Enhancing UPPAAL by Exploiting Symmetry

M. Hendriks

Nijmegen Institute for Computing and Information Sciences/

NIII-R0208 October

Nijmegen Institute for Computing and Information Sciences
Faculty of Science
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Enhancing UPPAAL by Exploiting Symmetry[⋆]

MARTIJN HENDRIKS martijnh@cs.kun.nl
Department of Computing Science, University of Nijmegen, The Netherlands

Abstract

Efficiency is one of the major concerns in the world of model checking. Consequently, many techniques to optimize the time and space usage of model checking algorithms have been invented. One of these techniques is reduction of the searchable state space through arguments of *symmetry*. This technique can be very profitable and has been implemented in various model checkers, but not yet in UPPAAL, a model checker for timed systems.

This paper proposes an enhancement of UPPAAL with symmetry reduction. We adopt the theory of symmetry of Ip and Dill and their scalarset data type, as implemented in the model checker MURφ. The main result of this paper is a soundness proof of our symmetry enhancement, which does not follow trivially from the work of Ip and Dill since the description languages of UPPAAL and MURφ, which are used to detect the symmetries, are quite different. A secondary result is a proof of the computational difficulty of space-optimal exploitation of full symmetry in a setting with DBM technology.

Keywords: Real-time, timed automata, symmetry, model checking.
AMS subject classification: 68Q60.
CR subject classification: D.2.4.

1 Introduction

Model checkers emerge as practical tools for the mechanical verification of all kinds of systems [9]. In this approach, a model of the system to be verified and the verification properties serve as input to the tool, which consequently computes whether or not the model satisfies the specification. Nowadays, many model checkers are available, ranging from model checkers for JAVA source code [10] to model checkers for timed automata [17, 21].

Despite the relative ease of use of model checkers, they are not applied on a large scale. An important reason for this is that they must cope with the *state space explosion* problem. This is the problem of the exponential growth of the state space as models

[⋆]Supported by the European Community Project IST-2001-35304 (AMETIST) and the PROGRESS project TES4999: Verification of Hard and Softly Timed Systems (HAAST).

become larger. This growth often renders the mechanical verification of realistic systems practically impossible: there just is not enough time or memory available.

A passive approach to the huge resource requirements of model checkers consists of waiting for faster computers and cheaper memory. A more preferable way is finding more efficient *techniques* to fight the state space explosion. The exploitation of behavioral symmetries is such a well-known technique that has been successfully implemented for various model checkers, e.g., MURφ [12, 14], SMV [18] and SPIN [13, 7]. Especially the exploitation of *full* symmetries in a model can be profitable, since its gain can approach a factorial magnitude.

There are two main problems that should be solved before symmetry reduction can be implemented. First, one should statically detect symmetries in the system description. Second, during the state space exploration, one must decide whether or not some discovered state has already been encountered in the past. This decision should of course take symmetry into account. It has been shown that this so-called *orbit problem* is – in general – at least as difficult as testing for graph-isomorphism [8] and, unfortunately, there are no known polynomial algorithms for this last problem. This does not necessarily mean that symmetry reduction is a lost case, since we can always try to revert to sub optimal, but still feasible, solutions. Moreover, it might happen that the considered instance of the orbit problem is not so difficult.

This paper proposes a symmetry enhancement of UPPAAL, a model checker for networks of timed automata, by applying the theory of symmetry of Ip and Dill and adding their scalarset data type [14]. The main result of this paper is a soundness proof of our symmetry enhancement. More precisely, we prove that certain permutations on the states are sound with respect to reachability properties: if we have seen a state s, then we can conclude that we have seen all states which are obtainable by applying these permutations to s. Thus, we can use the permutations to reduce the amount of states which need to be explored by the model checking algorithm.

Our main result does not follow trivially from the work of Ip and Dill since the description languages of UPPAAL and MURφ, which are used to detect the symmetries, are quite different. This difference disallows us to apply Ip and Dill's soundness proof in a straightforward manner without loosing confidence in its validity. Since formal methods require mathematical precision, we are forced to construct a new proof for the soundness of symmetry reduction in UPPAAL.

A secondary result is a proof of the computational difficulty of exhaustive exploitation of full symmetry when difference bounded matrices (DBMs) are used for the symbolic representation of clock values. In other words, the orbit problem is very difficult when the state is (partially) represented by a DBM. In order to let our symmetry reduction technique be profitable with respect to both time and space consumption, we should revert to "sub optimal" solutions. Empirical results of Ip and Dill show that this certainly is not as bad as it sounds [14].

This work is directly motivated by attempts to verify various distributed systems, which clearly exhibit full symmetry, using UPPAAL. For example, Fischer's mutual exclusion protocol (see, for instance, [1]) for 12 or more processes is practically unverifyable. Similarly, a simple model of a CSMA/CD protocol (see, for instance, [21]) is practically unverifyable when 13 or more processes are considered. The state space explosion problem is felt more directly during the attempts of verifying a distributed agreement algorithm [5]. Even the smallest interesting instance of the algorithm (three processes) could not be verified. Hopefully, implementation of symmetry enhanced UPPAAL can help us to overcome these boundaries.

Outline. In section 2 we summarize the theory of Ip and Dill for symmetry reduction. In section 3 we extend the description language of UPPAAL with the scalarset data type, and with multidimensional arrays of integer variables and channels. We illustrate the extended syntax by modeling Fischer's mutual exclusion protocol. Moreover, we give a formal definition of these SUPPAAL models, and we explain their semantics. In section 4 we extract the automorphisms from the system description, and we give the soundness proof. In section 5 we prove that the most optimal use (with respect to the size of the explorable state space) of these automorphisms is not feasible with respect to computation time. Finally, in section 6 we summarize this paper and we discuss future work.

Acknowledgement. We thank Frits Vaandrager for commenting on earlier versions of this paper.

2 A theory of symmetry

In this section we summarize the theory of symmetry developed by Ip and Dill [14]. They consider *state graphs*, which are tuples containing a set Q of states, a set $Q_0 \subseteq Q$ of initial states, a transition relation $\Delta \subseteq Q \times Q$ and a unique error state, which we omit in our presentation[1]. A state $q \in Q$ is *reachable*, iff a sequence $q_0, q_1, ..., q_{n-1}$ exists, such that $q_0 \in Q_0$, $q = q_{n-1}$, $q_i \in Q$ for all $0 \leq i < n$, and $(q_i, q_{i+1}) \in \Delta$ for all $0 \leq i < n-1$.

We assume the existence of a set of *state properties* Φ, for whose elements we can decide whether they are true or false in some state. If a state property $\phi \in \Phi$ is true in state q, then we denote this by $q \models \phi$. Figure 1 depicts a standard forward exploration algorithm, which (semi) decides whether or not a state is reachable which satisfies some given state property ϕ.

```
passed := $\emptyset$
waiting := $Q_0$
while waiting $\neq \emptyset$ do
    get $q$ from waiting
    if $q \models \phi$ then return YES
    else if $q \notin$ passed then
        add $q$ to passed
        waiting := waiting $\cup$ succ($q$)
    fi
od
return NO
```

Figure 1: Standard forward reachability analysis.

The algorithm starts by adding the initial states to the *waiting* set. Then it enters a loop that processes all the states in the *waiting* set in the following way. If some waiting state q satisfies the state property ϕ, then the algorithm returns YES. Otherwise, it

[1]Omitting the error state does not change the validity of Ip and Dill's results [14].

checks whether or not q has already been seen. If this is the case, $q \in$ *passed*, then the algorithm discards q and gets a new state from the *waiting* set. If q has not yet been encountered, then it is added to the *passed* set and all its successors are added to the *waiting* set. If the state space – the set Q – is finite, then this algorithm halts. Otherwise, it may not halt.

Ip and Dill define symmetry within a state graph as a graph automorphism different from the identity relation.

Definition 2.1 (Automorphism) *A graph automorphism on a state graph* (Q, Q_0, Δ) *is a bijection* $h : Q \to Q$ *such that*

(i) $q \in Q_0$ *iff* $h(q) \in Q_0$ *for all* $q \in Q$, *and*

(ii) $(q_1, q_2) \in \Delta$ *iff* $(h(q_1), h(q_2)) \in \Delta$ *for all* $q_1, q_2 \in Q$.

For any set of graph automorphisms H, the closure of $H \cup \{\mathbf{id}\}$, where $\mathbf{id}$ is the identity function, under inverse and composition, denoted by $\mathcal{C}(H)$, is a group. Such a *symmetry group* $\mathcal{C}(H)$ induces a relation $\approx_H \subseteq Q \times Q$ such that $q_1 \approx_H q_2$ iff there exists an $h \in \mathcal{C}(H)$ such that $h(q_1) = q_2$. This relation is an equivalence relation and we let $[q]$ denote the equivalence class of state q. Using these equivalence classes we can define a quotient graph.

Definition 2.2 (Quotient graph) *Let* $A = (Q, Q_0, \Delta)$ *be a state graph and let* $\mathcal{C}(H)$ *be a symmetry group for* A. *The quotient graph induced by* $\mathcal{C}(H)$ *is the graph* $A_{\approx_H} = (Q', Q'_0, \Delta')$, *where* $Q' = \{[q] \mid q \in Q\}$, $Q'_0 = \{[q] \mid q \in Q_0\}$ *and* $\Delta' = \{([p], [q]) \mid (p, q) \in \Delta\}$.

Ip and Dill observed that a quotient graph can be used to check reachability properties: q is reachable in A iff $[q]$ is reachable in $A_{\approx H}$. Since the quotient graph is at most as large as the original state graph, and in many cases smaller, the use of the quotient graph can speed up the model checking process.

As already mentioned in the introduction, the two major problems that should be solved in the actual implementation of symmetry reduction are the following:

- We must detect a set of automorphisms from the system description. The corresponding symmetry group induces the (smaller) quotient graph.
- During the exploration of the state space, we must be able to decide whether or not two states are symmetric. Thus, for states q and q', we must decide whether or not $[q] = [q']$.

In addition to these two problems, there is another, yet smaller, problem. We assumed a set of state properties Φ which we used in algorithm 1 for the forward exploration of the state space. If we apply symmetry reduction to the state graph, then we should also apply it to the state properties. In other words, validity of state properties should be symmetric:

$$\forall_{\phi \in \Phi} \, \forall_{q, q' \in Q} \, \left(q \approx_H q' \Rightarrow (q \models \phi \Leftrightarrow q' \models \phi)\right) \qquad (1)$$

In order to protect the gain of using the quotient graph, the approaches to both problems should be computationally cheap. In the next sections we add the well-known scalarset data type to UPPAAL in order to statically detect symmetries from the system description. As for the second problem, our strategy is to convert all explored states to a so-called *normal form* using the detected automorphisms. This normal form represents the equivalence class of the state. The only correctness requirement for our normal form operator θ is the following:

$$\forall_{q,q' \in Q} \ \big(\theta(q) = \theta(q') \ \Rightarrow \ [q] = [q']\big) \tag{2}$$

This criterion says that if two states have the same normal form, then they are contained in the same equivalence class. Note that if $\theta \in \mathcal{C}(H)$, then property (2) is certainly satisfied. If the implication of property (2) also holds the other way around, then the normal form operator is *canonical.*

The function $\Theta : 2^S \to 2^S$ converts a set of states to their normal forms in the regular way: $\Theta(S) = \{\, \theta(s) \mid s \in S \,\}$. We now can state a new forward exploration algorithm, depicted in figure 2, which uses the normal form operator to take symmetry into account.

```
$passed := \emptyset$
$waiting := \Theta(Q_0)$
while $waiting \neq \emptyset$ do
    get $q$ from $waiting$
    if $q \models \phi$ then return YES
    else if $q \notin passed$ then
        add $q$ to $passed$
        $waiting := waiting \cup \Theta(succ(q))$
    fi
od
return NO
```

Figure 2: Adding symmetry to the forward reachability analysis.

This new algorithm uses θ and Θ to convert all discovered states to their normal forms. If θ is canonical, then exactly the quotient graph will be explored. However, if θ is not canonical, then *at most* the original state graph will be explored.

3 From UPPAAL to SUPPAAL

The tool UPPAAL has been based on the theory of timed automata of Alur and Dill [4, 2]. In short, a UPPAAL model consists of a network of timed automata enhanced with (arrays of) bounded integer variables, which communicate through shared variables and by binary blocking synchronizations (see, for instance, the *help* menu in the tool itself and [17]). In this section we explain how we add the scalarset data type to the system description language of UPPAAL. We assume some knowledge about this description language, and in particular about the "templates" and their instantiation mechanism.

In section 3.1 we formally define the symmetry extension of the syntax of the system description language. In section 3.2 we give a formal definition of UPPAAL models enhanced with symmetry, which we call SUPPAAL models from now on. Moreover, we present a version of a model of Fischer's mutual exclusion protocol that has been adjusted for symmetry. Finally, in section 3.3 we explain the semantics of these models using the mathematical representation.

3.1 Adding the scalarset data type to Uppaal

In this section we explain how we extend the syntax of the system description language of Uppaal with scalarsets. We do not give the complete original syntax of this language here since it can be easily found in the help menu of the tool itself. At the end of this section we use the extended syntax to model Fischer's mutual exclusion protocol (see, e.g., [1]).

The additions we propose are split into three parts. First, we explain the additions to the declarations section, second, we explain the changes to template parameters, and third, we explain the additions to the process assignments section.

Additions to the declarations section

First, we add the scalarset data type to Uppaal. This is a sub range of integers with fixed size, and whose elements can be arbitrarily permuted without changing the behavior of the system. First, we want to be able to declare scalarset types with different sizes in the global declaration sections of Uppaal models as follows:

```
scalarset pid[3];
scalarset bid[5];
```

A scalarset α is the sub range $\{0, ..., |\alpha| - 1\}$, where $|\alpha|$ denotes the size of the scalarset. In order to use these scalarsets, we should be able to use them in declarations. For example:

```
int[0,1] input[bid];
```

The *scalarset array* `input` contains boolean values. Its size is fixed by the size of the scalarset, which in this case equals 5. In order to make optimal use of scalarsets we also add *multi dimensional* arrays of bounded integer variables and channels. For example:

```
int dim3[pid][bid][7];
chan cd[5][pid];
```

The three-dimensional integer array `dim3` thus contains $3 \times 5 \times 7$ elements, and `cd` is a two dimensional array of channels with 15 elements.

Next, we formally state the changes to the original syntax of Uppaal to facilitate the mechanisms sketched above (see the help menu of Uppaal for the complete original syntax definition). First, we add the scalarset type to the syntax:

```
Declarations ::= ( NewDecl ';' )*

NewDecl ::= Decl  |  'scalarset' ID '[' CExpr ']'
```

The second formal adjustment to the original syntax is the redefinition of the grammar for `IL`:

```
IL ::= ILID ( ‘,’ ILID )*

ILID ::= ID ( ‘[’ CExpr ‘]’ )*
```

This renewed definition of `IL` allows us to declare (multi-dimensional) arrays of channels. Note that with this syntax we can also declare (multi-dimensional) arrays of clocks. However, for reasons of simplicity we do not allow this. Finally, we redefine the variable identifiers:

```
VID ::= ID ( ‘[’ (CExpr | ID) ‘]’ )*   |   ID ‘:=’ CExpr
```

The renewed definition of `VID` allows us to declare multi-dimensional arrays of integers. The second appearance of the non-terminal `ID` in the first rule of the definition of `VID` must be bound to a scalarset. Thus, we can use a scalarset type to index arrays.

