sweep bindings are not negated into universal quantifiers since their role is only to choose elements from the channel set. They do not have the effect of disjunction since no two valuations specify the same channel. For an element, the quantifier free part is equivalent to a conjunction of l negated guards $g_{i_1} \wedge \cdots \wedge g_{i_l}$, which is equivalent to the corresponding form for elementary channels, §3.1.3.

Formula ψ_3 must be manipulated to form a set of valid Uppaal transitions. This means swapping the universal and existential quantifiers and transforming the quantifier-free subexpression into DNF before splitting the resulting clauses over multiple transitions. The techniques of previous sections remain applicable. The sweep variables are treated as state variables during the manipulations and then afterward reintroduced on each separate transition.

In the construction for formulas with unswappable quantifiers, Figure 5, sweep binding values must remain constant while testing the range of universal quantifier valuations. Thus a meta variable would also be introduced for each sweep binding; with a value assigned non-deterministically on entry to the committed state. The meta variables replace the bindings in guard expressions and when selecting a specific channel on the transitions into Err.

In Figure 6, $\mathcal{P}_6$ has two transitions on the same two dimensional channel array; both indexed by state variables, one with tail and i, the other with head and j. Two sweep bindings, s_0 and s_1, are introduced in $\mathcal{P}_6^{\mathsf{Err}}$. The negated guard for c? gives twelve transitions after conversion to DNF. The increase in transitions is unfortunate but less pronounced when selection bindings index channel arrays—which we consider next and expect to be more usual. The constants N_1 and N_2 are declared globally, but they could be template parameters; $\mathcal{P}_6^{\mathsf{Err}}$ is correct whatever their exact values.

3.2.2 *Limited selection bindings*

We now relax the restriction on selection bindings occurring in channel array subscripts. The set of m edges to be flipped is written as in the previous section, and disjointness per §3.1.3 is also assumed. Quantifiers are still restricted to guards, $\forall 1 \leq i, j \leq m, 1 \leq k \leq n_C.\ A_i \cap \mathsf{freevars}(e_k^j) = \emptyset$, but selection bindings may now occur as index expressions, $\forall 1 \leq i \leq m, 1 \leq k \leq n_C, s \in S_i.\ s \in \mathsf{freevars}(e_k^i) \Rightarrow e_k^i = s$. We assume, for now, that no selection variable appears in two different index positions on an edge. We will also assume that the type of each selection binding used in an array di-