Additions to the template parameters

Apart from using scalarset types to declare variables and to index arrays, we also want to use them for the instantiation of templates. Consider for example the well-known Fischer mutual exclusion protocol. It contains n processes, which only differ in a unique process identifier, which can be modeled by the scalarset `pid`. The UPPAAL model only contains a single template, say `P`, of a generic Fischer process which takes a process identifier as argument. All processes are created by instantiation of this template. We propose the following syntax for the template parameters of `P`:

```
process P ( scalarset pid; )
```

For the sake of simplicity we only allow templates to be instantiated with scalarsets. Thus, we redefine the syntax for template parameters as follows:

```
Param ::= ( ‘scalarset’ ID (’,’ ID)* )*
```

Note that this definition allows an empty parameter list.

Additions to the process assignments section

We explained above that we can model the unique process identifiers of the Fischer protocol as a scalarset, say `pid`. Additionally, we can instantiate the above mentioned template `P` with the elements of this scalarset. We propose the following syntax for this instantiation in the process assignments section:

```
FischerProcs := P(pid);
```

Informally, this construct means that the “object” `FischerProcs` is the parallel composition of |`pid`| instantiations of template `P` with *all* elements of the scalarset `pid`. Moreover, we also propose *layered* instantiation using the scalarsets. Assume that `pid` and `bid` are scalarsets, then

```
Procs1 := P(pid);
Procs2 := B(pid,bid);
```

creates the process `Procs1` as the parallel composition of |`pid`| instances of template `P` and `Procs2` as the parallel composition of |`pid`| × |`bid`| instances of template `B`. In general, we redefine the syntax for the process assignments, given by `PAList`, as follows:

```
PAList ::= (ID ':=' ID '(' [Values] ')' ';')*

Values ::= ID (',' ID)*
```

Of course, there are some syntactical and semantic restrictions for using the new process assignment construction. We do not elaborate on them here, since they are not very interesting and mostly straightforward.

Example: Fischer's mutual exclusion protocol

We can use the adjusted syntax to model Fischer's mutual exclusion protocol (based on the model that is distributed with Uppaal). We start with the global declarations of the Suppaal model:

```
scalarset process_id[3];
int id:=-1;
```

Next, there is the template `P`, which has a header `process P (scalarset pid)` and uses a local clock `x`. This template has been depicted in figure 3.

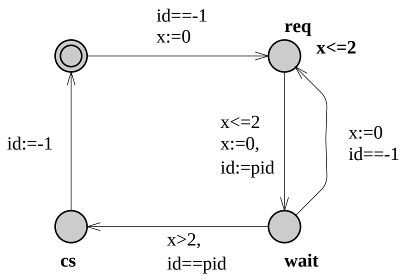

Figure 3: The Fischer process template.

The global idea of the protocol is that each "Fischer process" has a unique process identifier (elements of the scalarset `pid` in our model). As soon as a Fischer process wants to enter its critical section, it writes its own identifier in a global variable (`id` in our model). If the global variable still contains its identifier after a certain amount of time (2 time units in our model), then the Fischer process may enter its critical section. When it leaves its critical section, then it resets the global variable to a neutral value (-1 in our model). The process assignments section consists of only the following line:

```
Procs := P(process_id);
```

As we explained in the previous section, the object `Procs` denotes the parallel composition of 3 (since the scalarset `process_id` has size 3) Fischer processes. Note that the process identifiers range from 0 to 2, which ensures that -1 indeed is a neutral value for the global variable `id`. Finally, the system definition section only contains the line "`system Procs;`", which speaks for itself.

3.2 Mathematical description of SUPPAAL models

In the previous section we added multidimensional arrays and the scalarset data type to UPPAAL's system description language to obtain the system description language of SUPPAAL. In this section we give a mathematical representation of these SUPPAAL models, which can easily be derived from the new system description language.

To formally define a SUPPAAL model, we first need to define *guards*, *invariants* and *assignments* over the clocks and *guards* and *assignments* over the bounded integer variables.

Definition 3.1 (Clock guard) *A clock guard ϕ over a set of clocks X is defined by the grammar $\phi ::= x \sim n \,|\, x \sim y + n \,|\, \phi \wedge \phi$, where $x, y \in X$, $n \in \mathbb{Z}$ and $\sim \in \{\leq, <, =, >, \geq\}$. We let $CG(X)$ denote the set of all clock guards over X.*

Next, we define the set of invariants, which is a subset of the set of clock guards.

Definition 3.2 (Invariant) *An invariant ϕ over a set of clocks X is defined by the grammar $\phi ::= x \sim n \,|\, \phi \wedge \phi$, where $x \in X$, $n \in \mathbb{N}$ and $\sim \in \{\leq, <\}$. We let $Inv(X)$ denote the set of all invariants over X.*

Clock assignments reset clocks to an integer value greater than or equal to zero.

Definition 3.3 (Clock assignment) *A clock assignment ca over a set of clocks X is defined by the grammar $ca ::= x := n$, where $x \in X$ and $n \in \mathbb{N}$. We let $CA(X)$ denote the set of all clock assignments over X.*

Next, we define the integer expressions, which can be used as part of integer assignments, integer guards and synchronizations. For sake of simplicity, we do not allow arrays to index arrays.

Definition 3.4 (Integer expression) *An integer expression $IExpr$ over a set of variables V and a set of scalarsets Ω is defined by the grammar*

$$\begin{array}{lcl} SIExpr & ::= & z \;\;|\;\; \alpha \;\;|\;\; v \;\;|\;\; (SIExpr \oslash SIExpr) \\ \\ IExpr & ::= & SIexpr \;\;|\;\; v[SIExpr]^{+} \end{array}$$

where $z \in \mathbb{Z}$, $\alpha \in \Omega$, $v \in V$, and $\oslash \in \{/, *, +, -\}$. We let $IX(V, \Omega)$ denote the set of all integer expressions over V and Ω. Elements of $SIExpr$ are called simple integer expressions.

An integer guard consists of a conjunction of comparisons between two integer expressions.

Definition 3.5 (Integer guard) *An integer guard ϕ over a set of variables V and a set of scalarsets Ω is defined by the grammar $\phi ::= IExpr \sim IExpr \,|\, \phi \wedge \phi$, where $\sim \in \{\leq, <, =, \neq, >, \geq\}$. We let $IG(V, \Omega)$ denote the set of integer guards over V and Ω.*

An integer assignment assigns the value of some integer expression to an integer variable, or to an entry of a (multi dimensional) integer array.

Definition 3.6 (Integer assignment) *An integer assignment ϕ over a set of variables V and a set of scalarsets Ω is defined by the grammar $\phi ::= v[\mathit{SIExpr}]^* := \mathit{IExpr}$ where $v \in V$. We let $IA(V, \Omega)$ denote the set of all integer assignments over V and Ω.*

Finally, we define the set of synchronizations over a set of synchronization labels Σ, a set of variables V, and a set of scalarsets Ω.

Definition 3.7 (Synchronization) *A synchronization ϕ over a set of labels Σ, a set of variables V and a set of scalarsets Ω is defined by the grammar*

$$\phi ::= \quad \tau \quad | \quad \sigma[\mathit{SIExpr}]^*! \quad | \quad \sigma[\mathit{SIExpr}]^*?$$

where τ denotes the "empty" synchronization and $\sigma \in \Sigma$. We let $\mathit{Sync}(\Sigma, V, \Omega)$ denote the set of all synchronizations over Σ, V, and Ω.

For instance, if $s \in \Sigma$ and $\alpha \in \Omega$, then $s[\alpha]! \in \mathit{Sync}(\Sigma, V, \Omega)$. Such a synchronization does not mean that the value α is send over the channel s. Instead, it expresses the blocking synchronization over the α-*th* element of the array of channels s. This construct is motivated by models in which the symmetric components are modeled by the parallel composition of synchronizing processes, e.g., a computation process and a broadcast process.

A Suppaal model consists of a set of global integer variables, V^g, a set of global clocks, X^g, a set of channels, Σ, a set of scalarsets, Ω, and multiple Suppaal processes with local variables and clocks. The processes are created by instantiation of templates, and therefore we proceed with the definition of these templates over V^g, X^g, Ω and Σ.

Definition 3.8 (Suppaal template) *A Suppaal template over V^g, X^g, Ω and Σ is a tuple $T = (L, L^0, lt, X, V, S, I, vt, init, E)$, where*

- *L is a finite set of locations,*
- *$L^0 \in L$ is the initial location,*
- *$lt : L \to \{regular, urgent, committed\}$ assigns a type to every location,*
- *X is a finite set of local clocks (assume $X^g \cap X = \emptyset$),*
- *V is a finite set of local bounded integer variables (assume $V^g \cap V = \emptyset$),*
- *$S \subseteq \Omega$ is a finite set of scalarsets,*
- *$I : L \to \mathit{Inv}(X^g \cup X)$ assigns invariants to locations,*
- *$vt : V \to (\mathbb{N} \cup S)^*$ assigns a type to every local variable,*
- *$init : V \to \mathbb{Z}^+$ initializes every local variable, and*
- *$E \subseteq L \times \mathit{Sync}(\Sigma, V \cup V^g, S) \times G \times A \times L$ is a set of edges, where*
  - *G is a pair of guards in $IG(V \cup V^g, S) \times CG(X \cup X^g)$, and*
  - *A is a pair of assignments in $(IA(V \cup V^g, S))^* \times 2^{CA(X \cup X^g)}$.*

It is relatively straightforward to construct the mathematical SUPPAAL templates from the system description language. The only difficulty might occur when constructing the set S (and thereby vt and E). During the construction the template parameter is replaced by the scalarset it mimics. In the example of the Fischer protocol, this means that the set S for the template `P` contains only one scalarset, namely `process_id`, and the name `pid` in the edges of the template has been replaced by `process_id`.

The vt function can be explained by an example. Assume that v is a variable and that $vt(v) = (3, \alpha, 8)$, where α is a scalarset in S. This means that the variable v is a three dimensional array. Its first dimension is a regular dimension with size 3, its second dimension is indexed by scalarset α (its size is also determined by α, see below), and its third dimension also is a regular dimension, but with size 8. If $vt(v)$ equals the empty sequence, denoted by ϵ, then v is a regular variable.

The *init* function initializes the local variables. Our representation does not include the "decoding" scheme needed for arrays. However, this does not matter, since array entries are initialized to 0 by default.

We use indices to refer to the specific parts of templates. E.g., if T_i denotes a template, then X_i denotes the set of local clocks of T_i. With the previous definitions of SUPPAAL templates we are ready to define SUPPAAL models.

Definition 3.9 (SUPPAAL model) *A tuple* $M = (\Omega, s, V^g, vt, init, X^g, \Sigma, ct, \mathbb{T})$ *defines a* SUPPAAL *model, if*

- Ω *is a finite set of scalarsets,*
- $s : \Omega \to \mathbb{N}$ *defines the size of each scalarset,*
- V^g *is a finite set of global integer variables,*
- $vt : V^g \to (\mathbb{N} \cup \Omega)^*$ *assigns a type to every global variable,*
- $init : V^g \to \mathbb{Z}^+$ *initializes every global variable,*
- X^g *is a finite set of global clocks,*
- Σ *is a finite set of communication channels,*
- $ct : \Sigma \to \{regular, urgent\}$ *assigns a type to every channel, and*
- $\mathbb{T}$ *is a finite set of templates.*

The tuple M contains all information needed to construct the set of actual processes in the system, which defines the semantics of the model. We define these SUPPAAL processes as follows.

Definition 3.10 (SUPPAAL process) *Let M be a* SUPPAAL *model as above. A* SUPPAAL *process of M is a tuple $A_i = (T_i, \rho_i)$, where $T_i \in \mathbb{T}$ and $\rho_i : S_i \to \mathbb{N}$, such that $0 \leq \rho_i(\alpha) < s(\alpha)$ for all scalarsets $\alpha \in S$.*

As with templates, we use indices to refer to the components of processes. Thus, if A_i is a process, then T_i is its template, and V_i is the set of local variables of the process' template. Note that the number of valid scalarset valuation functions for a template T_i is finite. To be exact, the number of possibilities equals $\prod_{\alpha \in S_i} s(\alpha)$.

Note that SUPPAAL processes of M that originate from the same template have equal sets of local variables, clocks and scalarsets. To simplify the explanation of the semantics of M, which we define in the next section, we "flatten" the presentation. That is, we define the set of processes $\mathbb{A}$ associated with M as uniquely renamed SUPPAAL processes of M:

$$\mathbb{A} = \{\, rename_{(T_i,\rho_i)}((T_i,\rho_i)) \mid (T_i,\rho_i) \text{ is a SUPPAAL process of } M \,\} \tag{3}$$

By subscribing the *one-to-one* renaming functions with (T_i,ρ_i), we want to express that these renaming functions are unique in the sense that the local clocks, variables and constants of different processes share no elements. This allows us to merge the sets of clocks, variables and variable type functions of all processes of a SUPPAAL model. From now on, *Var* denotes the set of all variables, *tVar* denotes the set of all variable type mappings and *Clock* denotes the set of all clocks. As will become clear later, this assumption, which can be made without loss of generality, is very convenient for the definition of the semantics of the model.

We can easily extract the mathematical description of the model from any SUPPAAL system description. From this mathematical description, we can generate the set of processes $\mathbb{A}$ (thereby choosing suitable renaming functions). Moreover, we can derive the partial equivalence function $equiv^V_{ij} : V_i \to V_j$ for every pair of *processes* A_i and A_j:

$$equiv^V_{ij}(a) = \begin{cases} b & \text{if } T_i = T_j \text{ and } rename^{-1}_{(T_i,\rho_i)}(a) = rename^{-1}_{(T_j,\rho_j)}(b) \\ \uparrow & otherwise \end{cases}$$

This equivalence function links the local variables of processes which originate from the same template. Similarly, we can derive the partial equivalence function $equiv^X_{ij} : X_i \to X_j$ for local clocks. It is straightforward to see that if $equiv_{ij}(a)$ is defined, then $equiv_{ji} \circ equiv_{ij}(a) = a$.

Syntactical restrictions on SUPPAAL models

The mathematical description of a SUPPAAL model as defined previously should satisfy a number of restrictions to be syntactically correct. For instance, a reference to an array of integers must contain the right number of dimensions given by the *tVar* function. We do not elaborate on these well-understood restrictions here. Instead, we discuss restrictions concerning the newly introduced scalarsets.

First, we need to introduce the concept of *well-formed* integer expressions. Therefore, we need the help of a projection $[\,]_i$ which selects elements from arrays or sequences. This projection function is defined as follows:

$$[(e_0, e_1, ..., e_n)]_i = \begin{cases} e_i & \text{if } 0 \le i \le n \\ \uparrow & \text{otherwise} \end{cases}$$

Informally, an integer expression is well-formed if it is not an array, or if it is an array of which all scalarset dimensions are indexed by scalarset constants of the same type.

Definition 3.11 (Well-formedness) *Let* $exp \in IX(V^g \cup V_i, S_i)$ *be an integer expression of some process. We call exp well-formed, if exp is a simple integer expression, or if* $exp = a[i_0]...[i_m]$ *such that if* $[tVar(a)]_k = \alpha$*, then* $i_k = \alpha$ *for all* $\alpha \in \Omega$ *and for all* $0 \le k \le m$.

This concept of well-formedness can easily be lifted to well-formedness of synchronizations, integer assignments, integer guards and edges. For instance, assume that we have a SUPPAAL model with a scalarset `id` with size 3, and a variable array `a`, such that $tVar(\texttt{a}) = (\texttt{id})$. This means that `a` is a one-dimensional array indexed by the scalarset `id`. Moreover, there is a template in this model which is parameterized with this scalarset, and which has local integer variable `v`. Figure 4 depicts a well-formed edge of this template, and figure 5 depicts a non well-formed edge.

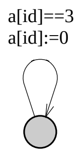

Figure 4: A well-formed edge.

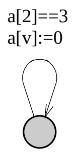

Figure 5: A non well-formed edge.

We also distinguish a special subset of non well-formed integer expressions and synchronizations.

Definition 3.12 ((α, n)-malformedness) *Let $a[i_0]...[n]...[i_m]$ be a non well-formed integer expression or synchronization. If the $n \in \{0, ..., s(\alpha)-1\}$ is the* unique *cause of this since it indexes an α dimension, then we call this integer expression or synchronization (α, n)-malformed.*

Again, we can easily lift this concept to edges: an edge is (α, n)-malformed, iff it contains (α, n)-malformed integer expressions or synchronizations. For instance, assume the same context as associated with the figures above and consider figure 6, which contains three $((\texttt{id}, n))$-malformed edges: one for every $n \in \{0, 1, 2\}$. Note that the edge of figure 5 is *not* (α, n)-malformed for any α or n.

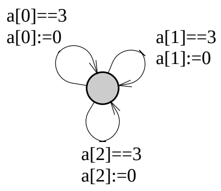

Figure 6: $(\texttt{id}, n)$-malformed edges (where $n \in \{0, 1, 2\}$).

Now we can state the restrictions concerning scalarsets in a SUPPAAL model. In short, there may be no symmetry breaking operations on scalarsets and we have formulated the following restrictions to achieve this:

(1) Consider a scalarset $\alpha \in S_i$ of process A_i. We can use α in only three ways:

  a) Assign the scalarset to a regular variable: $v := \alpha$ may appear in process A_i.

  b) Use the scalarset in guards: $v = \alpha$ and $v \neq \alpha$ may appear in process A_i.

c) We can use α to index α dimensions of arrays: $a[i_0]...[i_{k-1}][\alpha][i_{k+1}]...[i_n]$, where $a \in Var \cup \Sigma$, may appear in process A_i, if $[tVar(a)]_k = \alpha$. This means that well-formed integer expressions may appear in our model.

Thus, the scalarsets in processes may not be used in arithmetical expressions, clock guards and invariants.

We can compute a set of variables $used_\alpha$ for each scalarset α as follows: $a \in used_\alpha$ if and only if there exists an assignment $a := \alpha$ or there exists a guard $a = \alpha$ or $a \neq \alpha$ in the model.

(2) Let $v \in used_\alpha$ for some scalarset α. For instance, the global variable `id` in the Fischer protocol example at the end of section 3.1 is an element of $used_{\texttt{process\_id}}$.

a) v is initialized to a value $\notin \{0, ..., |\alpha| - 1\}$.

b) $used_\alpha \cap used_\beta = \emptyset$ for all scalarsets α and β.

c) v can *only* be used in assignments and guards as in (1a) and (1b) or we can assign an integer value z to v, or we can use v in a guard of the form $v = z$ or $v \neq z$, such that $z \notin \{0, 1, ..., |\alpha| - 1\}$.

Thus, v can neither be used in arithmetical expressions, nor can it be used to index arrays. Note that our Fischer example satisfies these conditions: `id` is initialized to -1, the Fischer template only assigns -1 or the scalarset constant to `id`, and it compares `id` only to -1 or to the scalarset constant.

(3) An edge $e = (src, \sigma, g, a, dst) \in E_i$ of a SUPPAAL model should either be well-formed, or it should be (α, n)-malformed, such that:

– the edge is not (β, m)-malformed for any $\beta \neq \alpha$ or $m \neq n$, and

– for every $n' \in \{0, ..., |\alpha| - 1\} \setminus \{n\}$ there exists an (α, n')-malformed edge $e' = (src, \sigma', g', a', dst) \in E_i$ such that σ', g' and a' only syntactically differ from σ, g and a in that the appearances of n as α dimension index have been replaced by n'.

For instance, the example belonging to figure 6 satisfies these constraints, since every $(\texttt{id}, n)$-malformed edge is not $(\texttt{id}, m)$-malformed for an $m \neq n$ (for instance, an edge labeled with the guard `a[1]==3` and the assignment `a[2]:=0` is not allowed). Moreover, there are the three required "equivalent" $(\texttt{id}, n)$-malformed edges: one for each $n \in \{0, 1, 2\}$.

Note that this restriction allows processes to "reset" scalarset dimensions of arrays without breaking the symmetry, which can be very convenient if not necessary for the modeling of many systems.

The restrictions above are very similar to those imposed by Ip and Dill, except for item (2). In contrast with their theory, we allow the assignment of a "scalarset type" to a regular variable (these variables are caught by the *used*-sets). Our motivation is based on the Fischer protocol, which would not fit in our framework without this extension, although it clearly exhibits full symmetry. With the previous restrictions we are able to prove in section 4 that our proposed symmetry reduction technique is sound.

3.3 Semantics of a Suppaal model

The semantics of a Suppaal model is, as with regular timed automata, defined by an infinite transition system which originates from the set of (renamed!) Suppaal processes $\mathbb{A}$. The states of this transition system are defined as follows:

Definition 3.13 (State) *A state of a* Uppaal *model containing n processes is a tuple $(\vec{l}, v, \nu)$, where*

- *$\vec{l}$ is the location vector, such that $l_i \in L_i$ for all $0 \leq i \leq n-1$,*
- *$v : Var \to \mathbb{Z}^*$ is the variable valuation, which maps every variable to a value (or a tuple of values in case of an array)*[2]*, and*
- *$\nu : Clock \to \mathbb{R}_+$ is the clock valuation which maps every clock to a non-negative real number including zero.*

The set of variable valuations for some model is denoted by Λ, the set of clock valuations for some model is denoted by Γ, and the set of all states is denoted by $\mathbb{S}$.

This definition explains the need for the renaming functions in definition 3.9 on page 11. If we allow equal local names of variables or clocks in processes, then we would need a clock and variable valuation for every process.

There are three kinds of transitions between states of a Uppaal model. Before we define these, we specify how the assignments and guards are interpreted over the clock and variable valuations. Since the integer guards and integer assignments might contain scalarsets, we need the context of a Suppaal process (more precisely, the scalarset valuation ρ) for evaluation.

$$eval : IX(Var, \Omega) \times \mathbb{A} \times \Lambda \hookrightarrow \mathbb{Z}$$

$$eval : CG(Clock) \times \Gamma \to \{true, false\}$$

$$eval : IG(Var, \Omega) \times \mathbb{A} \times \Lambda \hookrightarrow \{true, false\}$$

These functions are defined in the usual way. Using these evaluation functions, we can easily define the integer assignment execution function and the clock reset execution function, whose types are given below.

$$exec : (IA(Var, \Omega))^* \times \mathbb{A} \times \Lambda \hookrightarrow \Lambda$$

$$exec : 2^{CA(Clock)} \times \Gamma \to \Gamma$$

Now we are ready to define the transitions of a Suppaal model (we assume that there are n processes). The first kind of transition is a simple action transition in which an individual process executes an edge labeled with the "empty" channel τ:

[2] We do not explicitly explain the encoding and decoding of these arrays, since it is enough to assume that this happens in a consistent way.

Definition 3.14 (Simple action transition) *A tuple of states $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ is a simple action transition if an edge $(src, \tau, (\gamma_c, \gamma_v), (ac, av), dst) \in E_k$ exists, such that*

- $l_k = src$ *and* $l'_k = dst$ *and* $l'_j = l_j$ *for all* $j \neq k$,
- $eval((\gamma_c, \nu)) = eval((\gamma_v, A_k, v)) = true$,
- $v' = exec((av, A_k, v))$ *and* $\nu' = exec((ac, \nu))$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l'_i), \nu')) = true$ *for all* $0 \leq i \leq n-1$, *and*
- *if there exists a* l_i *such that* $lt_i(l_i) = committed$, *then* $lt_k(l_k) = committed$.

The second kind of transition is a synchronizing action transition in which two processes simultaneously execute an edge with matching synchronization labels.

Definition 3.15 (σ action transition) *A tuple $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ of states is called a σ action transition if there exists an edge $(src, (\sigma, !), (\gamma_c, \gamma_v), (ac, av), dst) \in E_k$ and an edge $(src', (\sigma, ?), (\gamma'_c, \gamma'_v), (ac', av'), dst') \in E_h$, such that :*

- $l_k = src$ *and* $l'_k = dst$ *and* $l_h = src'$ *and* $l'_h = dst'$ *and* $l'_i = l_i$ *for all* $i \neq h, k$,
- $eval((\gamma_c, \nu)) = eval((\gamma_v, A_k, v)) = eval((\gamma'_c, \nu)) = eval((\gamma'_v, A_h, v)) = true$,
- $v' = exec((av', A_h, exec(av, A_k, v))))$ *and* $\nu' = exec((\alpha'_c, exec((\alpha_c, \nu))))$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l'_i), \nu')) = true$ *for all* $0 \leq i \leq n-1$, *and*
- *if there exists a* l_i *such that* $lt_i(l_i) = committed$, *then* $lt_k(l_k) = committed$ *or* $lt_h(l_h) = committed$.

Note that the exclamation mark side of the synchronization precedes the question mark side of the synchronization with respect to assignments. The last kind of transition is transition in which only time elapses.

Definition 3.16 (δ delay transition) *A tuple of states $((\vec{l}, v, \nu), (\vec{l}, v, \nu'))$ is a δ delay transition, where $\delta \in \mathbb{R}^+$ and $\delta > 0$, if*

- $\nu'(x) = \nu(x) + \delta$ *for all* $x \in X$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l_i), \nu')) = true$ *for all* $0 \leq i \leq n-1$, *and*
- $lt_i(l_i) \neq committed$ *for all* $0 \leq i \leq n-1$,
- *if there exists an* i *such that* $lt_i(l_i) = urgent$, *then no state* r *exists such that* $((\vec{l}, v, \nu), r)$ *is a simple action transition or* σ *action transition for some* $\sigma \in \Sigma$,
- *no state* r *exists such that* $((\vec{l}, v, \nu), r)$ *is a* σ *action transition for some* $\sigma \in \Sigma$ *such that* $ct(\sigma) = urgent$.

With the definitions of states and the three transitions we have defined the structure of our transition system. We finish with describing the *initial state* of a SUPPAAL model. The location vector of this state is defined by the L^0_i for every process i, the initial variable valuation is given by the *init* functions, and finally, the initial clock valuation assigns 0 to all clocks in the model.

Definition 3.17 (Run) *A finite or infinite sequence of states $s_0, s_1, ...$ is a run, if s_0 is the initial state, and (s_i, s_{i+1}) is a simple action transition, a σ action transition, or a δ delay transition for all i.*

For a SUPPAAL model M we let $\mathcal{R}(M)$ denote the set of all runs of M, which thus captures the behavior of M. Since the theory of symmetry that we adopt (summarized in section 2) is solely concerned with reachability of states, we limit ourselves to reachability properties. Let us assume that we have a set of *state properties* Φ, for whose elements ϕ we can easily say whether they are true or false in some state.

Definition 3.18 (Reachability) *For a* SUPPAAL *model M and a state property ϕ, we say that ϕ is reachable in M, denoted by $M \models \exists\Diamond\phi$, if a run $s_0, s_1, ... \in \mathcal{R}(M)$ exists such that ϕ is true in some s_i of that run.*

The model checking engine of UPPAAL can decide whether or not $M \models \exists\Diamond\phi$. It constructs a finite abstraction of the transition system of M on-the-fly, using *difference bounded matrices* for symbolic representation of the clock valuation of the state [6, 11, 3]. This finite abstraction is then treated by a classical finite state model checking algorithm as depicted in figure 1 on page 3. It is not very difficult to adjust the engine to take symmetry into account, as is schematically depicted in figure 2 on page 5.

4 Extraction of automorphisms

In this section we extract automorphisms from an SUPPAAL model which has been extended with scalarsets. First, we define the so-called swap functions, and second, we prove that these swap functions are automorphisms.

We assume the context of a SUPPAAL model $M = (\Omega, s, V^g, vt, init, X^g, \Sigma, ct, \mathbb{T})$ which gives rise to n (uniquely renamed!) processes, denoted by $A_i = (T_i, \rho_i)$.

4.1 Defining the automorphisms

As Ip and Dill we define permutations on the state graph of, in our case, a SUPPAAL model. Part of the state of a SUPPAAL model consists of local contributions of the various processes of the model. Moreover, the behavior of the model is defined by the control structure of the processes. Therefore, we use processes which are (almost) syntactically equivalent to permute the state.