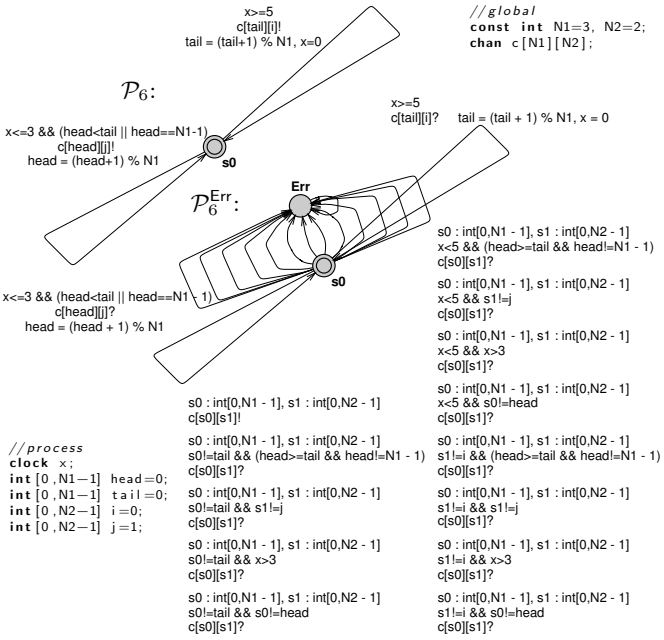

Figure 6: Channel arrays with state variables

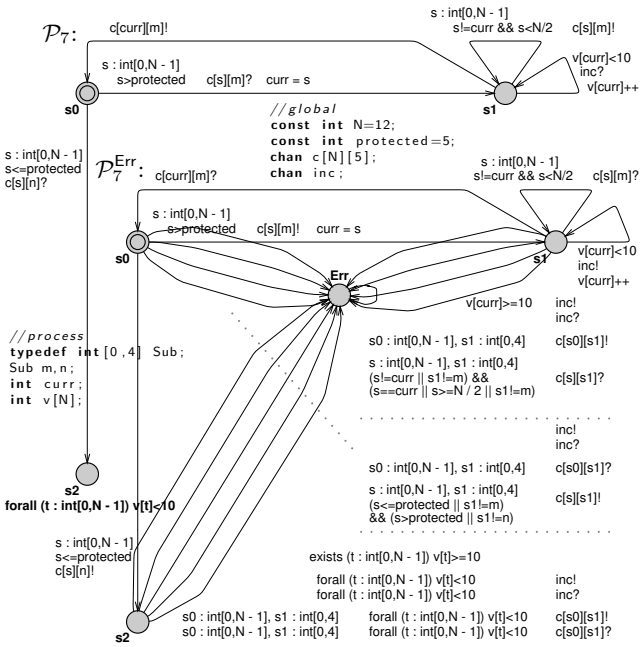

Figure 7: Channel arrays with selection bindings

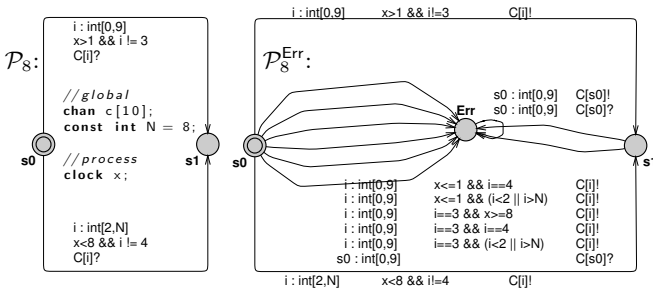

Figure 8: Channel selections with differing ranges

mension spans that entire dimension. Scalar bindings must, and while an integer binding s of type $[l, u]$ may index a dimension of larger size, its type can be altered if corresponding guards are further constrained by the clause $l \leq s \leq u$.

When merging transitions, a selection binding s that indexes array dimension i will be replaced by the sweep binding z_i, as for a state expression. But rather than include the restriction $z_i = s$ in the new guard conjunct observe that of all the possible valuations that include z_i and s, only those where the two are equal need actually be considered. Since both variables have the same type, we can replace s with z_i in both the set of selection bindings, effectively removing s, and the guard, where the restriction $z_i = z_i$ would have anyway become redundant.

Edges where a selection binding occurs in multiple index positions are readily addressed, despite the previous assumption, by further constraining their guards. For example given $S_w = \{z_1, z_2\}$, the edge $(\{s\}, A, g, \langle s, s\rangle)$ may become $(S_w, A, z_2 = z_1 \wedge g, \langle z_1, z_2\rangle)$. This manipulation can be combined with the one for stretching types.

Figure 7 is an example of channel selection involving selection bindings. The two edges leaving $\mathsf{s_0}$ are combined and negated to give an edge to Err. The selection bindings in the first dimension are conflated to a sweep binding. The state variables, m and n, are replaced by a sweep binding and new guard constraints. The two edges leaving $\mathsf{s_1}$ on c are handled similarly, even though one specifies the first dimension with a state variable curr and the other with a selection binding s. The resulting guards are not converted to DNF because they do not contain clock variables.

Another example is given in Figure 8. The selection binding on the lower transition is of type **int** [2, N], whereas the array dimension is of type **int** [0,9] . The merged guard thus becomes $(x > 1 \wedge i \neq 3) \vee (x < 8 \wedge i \neq 4 \wedge i \geq 2 \wedge i \leq N)$, which is negated to five transitions in $\mathcal{P}_8^{\mathsf{Err}}$. The current simplifier cannot eliminate the transition labeled $i_1 = 3 \wedge i_1 = 4$, which can obviously never be enabled.

The technique can be extended to allow selection bindings in some forms of composite index expressions, namely those preserving the property that each selection binding valuation specifies a different channel. Some operators preserve this property, like (non-zero) addition and integer multiplication, but others do not necessarily, like modulo, shift, division, and function calls (like $f(x) = 1$). Multiple selection binding valuations that specify the same channel cannot be considered separately but rather their guards and quantifiers must be grouped prior to negation. Implementing such features does not seem to be worth the additional complexity.

3.3 Inverting invariants

The validation construction produces τ-transitions for locations with non-trivial invariants (rule 1 of Definition 3.1). It is sometimes necessary to split negated invariants over transitions per §3.1.1, and to treat quantifiers per §3.1.3.

3.4 Urgent Channels and shared variables

A process is not usually obliged to synchronize on enabled channels if its location invariant permits further delay, but synchronization on urgent channels has priority over delay.