Definition 4.1 (Process swap) *Let A_i and A_j be processes of M that originate from the same template ($T_i = T_j$). We define a process swap $\xi_{ij} : \mathbb{S} \to \mathbb{S}$ as follows: $\xi_{ij}((\vec{l}, v, \nu)) = (\vec{l}', v', \nu')$, such that*

$$l'_i = l_j, l'_j = l_i, \textit{and } l'_k = l_k \textit{ for all } k \neq i, j$$

$$v'(a) = \begin{cases} v(equiv^V_{ij}(a)) & \textit{if } a \in V_i \\ v(equiv^V_{ji}(a)) & \textit{if } a \in V_j \\ v(a) & \textit{otherwise} \end{cases}$$

$$\nu'(x) = \begin{cases} \nu(equiv^X_{ij}(x)) & \textit{if } x \in X_i \\ \nu(equiv^X_{ji}(x)) & \textit{if } x \in X_j \\ \nu(x) & \textit{otherwise} \end{cases}$$

Lemma 4.1 *A process swap is its own inverse.*

Proof. We prove that $\xi_{ij} \circ \xi_{ij}((\vec{l}, v, \nu)) = \xi_{ij}((\vec{l'}, v', \nu')) = (\vec{l''}, v'', \nu'')$ such that $\vec{l''} = \vec{l}$, $v'' = v$, and $\nu'' = \nu$. We split the proof in three parts:

- $\vec{l''} = \vec{l}$. From definition 4.1 we know that $l'_i = l_j$ and $l'_j = l_i$. Applying ξ_{ij} again gives us that $l''_i = l'_j$ and $l''_j = l'_i$. Thus, $l''_i = l_i$, and $l''_j = l_j$ and $l''_k = l_k$ for all $k \neq i, j$.
- $v'' = v$. We prove that $v''(a) = v(a)$ for all variables a in three cases:
  - $a \notin V_i \cup V_j$. We see in definition 4.1 that the value of a remains unchanged. Thus $v''(a) = v'(a) = v(a)$.
  - $a \in V_i$. In definition 4.1 we read that $v'(a) = v(equiv_{ij}(a))$. Since the equivalence function is a bijection from V_i to V_j, we know that $equiv_{ij}(a) \in V_j$. Applying another process swap thus gives us that

    $$v''(a) = v(equiv_{ji} \circ equiv_{ij}(a))$$

    At the end of section 3.2 we explained that $equiv_{ij} \circ equiv_{ji} = \mathbf{id}$. Therefore, we can say that $v''(a) = v(a)$.
  - $a \in V_j$. The proof of this case is similar as the proof in the previous item.
- $\nu'' = \nu$. We can proof this by an argument similar to the one in the previous item.

□

Next, we define the multiple process swap.

Definition 4.2 (Multiple process swap) *Let $\alpha \in \Omega$ be a scalarset and let $0 \leq i \neq j < s(\alpha)$. A multiple process swap is the composition $\xi^{\alpha}_{ij} = \xi_{k_1k_2} \circ \xi_{k_3k_4} \circ ... \circ \xi_{k_{2m-1}k_{2m}}$, such that for all processes $A_{k_1}, ..., A_{k_{2m}}$ the following holds:*

- *$T_{k_{2p-1}} = T_{k_{2p}}$ for all $1 \leq p \leq m$,*
- *$\rho_{k_{2p-1}}(\alpha) = i$ and $\rho_{k_{2p}}(\alpha) = j$ for all $1 \leq p \leq m$,*
- *$\rho_{k_{2p-1}}(\beta) = \rho_{k_{2p}}(\beta)$ for all $1 \leq p \leq m$ and for all $\beta \neq \alpha$.*

Moreover, there are no process indices k_{x-1} and k_x such that the three items above are satisfied, and $\xi_{k_{x-1}k_x}$ is not in the composition. (Note that $\xi_{ij} = \xi_{ji}$).

Lemma 4.2 *A process is swapped at most once by a multiple process swap.*

Proof. Consider some process A_p such that $\rho_p(\alpha) = i$. Equation 3 on page 12 allows us to conclude that there is *exactly* one other process A_q which originates from the same template, $\rho_q(\alpha) = j$, and whose other scalarsets match those of A_p. Therefore, only the process swap ξ_{pq} (or, equivalently, ξ_{qp}) involving A_p can be present in the multiple process swap. □

Lemma 4.3 *If process A_p is not swapped by a multiple process swap ξ^{α}_{ij}, then $\rho_p(\alpha) \notin \{i, j\}$.*

Proof. We prove that if $\rho_p(\alpha) \in \{i, j\}$, then the process is swapped, which is logically equivalent to our lemma. Thus, we assume $\rho_p(\alpha) = i$. Equation 3 on page 12 allows us to conclude that there is *exactly* one other process A_q which originates from the same template, $\rho_q(\alpha) = j$, and whose other scalarsets match those of A_p. Therefore, the process swap ξ_{pq} must be in the multiple process swap ξ^α_{ij} according to definition 4.2. A similar argument holds for the situation $\rho_p(\alpha) = j$. □

Lemma 4.4 *Consider four processes A_p, $A_{p'}$, A_q and $A_{q'}$ and two process swaps $\xi_{p,p'}$ and $\xi_{q,q'}$. If the four processes are all different, then $\xi_{p,p'} \circ \xi_{q,q'} = \xi_{q,q'} \circ \xi_{p,p'}$.*

Proof. In definition 4.1 we see that the process swaps are orthogonal, because they only swap the *local* contributions to the state of the processes. Thus, if the four processes are different, then it does not matter in which order we apply the process swaps. □

The second step swaps the dimensions of integer variable arrays which are indexed by some scalarset, and it swaps the integer variables which are target of an assignment with a scalarset constant.

Definition 4.3 (Data swap) *Let $\alpha \in \Omega$ be a scalarset and let $0 \leq i \neq j < s(\alpha)\}$. A data swap is defined as $\zeta^\alpha_{ij}((\vec{l}, v, \nu)) = (\vec{l}, v', \nu)$, such that: for every regular variable a:*

$$v'(a) = \begin{cases} i & \text{if } v(a) = j \text{ and } a \in used_\alpha \\ j & \text{if } v(a) = i \text{ and } a \in used_\alpha \\ v(a) & \text{otherwise} \end{cases}$$

And for every n dimensional integer array a:

$$v'(a[i_0]...[i_{n-1}]) = v(a[(i_0)_{\alpha,0}]...[(i_{n-1})_{\alpha,n-1}])$$

where the functions $()_{\alpha,k} : \mathbb{N} \to \mathbb{N}$ are defined for array a as:

$$(c)_{\alpha,k} = \begin{cases} i & \text{if } c = j \text{ and } [tVar(a)]_k = \alpha \\ j & \text{if } c = i \text{ and } [tVar(a)]_k = \alpha \\ c & \text{otherwise} \end{cases}$$

Lemma 4.5 *A data swap is its own inverse.*

Proof. We prove that $\zeta^\alpha_{ij} \circ \zeta^\alpha_{ij}((\vec{l}, v, \nu)) = \zeta^\alpha_{ij}((\vec{l'}, v', \nu')) = (\vec{l''}, v'', \nu'')$ such that $\vec{l''} = \vec{l}$, $v'' = v$, and $\nu'' = \nu$. We split the proof in three parts:

- $\vec{l''} = \vec{l}$. From definition 4.3 we know that the data swap does not alter the location vector. Therefore, $\vec{l''} = \vec{l}$.
- $v'' = v$. First, we proof that $v''(a[i_0]...[i_{n-1}]) = v(a[i_0]...[i_{n-1}])$ for all n dimensional integer arrays a. Therefore, let us apply two data swaps to this array:

  $$v''(a[i_0]...[i_{n-1}]) = v(a[((i_0)_{\alpha,0})_{\alpha,0}]...[((i_{n-1})_{\alpha,n-1})_{\alpha,n-1}])$$

  Next, we prove that $((c)_{\alpha,k})_{\alpha,k} = c$. We distinguish three cases. First, $c = i$ and $[tVar(a)]_k = \alpha$. By definition $((c)_{\alpha,k})_{\alpha,k} = (j)_{\alpha,k} = i$. Second, $c = j$ and

$[tVar(a)]_k = \alpha$. Then $((c)_{\alpha,k})_{\alpha,k} = (i)_{\alpha,k} = j$. The third case encompasses all other situations, and thus $((c)_{\alpha,k})_{\alpha,k} = (c)_{\alpha,k} = c$ by definition 4.3.

Second, we must prove that $v''(a) = v(a)$ for all regular variables a. Again, we distinguish three cases. First, $v(a) = j$ and $a \in used_\alpha$. By definition $v'(a) = i$ and – of course – still $a \in used_\alpha$. Applying another swap thus gives us by definition that $v''(a) = j$. The second case, $v(a) = i$ and $a \in used_\alpha$, is similar. The third case encompasses all other situations, and again by definition 4.3 we know that $v''(a) = v'(a) = v(a)$.

- $\nu'' = \nu$, since the data swap does not alter the clock valuation.

□

Lemma 4.6 *Consider a process swap ξ_{pq} and a data swap ζ^α_{ij}. The order of application does not matter: $\xi_{pq} \circ \zeta^\alpha_{ij} = \zeta^\alpha_{ij} \circ \xi_{pq}$.*

Proof. Assume that $\xi_{pq} \circ \zeta^\alpha_{ij}((\vec{l}, v, \nu)) = \xi_{pq}((\vec{l'_1}, v'_1, \nu'_1)) = (\vec{l''_1}, v''_1, \nu''_1)$, and similarly $\zeta^\alpha_{ij} \circ \xi_{pq}((\vec{l}, v, \nu)) = \zeta^\alpha_{ij}((\vec{l'_2}, v'_2, \nu'_2)) = (\vec{l''_2}, v''_2, \nu''_2)$. We prove that $(\vec{l''_1}, v''_1, \nu''_1) = (\vec{l''_2}, v''_2, \nu''_2)$.

- $\vec{l''_1} = \vec{l''_2}$, because we now by definition 4.3 that the data swap does not alter the location vector.

- $v''_1 = v''_2$, thus $v''_1(a) = v''_2(a)$ for all (arrays of) integer variables a. We distinguish three cases:

  - $a \in V_p$. There are two situations. First, a can be a regular (non-array) variable. Then $v'_1(a)$ equals i if $v(a) = j$ and $a \in used_\alpha$, or j if $v(a) = i$ and $a \in used_\alpha$, or $v(a)$ otherwise. Similarly, $v'_1(b)$ equals i if $v(b) = j$ and $b \in used_\alpha$, or j if $v(b) = i$ and $b \in used_\alpha$, or $v(b)$ otherwise, where $b = equiv_{pq}(a)$. Note that b exists since a is a local variable of process A_p. Applying the process swap swaps the value of a with the value of b, thus:

    $$v''_1(a) = v'_1(b) = \begin{cases} i & \text{if } v(b) = j \text{ and } b \in used_\alpha \\ j & \text{if } v(b) = i \text{ and } b \in used_\alpha \\ v(b) & \text{otherwise} \end{cases}$$

    Next, we consider $v'_2(a)$, which results from a process swap. Thus: $v'_2(a) = v(b)$, where $b = equiv_{pq}(a)$. Applying the data swap gives us by definition:

    $$v''_2(a) = \begin{cases} i & \text{if } v'_2(a) = v(b) = j \text{ and } b \in used_\alpha \\ j & \text{if } v'_2(a) = v(b) = i \text{ and } b \in used_\alpha \\ v(b) & \text{otherwise} \end{cases}$$

    Therefore, $v''_1(a) = v''_2(a)$ for all regular variables a.
    Second, a can be a n-dimensional array of integers. First applying the data swap gives us that $v'_1(a[i_0]...[i_{n-1}]) = v(a[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$. The process swap then swaps all entries of a with $equiv_{pq}(a)$:

    $$v''_1(a[i_0]...[i_{n-1}]) = v(equiv_{pq}(a)[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$$

On the other hand, first applying the process swap gives $v'_2(a[i_0]...[i_{n-1}]) = v(equiv_{pq}(a)[i_0]...[i_{n-1}])$. Next, we apply the data swap with the result that

$$v''_2(a[i_0]...[i_{n-1}]) = v(equiv_{pq}(a)[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$$

Thus, we conclude that $v''_1(a) = v''_2(a)$ for all (arrays of) integer variables a.

  – $a \in V_q$. We can prove this with a similar argument as appears in the previous item.
  – $a \notin V_p \cup V_q$. These variables are left unchanged by the process swap, as we can read in definition 4.1. Therefore, obviously $v''_1(a) = v''_2(a)$ for these variables.

- $\nu''_1 = \nu''_2$, because we now by definition 4.3 that the data swap does not alter the clock valuation.

□

Using the multiple process swap and the data swap we can define the permutations which permute the i-th and the j-th element of a scalarset consistently through the state.

Definition 4.4 (State swap) *Let α be a scalarset and let $0 \leq i \neq j < s(\alpha)$. Then, $\zeta^\alpha_{ij} \circ \xi^\alpha_{ij}$ is a state swap abbreviated by π^α_{ij}.*

Lemma 4.7 *A state swap is its own inverse.*

Proof. Consider a state swap $p_1 \equiv \zeta^\alpha_{ij} \circ \xi^\alpha_{ij} = \zeta^\alpha_{ij} \circ \xi_{k_1k_2} \circ \xi_{k_3k_4} \circ ... \circ \xi_{k_{2m-1}k_{2m}}$. Since a multiple process swap does always swap a process at most once (see lemma 4.2 on page 18), we can use lemma 4.4 and lemma 4.6 to rewrite this state swap to $p2 \equiv \xi_{k_{2m-1}k_{2m}} \circ ... \circ \xi_{k_1k_2} \circ \zeta^\alpha_{ij}$. Thus $p_1 = p_2$, and using lemma 4.1 and lemma 4.5 we can rewrite $p_1 \circ p_2$ in the following way:

$$\begin{aligned}
p_1 \circ p_2 &= \zeta^\alpha_{ij} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-1}k_{2m}} \circ \xi_{k_{2m-1}k_{2m}} \circ ... \circ \xi_{k_1k_2} \circ \zeta^\alpha_{ij} \\
&= \zeta^\alpha_{ij} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-3}k_{2m-2}} \circ \ \mathbf{id} \ \circ \xi_{k_{2m-3}k_{2m-2}} \circ ... \circ \xi_{k_1k_2} \circ \zeta^\alpha_{ij} \\
&= \zeta^\alpha_{ij} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-3}k_{2m-2}} \circ \xi_{k_{2m-3}k_{2m-2}} \circ ... \circ \xi_{k_1k_2} \circ \zeta^\alpha_{ij} \\
&= .... \\
&= \mathbf{id}
\end{aligned}$$

Thus, a state swap is its own inverse. □

In the next section we define swap functions on the *syntax* of our models. We use these to prove that the state swaps as defined above are automorphisms.

4.2 Syntactical swaps

This section defines swaps on the syntax of a SUPPAAL model. We use these swaps in the soundness proof in the next section. In the remainder of this section we abbreviate the state swap π^α_{ij} by π, while maintaining the parameters α, i and j.

Definition 4.5 (Syntactical integer expression swap) *A swap of a syntactical integer expression is a function* $\pi^s_{pq} : IX(V^g \cup V_p, S_p) \to IX(V^g \cup V_q, S_q)$ *defined as follows:*

$$\pi^s_{pq}(exp) = \begin{cases} exp & \text{if } exp \in \mathbb{Z} \cup S_p \\ exp & \text{if } exp \in V \setminus V_p \\ equiv_{pq}(exp) & \text{if } exp \in V_p \\ v[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] & \text{if } exp \equiv v[e_1]...[e_m] \text{ and } v \in V \setminus V_p \\ equiv_{pq}(v)[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] & \text{if } exp \equiv v[e_1]...[e_m] \text{ and } v \in V_p \\ (\pi^s_{pq}(e_1) \oslash \pi^s_{pq}(e_2)) & \text{if } exp \equiv (e_1 \oslash e_2) \end{cases}$$

Lemma 4.8 *Let* $e \in IX(V^g \cup V_p, S_p)$ *be a well-formed expression of process* A_p*, let* π *be a state swap and let* π^s_{pq} *be a syntactical integer expression swap. If* $e \neq \alpha$ *and* $e \notin used_\alpha$*, then*

(1) If π *swaps* A_p *with* A_q*, then* $eval(e, A_p, [(\vec{l}, v, \nu)]_1) = eval(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1)$.

(2) If π *does not swap* A_p*, then* $eval(e, A_p, [(\vec{l}, v, \nu)]_1) = eval(e, A_p, [\pi(\vec{l}, v, \nu)]_1)$.

Proof. We proof all seven cases of definition 4.5 separately for both parts of the lemma.

- $e \in \mathbb{Z}$. We know that $eval(e, A_p, [(\vec{l}, v, \nu)_1]) = z$ by definition of the evaluation function. And $eval(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) = z$, since no variable interpretation is needed for the evaluation of a number, and the syntactical swap does not change such a number by definition. This proves part (1) and part (2) of the lemma.

- $e = \beta$ for a scalarset $\beta \neq \alpha$. We know by definition of the evaluation function that $eval(e, A_p, [(\vec{l}, v, \nu)]_1) = \rho_p(e)$. And therefore:

  $$eval(\pi^s_{pq}(e), [\pi(\vec{l}, v, \nu)]_1) = eval(e, [\pi(\vec{l}, v, \nu)]_1) = \rho_q(e)$$

  In definition 4.2 on page 18 we read that A_p and A_q do only differ in the value of their α scalarset. Therefore, $\rho_p(e) = \rho_q(e)$. This proves part (1) of the lemma. As for part (2), we say that the variable interpretation is not needed to evaluate scalarset e.

- $e \in V \setminus V_p$. Obviously, $eval(e, A_p, [(\vec{l}, v, \nu)_1]) = v(e)$. Second,

  $$eval(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) = eval(e, A_q, [\pi(\vec{l}, v, \nu)]_1) = eval(e, A_p, v') = v'(e)$$

  By definition, π's process swaps do not alter e, since e is not local to A_p. Moreover, π's data swap does also not alter the value of e, since we assumed in the lemma that $e \notin used_\alpha$. Therefore, $v'(e) = v(e)$. This proves part (1) of the lemma, and with a very similar argument we can easily prove part (2).

- $e \in V_p$. Obviously, $eval(e, A_p, [(\vec{l}, v, \nu)_1]) = v(e)$. Second,

  $$\begin{aligned} eval(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) &= eval(equiv_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) \\ &= eval(equiv_{pq}(e), A_q, v') \end{aligned}$$

  By definition, π's process swap does alter the value of e in the following way: $v'(e) = v(equiv_{pq}(e))$. Moreover, $v'(equiv_{pq}(e)) = v(equiv_{qp} \circ equiv_{pq}(e)) = v(e)$, since $equiv_{qp} \circ equiv_{pq} = \mathbf{id}$. The data swap does not alter the value of e, since we assumed in the lemma that $e \notin used_\alpha$. Thus, $eval(equiv_{pq}(e), A_q, v')$ equals $v(e)$. This proves part (1) of the lemma. As for part (2), note that A_p is not swapped. Therefore, the value of the local variable e is not changed.

- $e \equiv (e_1 \oslash e_2)$ By definition 3.4 on page 9 we know that e_1 and e_2 are simple integer expressions. For part (1) of the lemma we must prove that

  $$eval((e_1 \oslash e_2), A_p, [(\vec{l}, v, \nu)]_1) \quad = \quad eval(\pi^s_{pq}(e_1 \oslash e_2), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

  And thus by definition 4.5:

  $$\begin{gathered} eval(e_1, A_p, [(\vec{l}, v, \nu)]_1) \oslash eval(e_2, A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ eval(\pi^s_{pq}(e_1), A_q, [\pi(\vec{l}, v, \nu)]_1) \oslash eval(\pi^s_{pq}(e_2), A_q, [\pi(\vec{l}, v, \nu)]_1) \end{gathered}$$

  This can easily be proved by induction on the syntax of e_1 and e_2. The base is formed by the four previous items. The induction step is straightforward and we do not explicitly explain it. The proof of part (2) is very similar.

- $exp \equiv a[e_0]...[e_m]$ and $a \in V \setminus V_p$. We start with the proof of part (1) of the lemma. First we rewrite the first term in our lemma as follows:

  $$eval(a[e_0]...[e_m], A_p, v) = v(a[eval(e_0, A_p, v)]...[eval(e_m, A_p, v)])$$

  Second, we rewrite the second term in our lemma using the definition of the evaluation function and definition 4.5:

  $$\begin{gathered} eval(\pi^s_{pq}(a[e_0]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) = eval(a[\pi^s_{pq}(e_0)]...[\pi^s_{pq}(e_m)], A_q, v') \\ = \\ v'(a[eval(\pi^s_{pq}(e_0), A_q, v')]...[eval(\pi^s_{pq}(e_m), A_q, v')]) \end{gathered}$$

  where $v' \equiv [\pi(\vec{l}, v, \nu)]_1$. Note that π does not "alter" the values in the array a, since the process swap of π does not affect a (since $a \notin V_p$). Thus, entries of a are merely swapped around (see definition 4.3 of the data swap). Therefore, we can rewrite the previous term to:

  $$v(a[(eval(\pi^s_{pq}(e_0), A_q, v'))_{\alpha,0}]...[(eval(\pi^s_{pq}(e_m), A_q, v'))_{\alpha,m}])$$

  Thus, we must prove the following equality for all array dimensions k:

  $$eval(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (eval(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

  Again, we must consider all possible cases for the syntax of e_k. In definition 3.4 on page 9 we see that e_k must be a simple integer expression. Thus, we enumerate the possibilities:

  - $e_k \in \mathbb{Z}$. By definition, $eval(e_k, A_p, v) = e_k$, and $(eval(e_k, A_q, v'))_{\alpha,k} = (e_k)_{\alpha,k}$. Since we assumed that our expression is well-formed, this k-th dimension may not be a scalarset dimension. Therefore, $(e_k)_{\alpha,k} = e_k$.

- $e_k = \beta$ for some scalarset β. We know by definition that $\mathit{eval}(e_k, A_p, v) = \rho_p(e_k)$. As for the second part:

  $$(\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = (\rho_q(e_k))_{\alpha,k}$$

  We now can distinguish two cases. First, $\beta \neq \alpha$. Since the expression is well-formed, we know that the k-th dimension is not an α dimension. Thus we conclude that $(\rho_q(e_k))_{\alpha,k} = \rho_q(e_k)$. Since π swaps process A_p with A_q, we can conclude from definition 4.2 that the β constant of these processes is the same. Thus: $\rho_p(e_k) = \rho_q(e_k)$.

  Second, $\beta = \alpha$. Since π swaps process A_p with A_q, we know that $\rho_p(e_k) = i$, and $\rho_q(e_k) = j$. Thus, $(\rho_q(e_k))_{\alpha,k} = (j)_{\alpha,k} = i$ by definition, because the k-th dimension now is an α dimension.
- $e_k \in V^g \cup V_p$ or $e_k \equiv (e_1 \oslash e_2)$, where e_1 and e_2 are simple expressions. Since we assumed that the expression under consideration is well-formed, we conclude that dimension k is a non-scalarset dimension. Therefore,

  $$(\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = \mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

  The proof for this case can be found in the third, fourth and fifth main item of the proof of this lemma.

Next, we prove the second part of the lemma. With an argument similar to the one at the start of this item, we conclude that we must prove the equality

$$\mathit{eval}(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

where e_k is a simple integer expression. Again, we distinguish three cases.

- $e_k \in \mathbb{Z}$. By definition, $\mathit{eval}(e_k, A_p, v) = e_k$, and $(\mathit{eval}(e_k, A_p, v'))_{\alpha,k} = (e_k)_{\alpha,k}$. Since we assumed that our expression is well-formed, this k-th dimension may not be a scalarset dimension. Therefore, $(e_k)_{\alpha,k} = e_k$.
- $e_k = \beta$ for some scalarset β. We know by definition that $\mathit{eval}(e_k, A_p, v) = \rho_p(e_k)$. As for the second part:

  $$(\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = (\rho_p(e_k))_{\alpha,k}$$

  We now can distinguish two cases. First, $\beta \neq \alpha$. Since the expression is well-formed, we know that the k-th dimension is not an α dimension. Thus, $(\rho_p(e_k))_{\alpha,k} = \rho_p(e_k)$. Second, $\beta = \alpha$. Since π does not swap process A_p, we know by lemma 4.3 on page 18 that $\rho_p(e_k) \neq i, j$. Thus, $(\rho_p(e_k))_{\alpha,k} = \rho_p(e_k)$.
- $e_k \in V^g \cup V_p$ or $e_k \equiv (e_1 \oslash e_2)$, where e_1 and e_2 are simple expressions. Since we assumed that the expression under consideration is well-formed, we conclude that dimension k is a non-scalarset dimension. Therefore,

  $$(\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = \mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1)$$

  The proof for this case can be found in the third, fourth and fifth main item of the proof of this lemma.

- $exp \equiv a[e_0]...[e_m]$ and $a \in V_p$. We start with the first part of the lemma. We rewrite the first term in our lemma as follows:

  $$eval(a[e_0]...[e_m], A_p, v) = v(a[eval(e_0, A_p, v)]...[eval(e_m, A_p, v)])$$

  Second, we rewrite the second term in our lemma using the definition of the evaluation function and definition 4.5:

  $$eval(\pi^s_{pq}(a[e_0]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) = eval(equiv_{pq}(a)[\pi^s_{pq}(e_0)]...[\pi^s_{pq}(e_m)], A_q, v')$$

  Again, we can rewrite this to

  $$v'(equiv_{pq}(a)[eval(\pi^s_{pq}(e_0), A_q, v')]...[eval(\pi^s_{pq}(e_m), A_q, v')])$$

  Note that π consists of a multiple process swap and a data swap. The process swaps swap entire arrays, while the data swap swaps dimensions of arrays. More precisely, $v'(equiv_{pq}(a)[i_0]...[i_m]) = v(equiv_{qp} \circ equiv_{pq}(a)[(i_0)_{\alpha,0}]...[(i_m)_{\alpha,m}])$. Thus, we can rewrite the previous term to:

  $$v(a[(eval(\pi^s_{pq}(e_0), A_q, v'))_{\alpha,0}]...[(eval(\pi^s_{pq}(e_m), A_q, v'))_{\alpha,m}])$$

  Thus, we must prove the following equality for all array dimensions k:

  $$eval(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (eval(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

  We have already done this in the previous item.

  The proof of the second part of the lemma is very similar to the proof in the previous item, and we do not explicitly explain it.

□

Lemma 4.9 *Let us consider some (β, n)-malformed integer expression in dimension d, say $a[e_0]...[e_{d-1}][n][e_{d+1}]...[e_m]$. There exists an $n' \in \{0, ..., |\beta| - 1\}$ such that*

$$\begin{gathered} eval(a[e_0]...[e_{d-1}][n][e_{d+1}]...[e_m], A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ eval(\pi^s_{pq}(a[e_0]...[e_{d-1}][n'][e_{d+1}]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \end{gathered}$$

Proof. We can use the same argument as in the last two items of the proof of lemma 4.8 to argue that we first must prove that

$$eval(e_k, A_p, [(\vec{l}, v, \nu)]_1 = (eval(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

for every array dimension $k \neq d$. We can use the last two items of the proof of lemma 4.8 to prove this.

Now we consider the remaining situation for dimension d. Since the expression is (β, n)-malformed in this dimension, we thus know that $n \in \{0, ..., |\beta| - 1\}$. From definition 4.5 it is clear that $\pi^s_{pq}(n) = n$. Moreover, we do not need the variable interpretation for the evaluation of n. Thus, we must show that we can find an $n' \in \{0, ..., |\beta| - 1\}$ such that:

$$n = (n')_{\alpha,d}$$

It is not difficult to see that the following definition of n' satisfies this equality:

$$n' = \begin{cases} i & \text{if } n = j \text{ and } \alpha = \beta \\ j & \text{if } n = i \text{ and } \alpha = \beta \\ n & \text{otherwise} \end{cases}$$

(See definition 4.3 on page 19 for the definition of the $()_{\alpha,k}$ function.) $\square$

We can also define syntactical swaps of the integer assignments. These functions take an integer assignment of process A_p and change the syntax in such a way that it becomes an integer assignment of process A_q, if A_p and A_q originate from the same template.

Definition 4.6 (Syntactical integer assignment swap) *A swap of a syntactical integer assignment is a function $\pi^s_{pq} : IA(V^g \cup V_p, S_p) \to IA(V^g \cup V_q, S_q)$ defined as follows:*

$$\pi^s_{pq}(a) = \begin{cases} b := \pi^s_{pq}(exp) & \text{if } a \equiv b := exp \text{ and } b \notin V_p \\ equiv_{pq}(b) := \pi^s_{pq}(exp) & \text{if } a \equiv b := exp \text{ and } b \in V_p \\ b[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] := \pi^s_{pq}(exp) & \text{if } a \equiv b[e_1]...[e_m] := exp \\ & \quad \text{and } b \notin V_p \\ equiv_{pq}(b)[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] := \pi^s_{pq}(exp) & \text{if } a \equiv b[e_1]...[e_m] := exp \\ & \quad \text{and } b \in V_p \end{cases}$$

where the function π^s_{pq} which appears right of the large bracket is the syntactical integer expression swap.

The next lemma states that an integer assignment in process A_p has the same effect as an integer assignment in process A_q modulo symmetry, if they are swapped by a state swap.

Lemma 4.10 *Consider a well-formed integer assignment $ia \in IA(V^g \cup V_p, S_p)$ of A_p, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $v' = exec((ia, A_p, v))$, a syntactical assignment swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then $[\pi(\vec{l'}, v', \nu')]_1 = exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_1 = exec(ia, A_p, [\pi(\vec{l}, v, \nu)]_1))$.