Basic timed trace inclusion is insufficient for models with urgent channels, instead *timed ready simulation* and a val-

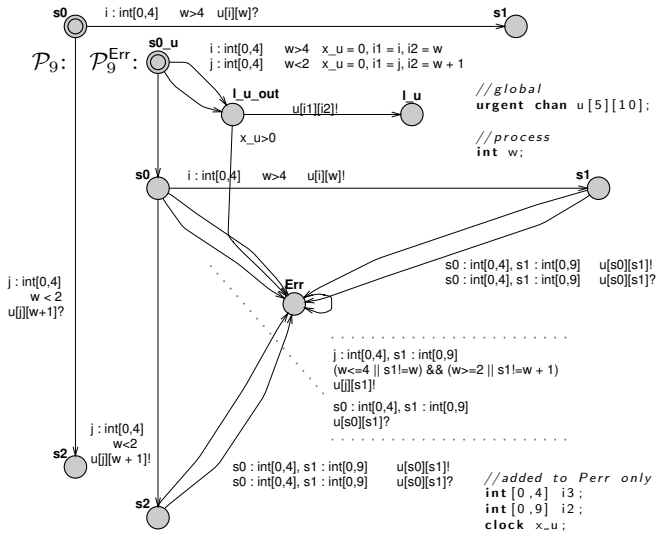

Figure 9: Validating an array of urgent channels

idation construction that can also handle shared variables have been proposed [3]. There is only space to note that this technique can be extended and implemented for the processes we consider. Arrays of urgent channels are handled by storing index values in a variable across the two-transition check for immediate synchronization, as in the example of Figure 9. Uppaal supports direct array comparison, so shared variable arrays present no special challenges.

Uppaal does not allow clock variables in the guards of transitions that synchronize on urgent channels. They may be introduced when transforming models where clock variables are used in location invariants.[5]

4. IMPLEMENTATION

The techniques described have been implemented in a tool called *urpal*, written in Standard ML, that parses Uppaal models and generates validation automata if possible.

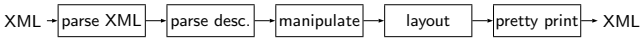

Figure 10: High-level structure of Urpal

The five major subsystems of the tool are shown in Figure 10: a parser for the Uppaal XML format; a parser for the description language used in declarations, expressions and actions; algorithms for transforming models; an interface to Graphviz for placing nodes and routing transitions; and a pretty printer back to an Uppaal model file.

Uppaal is distributed with a library for parsing the file format and description language, and type checking models. We decided that integrating the object-oriented library into Standard ML would involve as much work as writing a custom parser, besides making the tool dependent on third-party binary releases and complicating installation.

An interface to Graphviz tries to untangle introduced transitions while preserving the original layout. Producing readable models is essential to the understandability of counter-example traces if timed trace inclusion validation fails.

Although the tool focuses on the validation construction, its subsystems are suitable for implementing other model transformations. We have, for instance, added an expression language for pruning transitions and manipulating nodes.

The tool does not validate the assumption of determinism, which ultimately depends on reachable valuations of state variables. Fortunately, since timed trace inclusion is reflexive, the assumption can be checked by attempting to verify $\mathcal{P} \parallel \mathcal{P}^{\mathsf{Err}} \models \mathsf{A}\Box\neg\mathsf{Err}$. A failure indicates either a model for which determinism or τ-freedom do not hold, or a flaw in the implementation or in Uppaal. Unfortunately the converse is not true in general.

Efficiency has not been a major concern as the tool need only transform models for which model checking is feasible.

[5]Unless the negated disjunction of guards simplifies to false.

5. CONCLUDING REMARKS

Uppaal contains features that increase both modeling convenience and the complexity of manipulation. We have formalized many of these features and shown how to transform them for timed trace inclusion validation. We have not evaluated the output models for efficiency of analysis.

Our implementation could be improved with support for committed locations and priorities, more sophisticated term rewriting, features for relabeling to eliminate nondeterminism, and support for multi-process specifications.

6. ACKNOWLEDGMENTS

This work was inspired by a pleasant afternoon at Radboud University with Frits Vaandrager. The first author thanks Ansgar Fehnker and Rob van Glabbeek of NICTA for their support and insights. We also thank Kim Larsen for suggesting the railway example and the anonymous referees for their comments.

7. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Comp. Sci.*, 126(2):183–235, Apr. 1994.

[2] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information & Computation*, 111(2):192–244, June 1994.

[3] H. E. Jensen, K. G. Larsen, and A. Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In M. Joseph, editor, *Proc. 6th Int. Symp. Formal Techniques for Real-Time and Fault-Tolerance*, volume 1926 of *LNCS*, pages 19–30, Pune, India, Sept. 2000. Springer-Verlag.

[4] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. *Int. J. Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[5] N. Lynch and F. Vaandrager. Forward and backward simulations. Part I: Untimed systems. *Information & Computation*, 121(2):214–233, Sept. 1995.

[6] N. Lynch and F. Vaandrager. Forward and backward simulations. Part II: Timing-based systems. *Information & Computation*, 128(1):1–25, July 1996.

[7] M. I. Stoelinga. *Alea Jacta est: Verification of probabilistic, real-time and parametric systems*. PhD thesis, Katholieke Universiteit Nijmegen, Apr. 2002.

[8] Y. Wang, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII*, pages 223–238, Berne, Switzerland, 1994. IFIP, Chapman & Hall.