Proof. We use the following abbreviations in our proof: $v'' = [\pi(\vec{l'}, v', \nu')]_1$ and $v''' = exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1))$. We start with part (1) of the lemma, for which we distinguish two cases. First, ia is of the form $a := exp$. In this case, we distinguish another four cases:

- $a \in V_p$ and $a \in used_\beta$ for some scalarset β. In this case, exp may only be equivalent to to a value $z \in \mathbb{Z} \setminus \{0, 1, ..., s(\beta) - 1\}$ or to β (see restriction (2c) on page 14). In the first situation,

$$v''' = exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1)) =$$
$$exec((equiv_{pq}(a) := \pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)) =$$
$$exec((equiv_{pq}(a) := z, A_q, [\pi(\vec{l}, v, \nu)]_1))$$

Thus, we can define v''' as follows:

$$v'''(b) = \begin{cases} z & \text{if } b = equiv_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Similarly, we can deduct that

$$v'(b) = \begin{cases} z & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$

And therefore we can conclude that

$$v''(b) = [\pi(\vec{l'}, v', \nu')]_1(b) = \begin{cases} z & \text{if } b = equiv_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Concluding, $v'' = v'''$. In the second case we can rewrite v''' as follows:

$$\begin{aligned} v''' &= exec((\pi^s_{pq}(a := \beta), A_q, [\pi(\vec{l}, v, \nu)]_1)) \\ &= exec((equiv_{pq}(a) := \beta, A_q, [\pi(\vec{l}, v, \nu)]_1)) \end{aligned}$$

Thus, we can define v''' as follows:

$$v'''(b) = \begin{cases} \rho_q(\beta) & \text{if } b = equiv_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Similarly, we can deduct that

$$v'(b) = \begin{cases} \rho_p(\beta) & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$

Now we distinguish two further possibilities. First, $\beta = \alpha$. Then we now by definition 4.2 that $\rho_p(\beta) = i$, and that $\rho_q(\beta) = j$. Applying the state swap thus changes the value of a and swaps it with its equivalent in process A_q:

$$v''(b) = [\pi(\vec{l'}, v', \nu')]_1(b) = \begin{cases} j & \text{if } b = equiv_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

We can conclude that $v'' = v'''$. Second, $\beta \neq \alpha$. In this case we now by definition 4.2 that $\rho_p(\beta) = \rho_q(\beta)$. Applying the state swap to v' does only swap the variable a with its equivalent in process A_q:

$$v''(b) = [\pi(\vec{l'}, v', \nu')]_1(b) = \begin{cases} \rho_p(\beta) & \text{if } b = equiv_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Again, we can conclude that $v'' = v'''$.

- $a \in V^g$ and $a \in used_\beta$ for some scalarset β. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

- $a \in V_p$ and $a \notin \mathit{used}_\beta$ for all scalarsets β. Again, we rewrite v''' using the definitions.
$$v''' = \mathit{exec}((\pi^s_{pq}(a := \mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1))$$
$$=$$
$$\mathit{exec}((\mathit{equiv}_{pq}(a) := \pi^s_{pq}(\mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1))$$
Thus, we can define v''' as follows:
$$v'''(b) = \begin{cases} \mathit{eval}(\pi^s_{pq}(\mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$
Similarly, we can deduct that
$$v'(b) = \begin{cases} \mathit{eval}(\mathit{exp}, A_p, v) & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$
And therefore we can conclude that
$$v''(b) = [\pi(\vec{l'}, v', \nu')]_1(b) = \begin{cases} \mathit{eval}(\mathit{exp}, A_p, v) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$
To prove $v'' = v'''$ we prove that $\mathit{eval}(\mathit{exp}, A_p, v) = \mathit{eval}(\pi^s_{pq}(\mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1)$. We know that $\mathit{exp} \neq \beta$ for all scalarsets β, since $a \notin \mathit{used}_\beta$. Moreover, according to restriction (3) on page 14, $\mathit{exp} \notin \mathit{used}_\beta$ for all scalarsets β. Therefore, we can immediately use part (1) of lemma 4.8 on page 22 to conclude that $v'' = v'''$.

- $a \in V^g$ and $a \notin \mathit{used}_\beta$ for all scalarsets β. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

Second, ai is of the form $a[i_0]...[i_m] := \mathit{exp}$. By the restrictions on the syntax of the model stated on page 13, we know that $a, \mathit{exp} \notin \mathit{used}_\beta$ and $\mathit{exp} \neq \beta$ for all scalarsets β. We only distinguish two other cases:

- $a \in V_p$. We can rewrite v''' as follows:
$$v''' = \mathit{exec}(\pi^s_{pq}(a[i_0]...[i_m] := \mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1)$$
$$=$$
$$\mathit{exec}(\mathit{equiv}_{pq}(a)[\pi^s_{pq}(i_0)]...[\pi^s_{pq}(i_m)] := \pi^s_{pq}(\mathit{exp})), A_q, [\pi(\vec{l}, v, \nu)]_1)$$
This enables us to define v''' as follows:
$$v'''(b) = \begin{cases} \mathit{eval}(\pi^s_{pq}(\mathit{exp}), A_q, [\pi(\vec{l}, v, \nu)]_1) & \text{if } b = \mathit{equiv}_{pq}(a)[i'_0]...[i'_m] \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$
where $i'_k = \mathit{eval}(\pi^s_{pq}(i_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Next, we construct a definition for v'' using the following definition of v':
$$v'(b) = \begin{cases} \mathit{eval}(\mathit{exp}, A_p, v) & \text{if } b = a[\mathit{eval}(i_0, A_p, v)]...[\mathit{eval}(i_m, A_p, v)] \\ v(b) & \text{otherwise} \end{cases}$$

The process swap of π interchanges the whole array with an equivalent array in process A_q, and it swaps around the data in the array:

$$v''(b) = \begin{cases} eval(exp, A_p, v) & \text{if } b = equiv_{pq}(a)[(eval(i_0, A_p, v))_{\alpha,0}]... \\ & \quad [(eval(i_m, A_p, v))_{\alpha,m}] \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Thus, we must prove the following two statements in order to prove that $v'' = v'''$:

- $eval(exp, A_p, v) = eval(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Due to the form of exp, we can immediately use part (1) of lemma 4.8 on page 22 to conclude that this statement is true.
- $(eval(i_k, A_p, v))_{\alpha,k} = eval(\pi^s_{pq}(i_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$ for all simple integer expressions i_k. There are many possibilities for the form of i_k. If $i_k \neq \alpha$ and $i_k \notin used_\alpha$, then we can immediately use part (1) of lemma 4.8 on page 22 in conjunction with the fact that the array assignment is well formed to conclude that this statement is true.

  Now let us consider the other cases. First, assume that $i_k = \alpha$. Since A_p is swapped, we know by definition 4.2 that $\rho_p(i_k) = i$, and that $\rho_q(equiv_{pq}(i_k)) = j$. Applying the definition of the $()_{\alpha,k}$ function gives us that $(eval(i_k, v))_{\alpha,k} = j$. Moreover,

  $$eval(\pi^s_{pq}(i_k), [\pi(\vec{l}, v, \nu)]_1) = eval(equiv_{pq}(i_k), [\pi(\vec{l}, v, \nu)]_1) = \rho_q(equiv_{pq}(i_k))$$

  This proves the first case. The second case, $i_k \in used_\alpha$, cannot exist, since variables in the *used* sets may not be used to index arrays according to the restrictions on page 13.

- $a \in V^g$. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

This concludes part (1) of the lemma. As for part (2) we only say that the proof is very similar – with respect to the structure – to the proof given above. The key observation is that the state swap π now does not affect local variables of a, and that $\rho_p(\alpha) \neq i, j$ by lemma 4.3 on page 18.

□

Lemma 4.11 *Consider a (β, n)-malformed integer assignment $ia \in IA(V^g \cup V_p, S_p)$, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $v' = exec((ia, A_p, v))$, a syntactical assignment swap π^s_{pq} and a state swap π. We can find a $n' \in \{0, ..., |\beta| - 1\}$ such that for ia', which results from replacing malformation n by n' in ia, the following holds:*

(1) If π swaps A_p with A_q, then $[\pi(\vec{l'}, v', \nu')]_1 = exec((\pi^s_{pq}(ia'), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_1 = exec(ia', A_p, [\pi(\vec{l}, v, \nu)]_1))$.

Proof. We can follow the structure of the proof of the previous lemma. This gives the desired result without much effort, when used in conjunction with lemma 4.9 on page 25. □

Lemma 4.12 *Consider a well-formed integer guard $g \in IG(V^g \cup V_p, S_p)$ of A_p, a state $(\vec{l}, v, \nu)$, a syntactical assignment swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then $eval(g, A_p, [(\vec{l}, v, \nu)]_1) = eval((\pi^s_{pq}(g), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then $eval(g, A_p, [(\vec{l}, v, \nu)]_1) = eval((g, A_p, [\pi(\vec{l}, v, \nu)]_1))$.

Proof. We start by proving part (1) of the lemma. According to definition 3.5, the integer guard g has the form $e_1 \sim e_2$, where $e_1, e_2 \in IX(V^g \cup V_p, S_p)$. By definition, the evaluation of the integer guard is expressed by the evaluation of both integer expressions. We prove that

$$eval((e_1 \sim e_2), A_p, [(\vec{l}, v, \nu)]_1) = eval(\pi^s_{pq}(e_1 \sim e_2), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

Which is, by definition, equivalent to the following equality:

$$\begin{gathered} eval(e_1, A_p, [(\vec{l}, v, \nu)]_1) \sim eval(e_2, A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ eval(\pi^s_{pq}(e_1), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim eval(\pi^s_{pq}(e_2), A_q, [\pi(\vec{l}, v, \nu)]_1) \end{gathered}$$

This can easily be proved for a large set of integer expressions using lemma 4.8. However, the following, remaining, cases are not yet covered by using the lemma:

- $g \equiv a = \alpha$ or $g \equiv a \neq \alpha$, where $a \in used_\alpha$ and $\alpha \in S_p$. If $a \in V^g$, then we can rewrite the equality above to:

  $$v(a) \sim \rho_p(\alpha) = eval(a, [\pi(\vec{l}, v, \nu)]_1) \sim \rho_q(\alpha)$$

  We now by definition of the state swap that $\rho_p(\alpha) = i$ and $\rho_q(\alpha) = j$. Now we can distinguish three cases. First, $v(a) = i$. Since $a \in used_\alpha$, the data swap of π is such that $[\pi(\vec{l}, v, \nu)]_1(a) = j$. This proves the first case. The second case, $v(a) = j$, can be proved with a similar argument. Third, $v(a) \neq i, j$. Now, the data swap does not alter the value of a and we can derive: $v(a) \sim i = [\pi(\vec{l}, v, \nu)]_1(a) \sim j$ for $\sim \in \{=, \neq\}$, since both variable interpretations assign a value not equal to i or j to a.

  Now the case remains where $a \in V_p$. We must prove:

  $$v(a) \sim \rho_p(\alpha) = eval(equiv_{pq}(a), [\pi(\vec{l}, v, \nu)]_1) \sim \rho_q(\alpha)$$

  Again, we know by definition 4.4 that $\rho_p(\alpha) = i$ and $\rho_q(\alpha) = j$. We distinguish three cases. First, $v(a) = i$. Then, $[\pi(\vec{l}, v, \nu)]_1(equiv_{pq}(a)) = j$, since π swaps the values of a and $equiv_{pq}(a)$, and it applies a data swap. Second, if $v(a) = j$, then we can conclude that $[\pi(\vec{l}, v, \nu)]_1(equiv_{pq}(a)) = i$ by a similar argument. Third, if $v(a) \neq i, j$, then $[\pi(\vec{l}, v, \nu)]_1(equiv_{pq}(a)) \neq i, j$, since the process swap interchanges the values of a and $equiv_{pq}(a)$ and the data swap leaves these values unchanged.

- $g \equiv a = z$ or $g \equiv a \neq z$, where $a \in used_\alpha$ and $z \in \mathbb{Z} \setminus \{0, 1, ..., s(\alpha) - 1\}$. If $a \in V^g$, then we can rewrite the equality above to:

  $$v(a) \sim z = eval(a, A_q, [\pi(\vec{l}, v, \nu)]_1) \sim z$$

If $v(a) = i$ or $v(a) = j$, then the data swap part of the state swap changes the value of a. Since z is either larger than both i and j, or smaller than both i and j, the equality holds. If $v(a) \neq i, j$, then the value of a is left unchanged by the state swap. Therefore, the equality holds.

If $a \in V_p$, then we can rewrite the equality above to:

$$v(a) \sim z = \mathit{eval}(\mathit{equiv}_{pq}(a), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim z$$

If $v(a) = i$ or $v(a) = j$, then the process swap and data swap of the state swap act in such a way that the value of $\mathit{equiv}_{pq}(a)$ becomes j or i. Since z is either larger than both i and j, or smaller than both i and j, the equality holds. If $v(a) \neq i, j$, then the state swap acts in such a way that the value of $\mathit{equiv}_{pq}(a)$ becomes the value of a. Thus, the equality holds.

As for part (2) of the lemma, we only say that it can easily be proved by arguments very similar as above. The key observation is that the value of variable a is only possibly changed by the data swap, and *not* by the multiple process swap. □

Lemma 4.13 *Consider a (β, n)-malformed integer guard $g \in IG(V^g \cup V_p, S_p)$, a state $(\vec{l}, v, \nu)$, a syntactical assignment swap π^s_{pq} and a state swap π. We can find a $n' \in \{0, ..., s(\beta) - 1\}$ such that for g', which results from replacing malformation n by n' in g, the following holds:*

(1) If π swaps A_p with A_q, $\mathit{eval}(g', A_p, [(\vec{l}, v, \nu)]_1) = \mathit{eval}((\pi^s_{pq}(g'), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then $\mathit{eval}(g', A_p, [(\vec{l}, v, \nu)]_1) = \mathit{eval}((g', A_p, [\pi(\vec{l}, v, \nu)]_1))$.

Proof. A (β, n)-malformed integer guard can appear in two different shapes. First, it can appear as $a[i_0]...[n]...[i_m] \sim exp$, where the left side is malformed and the right side is well-formed. Second, it can appear as $a[i_0]...[n]...[i_m] \sim b[j_0]...[n]...[j_r]$, where both sides are malformed.

Let us consider the first case. With an argument as appears in the previous proof we can show that we must prove that we can find a n' such that

$$\begin{gathered}\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) \sim \mathit{eval}(exp, A_p, v)\\ =\\ \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)\end{gathered}$$

By restrictions 2 and 3 on page 13 we know that exp is not equal to α and that it is not in used_α. Therefore, we can use lemma 4.8 on page 22 to conclude that $\mathit{eval}(exp, A_p, v) = \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Moreover, by lemma 4.9 on page 25 we can find a n' such that

$$\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) = \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

This proves part (1) of the lemma for the first appearance of the malformed guard.

Let us consider the second case. Now we must prove that we can find a n' such that

$$\begin{gathered}\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) \sim \mathit{eval}(b[j_0]...[n']...[j_r], A_p, v)\\ =\\ \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim \mathit{eval}(\pi^s_{pq}(b[j_0]...[n']...[j_r]), A_q, [\pi(\vec{l}, v, \nu)]_1)\end{gathered}$$

Since both expressions are (β, n)-malformed, we can find one n' that proves this equivalence by lemma 4.9 on page 25. This proves part (1) of the lemma.

Part (2) of the lemma.is very similar to the proof above, and we do not explicitly explain it. □

Definition 4.7 (Syntactical clock assignment swap) *A syntactical swap of a clock assignment is a function $\pi^s_{pq} : CA(X^g \cup X_p) \to CA(X^g \cup X_q)$ defined as:*

$$\pi^s_{pq}(x := n) = \begin{cases} equiv_{pq}(x) := n & \text{if } x \in X_p \\ x := n & \text{otherwise} \end{cases}$$

Lemma 4.14 *Consider a clock reset $ca \in CA(X^g \cup X_p)$, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $\nu' = exec((ca, \nu))$, a clock reset swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then $[\pi(\vec{l'}, v', \nu')]_2 = exec((\pi^s_{pq}(ca), [\pi(\vec{l}, v, \nu)]_2))$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_2 = exec((ca, [\pi(\vec{l}, v, \nu)]_2))$.

Proof. According to definition 3.3, the clock reset is of the form $x := n$. We distinguish two cases for the proof of part (1) of the lemma:

- $x \in X^g$. We know by definition that $\nu'(y) = n$ if $y = x$ and $\nu(y)$ otherwise. Applying the state swap gives us that $[\pi(\vec{l'}, v', \nu')]_2(y) = n$ if $y = x$ and $[\pi(\vec{l}, v, \nu)]_2(y)$ otherwise, since $x \in X^g$. We now evaluate the right hand side of the equality. By definition: $exec(x := n, [\pi(\vec{l}, v, \nu)]_2)(y) = n$ if $y = x$, and $[\pi(\vec{l}, v, \nu)]_2(y)$ otherwise. We see that the definitions of both sides of the equality match.

- $x \in X_p$. First, we know by definition that $\nu'(y) = n$ if $y = x$ and $\nu(y)$ otherwise. Applying the state swap has as effect that the values of the local clocks of A_p and A_q are swapped. Therefore, we can expand the left hand side of the equality to:

$$[\pi(\vec{l'}, v', \nu')]_2(y) = \begin{cases} n & \text{if } y = equiv_{pq}(x), \text{ since } x \in X_p \\ [\pi(\vec{l}, v, \nu)]_2(y) & \text{otherwise} \end{cases}$$

  It is straightforward to expand the right hand side of the equality to the same definition.

As for part (2) of the lemma we note that the state swap π can only change the values of local clocks by the process swaps. Since π does not swap A_p, the values of the local clocks of A_p remain unchanged. Therefore, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(x)$ for all $x \in X^g \cup X_p$, which proves part (2). □

Definition 4.8 (Syntactical clock guard swap) *A swap of a clock guard is a function $\pi^s_{pq} : CG(X^g \cup X_p) \to CG(X^g \cup X_q)$ defined as:*

$$\pi^s_{pq}(cg) = \begin{cases} \pi_{pq}(x) \sim n & \text{if } cg \equiv x \sim n \\ \pi_{pq}(x) \sim \pi_{pq}(y) & \text{if } cg \equiv x \sim y \\ \pi_{pq}(x) \sim \pi_{pq}(y) + n & \text{if } cg \equiv x \sim y + n \end{cases}$$

where $x, y \in X^g \cup X_q$ and $n \in \mathbb{N}$, and the syntactical clock swap $\pi_{pq} : CG(X^g \cup X_p) \to CG(X^g \cup X_q)$ is defined as: $\pi_{pq}(x) = equiv_{pq}(x)$ if $x \in X_p$ and $\pi_{pq}(x) = x$ otherwise.

Lemma 4.15 *Consider a clock guard $cg \in CG(X^g \cup X_p)$, a state $(\vec{l}, v, \nu)$, a syntactical clock guard swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then $eval(cg, [(\vec{l}, v, \nu)]_2) = eval((\pi^s_{pq}(cg), [\pi(\vec{l}, v, \nu)]_2))$.

(2) If π does not swap A_p, then $eval(cg, [(\vec{l}, v, \nu)]_2) = eval(cg, [\pi(\vec{l}, v, \nu)]_2))$.

Proof. We distinguish three cases for the proof of part (1) of the lemma:

- $cg \equiv x \sim n$. We prove that $\nu(x) \sim n$ equals $[\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim n$. We distinguish two cases. First, $x \in X^g$. Then, by definition the state swap π does not alter the value of x. Moreover, the syntactical clock swap does not change x. Thus, the right hand side can be written as $\nu(x) \sim n$. Second, $x \in X_p$. Then the state swap interchanges the values of the local clocks of process A_p and A_q. Thus, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(equiv_{pq}(x))$ and similarly, $[\pi(\vec{l}, v, \nu)]_2(equiv_{pq}(x)) = \nu(x)$. Using this, we rewrite the right hand side of the equality as follows:

$$\begin{aligned} [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim n &= [\pi(\vec{l}, v, \nu)]_2(equiv_{pq}(x)) \sim n \\ &= \nu(x) \sim n \end{aligned}$$

- $cg \equiv x \sim y$. To prove the equality, we must prove that $\nu(x) \sim \nu(y)$ equals

$$[\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$$

  In the previous item we have shown that $\nu(x) = [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$, if $x \in X^g \cup X_p$. This proves the equality.

- $cg \equiv x \sim y + n$. The proof of this case, again, is straightforward, since $\nu(x) = [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$, if $x \in X^g \cup X_p$

As for part (2) of the lemma we note that the state swap π can only change the values of local clocks by the process swaps. Since π does not swap A_p, the values of the local clocks of A_p remain unchanged. Therefore, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(x)$ for all $x \in X^g \cup X_p$, which proves part (2). □

Definition 4.9 (Syntactical synchronization swap) *A swap of a synchronization is a function $\pi^s_{pq} : Sync(\Sigma, V^g \cup V_p, S_p) \to Sync(\Sigma, V^g \cup V_q, S_q)$ defined as:*

$$\pi^s_{pq}(s) = \begin{cases} \tau & \text{if } s \equiv \tau \\ \sigma & \text{if } s \equiv \sigma \in \Sigma \\ \sigma[\pi^s_{pq}(i_0)]...[\pi^s_{pq}(i_m)] & \text{if } s \equiv \sigma[i_0]...[i_m] \end{cases}$$

where $\pi^s_{pq}(i_k)$ is a syntactical integer expression swap of definition 4.5 on page 21 which acts on the simple integer expression i_k.

Lemma 4.16 *Assume that π is a state swap which swaps process A_p with process A_q. If $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$, then*

$$(src, \pi^s_{pq}(\sigma), (\pi^s_{pq}(ig), \pi^s_{pq}(cg)), (\pi^s_{pq}(ia), \pi^s_{pq}(ca)), dst) \in E_q$$

Proof. The syntactical swaps, denoted by the overloaded function π^s_{pq}, convert local clocks, local variables and local constants of process A_p to equivalent local clocks, local variables and local constants of process A_q. These equivalence functions come forth from the template instantiation mechanism. One can see that this mechanism is such that the syntactical equivalence between the edges of the lemma holds. □

In the next section we prove that the state swaps of definition 4.4 on page 21 are automorphisms.

4.3 Proving soundness

In this section we prove that the state swaps defined in definition 4.4 are automorphisms as defined in definition 2.1 on page 4. We split the proof in five small lemmas. (We abbreviate π^α_{ij} to π, ξ^α_{ij} to ξ, and ζ^α_{ij} to ζ while maintaining the parameters α, i and j.)

Lemma 4.17 *If π is a state swap, then s_0 is the initial state iff $\pi(s_0)$ is the initial state.*

Proof. We must prove that $\pi(s_0) = s_0$. First, we note that the initial locations of instances of the same template are the same. Since the process swaps only swap instances of the same template and the data swaps leave the location vector untouched, we can conclude that the location vector remains the same.

Now let us consider the variable interpretation. First, we note that array entries are initialized to zero. Therefore, swapping equivalent arrays by a process swap, or swapping dimensions by a data swap does not have any effect. This leaves us the regular variables and we distinguish two cases.

- The regular variable $\in$ *used*$_\alpha$. If the variable is global, then only the data swap can change its value. However, it does not do that, since in restriction (2) (page 14) we required that the variable is initialized to a value not in $\{0, ..., s(\alpha) - 1\}$. If the variable is local to a template, then every instance of the template initializes the variable to the same value. This is due to the initialization requirement mentioned above.

- The regular variable $\notin$ *used*$_\alpha$. In this case, the value might only be changed due to a process swap if the variable is local. This does not change the variable interpretation, since the equivalent variable of the other template instance can only be initialized to the same value (otherwise, the variable would be $\in$ *used*$_\alpha$).

Therefore, state swap does not change the variable interpretation of the initial state.

Finally, consider the clock interpretation. Since all clocks are set to zero in the initial state, the clock interpretation does not change by state swap of clock values. □

Lemma 4.18 *(s, s') is a simple action transition iff $(\pi(s), \pi(s'))$ is a simple action transition.*

Proof. We separately proof both sides of the equivalence, and we start with the implication to the right. Assume that $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ is a simple action transition as defined on page 16. We prove that $(\pi(\vec{l}, v, \nu), \pi(\vec{l'}, v', \nu'))$ is a simple action transition too. We split the proof in two parts.

- The transition is due to a well-formed edge $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$. We claim that if process A_p is not swapped by π, then this edge is still enabled. Otherwise, the edge

$$(src, \pi^s_{pq}(\sigma), (\pi^s_{pq}(ig), \pi^s_{pq}(cg)), (\pi^s_{pq}(ia), \pi^s_{pq}(ca)), dst)$$

  which is an edge of A_q by lemma 4.16 on page 33, is enabled. We prove all items of the definition:

  - From our main assumption we know that $l_p = src$ and $l'_p = dst$. If A_p is not swapped, the the location of A_p is not changed. Thus, obviously $[\pi(\vec{l}, v, \nu)]_{1_p} = src$ and $[\pi(\vec{l'}, v', \nu')]_{1_p} = dst$. If A_p is swapped with A_q, then the active locations of these processes are interchanged. Thus, $[\pi(\vec{l}, v, \nu)]_{1_q} = l_p = src$ and $[\pi(\vec{l'}, v', \nu')]_{1_q} = l'_p = dst$.
  - Our main assumption says that $eval(ig, A_p, v) = true$ and $eval(cg, \nu) = true$. Assume that A_p is not swapped. By part (2) of lemma 4.12 on page 30 we know that $eval(ig, A_p, [\pi(\vec{l}, v, \nu)]_1) = eval(ig, A_p, v) = true$. Similarly, by part (2) of lemma 4.15 on page 33 we know that $eval(cg, [\pi(\vec{l}, v, \nu)]_2) = eval(cg, \nu) = true$.
    Now let us assume that A_p is swapped with A_q. We must prove that both $\pi^s_{pq}(ig)$ and $\pi^s_{pq}(cg)$ are true. By part (1) of lemma 4.12 we know that $eval(\pi^s_{pq}(ig), A_q, [\pi(\vec{l}, v, \nu)]_1)$ equals $eval(ig, A_p, v) = true$. Similarly, by part (1) of lemma 4.15 on page 33 we know that $eval(\pi^s_{pq}(cg), [\pi(\vec{l}, v, \nu)]_2) = eval(cg, \nu) = true$.
    Concluding, in both cases the guards are satisfied.
  - Our main assumption says that $v' = exec(ia, A_p, v)$ and $\nu' = exec(ca, \nu)$. Assume that A_p is not swapped. By part (2) of lemma 4.10 on page 26 we know that $[\pi(\vec{l'}, v', \nu')]_1 = exec(ia, A_p, [\pi(\vec{l}, v, \nu)]_1)$. Similarly, by part (2) of lemma 4.14 on page 32 we know that $[\pi(\vec{l'}, v', \nu')]_2 = exec(ca, [\pi(\vec{l}, v, \nu)]_2)$.
    Assume that A_p is swapped with A_q. By part (1) of lemma 4.10 on page 26 we know that $[\pi(\vec{l'}, v', \nu')]_1 = exec(\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Similarly, by part (1) of lemma 4.14 we know that $[\pi(\vec{l'}, v', \nu')]_2 = exec(\pi^s_{pq}(ca), [\pi(\vec{l}, v, \nu)]_2)$.
    Concluding, in both cases the interpretations match the assignments.
  - We know by our assumption that $eval(I_i(l_i), \nu) = true$ for all processes i. Now consider an invariant $I_p(l_p)$. If π does not swap A_p, then we know by lemma 4.15 on page 33 that $eval(I_p(l_p), [\pi(\vec{l}, v, \nu)]_2) = eval(I_p(l_p), [(\vec{l}, v, \nu)]_2) = true$. Similarly, $eval(I_p(l_p), [\pi(\vec{l}, v, \nu')]_2) = true$.
    If π does swap A_p with A_q, then we know by lemma 4.15 that

$$eval(\pi^s_{pq}(I_p(l_p)), [\pi(\vec{l}, v, \nu)]_2) = eval(I_p(l_p), [(\vec{l}, v, \nu)]_2) = true$$

    Similarly,

$$eval(\pi^s_{qp}(I_q(l_q)), [\pi(\vec{l}, v, \nu)]_2) = eval(I_q(l_q), [(\vec{l}, v, \nu)]_2) = true$$

    Observe that the syntactical swap converts the invariants: $\pi^s_{pq}(I_p(l_p)) = I_q(l_q)$, and vice versa. Thus, we may conclude that $eval(I_p(l_p), [\pi(\vec{l}, v, \nu)]_2) = true$ and the same for $I_q(l_q)$. We can repeat the same argument for $(\vec{l'}, v', \nu')$. Concluding, all invariants are still satisfied after the state swap.

– Observe that the state swap *permutes* the location vector. Therefore, the count of committed locations does not change. Moreover, if A_p is swapped with A_q, then the active locations are interchanged. Thus, l_p is active iff $[\pi(\vec{l}, v, \nu)]_{1_q}$ is active. (The processes originate from the same template, thus the same locations are committed.)

- The transition is due to an (β, n)-malformed edge $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$. We claim that if A_p is not swapped by π, then we can use this edge, or we can find another edge in E_p which proves that $(\pi(s), \pi(s'))$ is a simple action transition. The proof follows the same structure as in the previous item, except we use the lemma's 4.11 on page 29 and 4.13 on page 31 in conjunction with restriction (3) on page 14. On the other hand, if A_p is swapped with A_q by π, then we can find an edge in E_q which proves or claim. Again, we need the lemma's 4.11 and 4.13 in conjunction with restriction (3).

The implication to the left can easily be proved. Assume that $(\pi(s), \pi(s'))$ is a simple action transition. Above we have proved that if (s, s') is a simple action transition, then $(\pi(s), \pi(s'))$ is a simple action transition. Thus, we can say that $(\pi \circ \pi(s), \pi \circ \pi(s'))$ is a simple action transition. In lemma 4.7 we have proved that $\pi \circ \pi(s) = s$. Therefore, (s, s') is a simple action transition. □

Lemma 4.19 *(s, s') is a σ action transition iff $(\pi(s), \pi(s'))$ is a σ action transition.*

Proof. The proof of this lemma is very similar to the prove given for lemma 4.18. Therefore, we do not explicitly explain it. □

Lemma 4.20 *(s, s') is a δ delay transition iff $(\pi(s), \pi(s'))$ is a δ delay transition.*

Proof. We defined a δ delay transition on page 16. As in the previous proofs, we first prove the implication to the right. The implication to the left follows without effort.

Assume that $((\vec{l}, v, \nu), (\vec{l}, v, \nu'))$ is a δ delay transition. We consider every item of the definition separately:

- We know by our assumption that $\nu'(x) = \nu(x) + \delta$ for all clocks x. Now consider a clock y and assume that it is not swapped. Thus: $[\pi(\vec{l}, v, \nu)]_2(y) = \nu(y)$, and $[\pi(\vec{l}, v, \nu')]_2(y) = \nu'(y)$. Since we assumed that $\nu'(x) = \nu(x) + \delta$ for all clocks x, we can conclude that $[\pi(\vec{l}, v, \nu')]_2(y) = [\pi(\vec{l}, v, \nu)]_2(y) + \delta$.

  If y is swapped, then $[\pi(\vec{l}, v, \nu)]_2(y) = \nu(z)$ and $[\pi(\vec{l}, v, \nu')]_2(y) = \nu'(z)$ for some equivalent clock z. Since we assumed that $\nu'(x) = \nu(x) + \delta$ for all clocks x, we know that $[\pi(\vec{l}, v, \nu')]_2(y) = [\pi(\vec{l}, v, \nu)]_2(y) + \delta$.

- See the fourth item of the first bullet in the proof of lemma 4.18.

- The state swap only *permutes* the active locations. Therefore, no committed locations become active by swapping.

- By our main assumption we know that if there exists an i such that $lt_i(l_i) = urgent$, then no state r exists such that (s, r) is a simple or σ action transition.

We must prove that if there exists an i such that $lt_i(l_i) = urgent$, then no state r' exists such that $(\pi(s), r')$ is a simple or σ action transition. It suffices to prove the right hand side of the implication. Therefore, assume that this state r' does exist. From lemma 4.18 and lemma 4.19 we know that $(\pi \circ \pi(s), \pi(r'))$ also is a simple or σ action transition. By lemma 4.7 on page 21 we can conclude that $(s, \pi(r'))$ is a simple or σ action transition. Thus, the state r mentioned above does exist, namely $t = \pi(r')$. From this contradiction we can conclude that the state r' does not exist.

- The argument is similar to the one in the previous item.

□

The following corollary is the main result of this paper.

Corollary 4.1 (Soundness) *Every state swap is an automorphism.*

Proof. By combination of lemmas 4.7, 4.17, 4.18, 4.19 and 4.20. □

In the next section we explain how we can use these state swaps for reduction of the searchable state space of SUPPAAL models.

5 Using the scalarsets for symmetry reduction

In the previous section we have identified a set of automorphisms on the state graph of a SUPPAAL model (see definition 4.4 on page 21). We can use these automorphisms to construct a normal form operator θ, which can be used by the forward exploration algorithm, see section 2. In the next subsection we prove that computation of a canonical representative is practically infeasible.

5.1 The complexity of sorting matrices

In this section we consider difference bounded matrices which are used by UPPAAL to represent sets of clock interpretations [6, 11]. Therefore, we let B denote the bounds domain $\mathbb{Z} \times \{0, 1\} \cup \infty$ and we let $\mathcal{D}_k$ denote the set of all $k \times k$ matrices over B. For any $D \in \mathcal{D}_k$ we let D_{ij} denote the bound in the i-th row and the j-th column of the matrix. We can easily represent these square matrices by graphs.

Definition 5.1 (Graph representation) *Given a matrix $D \in \mathcal{D}_k$. We represent this matrix by an edge-labeled directed graph $\mathcal{G}(D) = (V, E)$, where*

- $V = \{ x_i \mid 1 \leq i \leq k \}$
- $E = \{ (x_i, D_{ij}, x_j) \mid 1 \leq i, j \leq k \}$

Note that we can reconstruct the matrix D from its graph representation $\mathcal{G}(D)$. Throughout the rest of this section we will use the graph representation of the difference bounded matrices.

The set of graph representations of matrices in $\mathcal{D}_k$ is denoted by $\mathcal{G}_k$. These graphs are so-called *strongly regular*. To compare graphs with each other, we use the well-known notion of *isomorphism* on graphs [19]. We give the definition of graph isomorphism in terms of a decision problem.

Definition 5.2 (Graph isomorphism) *Imagine a pair of edge-labeled and directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $|V_1| = |V_2|$. The graph isomorphism problem asks if there exists a bijection $f : V_1 \to V_2$ such that $(v, l, v') \in E_1$ if and only if $(f(v), l, f(v')) \in E_2$.*

If a graph G is isomorphic with a graph G' (that is, the bijection f exists), then we denote this by $G \simeq G'$. If we assume standard representations of graphs, i.e., as a linked adjacency list or an adjacency matrix, then there are no polynomial time algorithms known for this general problem. However, the problem has not been proved to be $\mathcal{NP}$-complete [16]. There are polynomial time algorithms known for certain subsets of graphs, but these subsets do not include the strongly regular graphs [20].

Let H be some set of bijections from $\mathcal{D}_k$ to itself. We let $\mathcal{C}(H)$ denote the closure of $H \cup \{\mathbf{id}\}$ under inverse and composition. This closure of a set of bijections can be complete when we consider the graph domain.

Definition 5.3 (Vertex completeness) *A set of bijections $H : \mathcal{G}_k \to \mathcal{G}_k$ is vertex complete if and only if for all graphs $G \in \mathcal{G}_k$:*

$$\{ h(G) \mid h \in \mathcal{C}(H) \} = \{ G' \in \mathcal{G}_k \mid G' \simeq G \}$$

This somewhat tricky definition means that a set of bijections H is vertex complete if and only if for all graphs G we can use the elements of $\mathcal{C}(H)$ to obtain exactly all graphs which are isomorphic to G.

We now define the closure of some matrix under certain sets of bijections. This closure contains all matrices that can be derived from the original one using the bijections, their inverses, and their compositions.

Definition 5.4 (H-closure) *Consider a graph $G \in \mathcal{G}_k$ and let H be a set of bijections from $\mathcal{G}_k$ to itself. The H-closure of G, denoted by close(G, H), is defined as*

$$close(G, H) = \{ G' \mid \exists_{h \in \mathcal{C}(H)}\ h(G) = G' \}$$

Let us assume a total ordering $\preceq_k$ on the set $\mathcal{G}_k$. Then every finite subset of $\mathcal{G}_k$ contains exactly one minimal element with respect to the ordering. To find a canonical representation of a set of (graph representations of) matrices, we would very much like to find that $\preceq_k$-minimal element. This gives us the following computational problem.

Definition 5.5 (H-minimalization) *Given $G \in \mathcal{G}_k$ and a set H of bijections $\mathcal{G}_k \to \mathcal{G}_k$ such that close(G, H) is finite. Let $\preceq_k$ be a total ordering on $\mathcal{G}_k$. The H-minimalization problem asks to compute the $\preceq_k$-minimal element of close(G, H).*

In order to argue about the complexity of this minimalization problem, we relate it to a decision problem.

Definition 5.6 (H-decision) *Given $G, G' \in \mathcal{G}_k$ and a set H of bijections $\mathcal{G}_k \to \mathcal{G}_k$. The H-decision problem asks if there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$.*

Lemma 5.1 *If we can solve the H-minimalization problem in polynomial time, then we can solve the H-decision problem in polynomial time.*

Proof. We assume the existence of a polynomial time algorithm for the minimalization problem. If we want to solve the decision problem for G and G', then we apply this algorithm to both graphs. This gives us – in polynomial time – the minimal elements G_{min} and G'_{min}. We claim that $G_{min} = G'_{min}$ if and only if there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$.

The $=$ relation on graphs in $\mathcal{D}_k$ can be computed, if we use the adjacency matrix representation, with k^2 comparisons between the elements of the matrices. So, if we assume that these comparisons can be done in constant time, then the overall complexity of the decision problem is polynomial. Next, we prove both sides of the double implication of our claim.

- $\Rightarrow$ Assume that $G_{min} = G'_{min}$. By our definition of the H-closure, we know that there exists a $h_1 \in \mathcal{C}(H)$ such that $h_1(G) = G_{min}$, and that there exists a $h_2 \in \mathcal{C}(H)$ such that $h_2(G') = G'_{min}$. By the definition of $\mathcal{C}(H)$ we know that $h_2^{-1}(G'_{min}) = G'$. Since $G_{min} = G'_{min}$, we can conclude that $h_2^{-1}(G_{min}) = G'$. Therefore, $(h_2^{-1} \circ h_1)(G) = G'$. This composition clearly is an element of $\mathcal{C}(H)$.

- $\Leftarrow$ Let us assume that there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$. Consider the function g that maps G' to G'_{min}. By definition $g \in \mathcal{C}(H)$. Combination of g with h gives that $(g \circ h)(G) = G'_{min}$. This composition naturally is an element of $\mathcal{C}(H)$ and we conclude that $G'_{min} \in close(G, H)$. Of course, $G_{min} \in close(G, H)$. Thus, both G_{min} and G'_{min} are minimal elements of $close(G, H)$. Since our ordering $\preceq_k$ is total, either $G_{min} \preceq_k G'_{min}$ or $G'_{min} \preceq_k G_{min}$. In both cases we can easily conclude that $G_{min} = G'_{min}$. (Remember that an element $a \in A$ is minimal if and only if $b \in A, b \preceq a \Rightarrow b = a$.)

□

A consequence of the previous lemma is that if we cannot solve the H-decision problem in polynomial time, then we cannot solve the H-minimalization problem in polynomial time. We show that we indeed cannot solve one important H-decision problem in polynomial time by linking it to a difficult problem on graphs, namely the graph isomorphism problem of definition 5.2.

Lemma 5.2 *If H is vertex complete, then we can solve the graph isomorphism problem for strongly regular graphs in polynomial time only if we can solve the H-decision problem in polynomial time.*

Proof. Let us assume that H is vertex complete and that we know an algorithm that solves the H-decision problem in polynomial time. If we want to know whether $G \simeq G'$, then we just run the algorithm on G and G'. We claim that there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$ if and only if $G \simeq G'$ for all $G, G' \in \mathcal{G}_k$. We prove the two directions of this implication.

- $\Rightarrow$ Assume that there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$. Since H is vertex complete, we know by definition 5.3 that $h(G) \in \{ G'' \in \mathcal{G}_k \mid G'' \simeq G \}$ and thus (since $h(G) = G'$) $G' \simeq G$.

- $\Leftarrow$ Assume that $G \simeq G'$. Since H is vertex complete, we know by definition 5.3 that $G' \in \{ h(G) \mid h \in \mathcal{C}(H) \}$. We must conclude that there exists a $h \in \mathcal{C}(H)$ such that $h(G) = G'$.

□

Corollary 5.1 *If we cannot solve the graph isomorphism problem for strongly regular graphs in polynomial time, then we cannot solve the H-minimalization problem in polynomial time when H is vertex complete.*

Proof. By combination of lemma 5.1 and lemma 5.2. □

We already mentioned that there are no known polynomial algorithms for the graph isomorphism problem for strongly regular graphs. Therefore, we must conclude that solving the H-minimalization problem for a set H of vertex complete bijections is a very time intensive job.

The state swaps projected to the clock interpretation are bijections from the set of DBMs to itself. (Note that swapping clocks x_i and x_j comes down to swapping vertex labels in the graph representation of the DBM. This is equivalent to interchanging the columns i and j and then the rows i and j in the DBM.) The set of all state swaps is *almost* vertex complete, since they permute only a subset of the clocks. In order to talk about the complexity of the minimalization problem in this setting, we need a slightly less general version of the graph isomorphism problem.

The partial graph isomorphism problem asks for two graphs whether or not fixed subgraphs (namely the subgraphs which represent the swappable clocks) are isomorphic. This problem is exactly as difficult as the regular graph isomorphism problem for strongly regular graphs with respect to polynomial or exponential complexity.

With the partial graph isomorphism problem, we can redefine the notion of vertex completeness. Then, the “clock part” of the set of all state swaps does satisfy the condition to be vertex complete. Lemma 5.2 and corollary 5.1 are still valid for this “partial” setting. Therefore, computing a canonical representative for full symmetry in a setting of DBM technology is a hard job!

5.2 A sub optimal approach

Instead of computing a canonical representative of a symmetry class, it is practically more useful to use a fast, but non-canonical, normal form operator. This increases the memory usage in comparison with a canonical normal form operator, but is, very probably, much faster. There are many possibilities for a non-canonical normal form operator (for example, see [7]). These are all based on minimizing the state using the automorphisms. The challenge is to find a computationally efficient normal form operator, which improves on the non-symmetric tool in most situations. Since we find reasoning about the gain of normal form operators for different models very difficult, we like to address this with experimental research which can be conducted after the core implementation.

As a final note we mention that the so-called state properties, which are used to define sets of states, should also be taken into account by our normal form operator. However, we do not expect this to be a difficult problem.

6 Summary and future work

We have proposed an enhancement of the model checker Uppaal, which exploits structural symmetries to reduce the searchable state space. In section 2 we summarized work of Ip and Dill [14] which we used for our symmetry reduction technique. In section 3 we have proposed a syntactical adjustment of the system description language of Uppaal. More precisely, we have added the well-known scalarset data type, and multi-dimensional arrays of integer variables and channels. In section 4 we used these scalarsets to extract automorphisms on the state graph of our models. The main result of this paper is corollary 4.1 on page 37, which states that our technique is sound. Finally, in section 5 we sketch some possibilities to use the automorphisms to reduce the searchable state space. A second result of this paper is corollary 5.1 on page 40, which states that the so-called orbit problem in a setting with DBM technology is at least as difficult as the graph isomorphism problem for strongly regular graphs. This renders the computation of a canonical representative of a symmetry class practically infeasible.

Future work includes the implementation of the proposed technique. As experiences of Ip and Dill already showed, the actual implementation of the computation of the representative of a symmetry class is very important [15]. Canonical representatives minimize the space usage, but can be very costly in time. The feasibility of non-canonical representatives, however, might vary per model. Therefore, experiments with different algorithms to compute non-canonical representatives and a fairly large set of different models are necessary to asses the effectiveness of scalarsets in a dense-time setting.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[2] R. Alur. Timed automata. In *11th International Conference on Computer Aided Verification*, number 1633 in LNCS, pages 8–22. Springer–Verlag, 1999.

[3] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[4] R. Alur and D.L. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages, and Programming*, pages 322–335, 1990.

[5] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.

[6] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[7] D. Bosnacki, D. Dams, and L. Holenderski. A heuristic for symmetry reductions with scalarsets. In *International Symposium on FME 2001: Formal Methods for Increasing Software Productivity, LNCS*, volume 1, 2001.

[8] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[9] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, 2000.

[10] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000. Available through URL `http://www.cis.ksu.edu/santos/bandera/`.

[11] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 197–212. Springer–Verlag, 1989.

[12] D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[13] G. J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[14] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands.

[15] C.N. Ip and D.L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, Cambridge, MA, 1993.

[16] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhäuser, 1993.

[17] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, pages 134–152, 1998.

[18] K. L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, Pittsburgh, May 1992.

[19] R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

[20] D.A. Spielman. Faster isomorphism testing of strongly regular graphs. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 576–584, May 1996.

[21] S. Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(2), 1997.