Adding Symmetry Reduction to Uppaal*

Martijn Hendriks[1], Gerd Behrmann[2], Kim Larsen[2],
Peter Niebert[3]**, and Frits Vaandrager[1]

[1] Nijmeegs Instituut voor Informatica en Informatiekunde
University of Nijmegen
The Netherlands
{martijnh,fvaan}@cs.kun.nl
[2] Department of Computing Science
Aalborg University
Denmark
{behrmann,kgl}@cs.auc.dk
[3] Laboratoire d'Informatique Fondementale, CMI
Université de Provence
France
peter.niebert@lif.univ-mrs.fr

Abstract. We describe a prototype extension of the real-time model checking tool Uppaal with symmetry reduction. The symmetric data type *scalarset*, which is also used in the Murφ model checker, was added to Uppaal's system description language to support the easy static detection of symmetries. Our prototype tool uses *state swaps*, described and proven sound earlier by Hendriks, to reduce the space and memory consumption of Uppaal. Moreover, the reduction strategy is *canonical*, which means that the symmetries are optimally used. For all examples that we experimented with (both academic toy examples and industrial cases), we obtained a drastic reduction of both computation time and memory usage, exponential in the size of the scalar sets used.

1 Introduction

Model checking is a semi-automated technique for the validation and verification of all kinds of systems [8]. The approach requires the construction of a *model* of the system and the definition of a *specification* for the system. A model checking tool then computes whether the model satisfies its specification. Nowadays, model checkers are available for many application areas, e.g., hardware systems [10, 22], finite-state distributed systems [17], and timed and hybrid systems [21, 27, 25, 16].

* Supported by the European Community Project IST-2001-35304 (AMETIST), http://ametist.cs.utwente.nl.

** Peter Niebert suggested the method for efficient computation of canonical representatives at an AMETIST project meeting, and was therefore invited to join the list of authors after acceptance of the paper.

K.G. Larsen and P. Niebert (Eds.): FORMATS 2003, LNCS 2791, pp. 46–59, 2004.

Despite the fact that model checkers are relatively easy to use compared to manual verification techniques or theorem provers, they are not being applied on a large scale. An important reason for this is that they must cope with the *state space explosion* problem, which is the problem of the exponential growth of the state space as models become larger. This growth often renders the mechanical verification of realistic systems practically impossible: there just is not enough time or memory available. As a consequence, much research has been directed at finding techniques to fight the state space explosion. One such a technique is the exploitation of behavioral symmetries [18, 23, 20, 19, 12, 7]. The exploitation of *full* symmetries can be particularly profitable, since its gain can approach a factorial magnitude.

There are many timed systems which clearly exhibit full symmetry, e.g., Fischer's mutual exclusion protocol [1], the CSMA/CD protocol [24, 27], industrial audio/video protocols [13], and distributed algorithms, for instance [4].

Motivated by these examples, the work presented in [14] describes how Uppaal, a model checker for networks of timed automata [21, 3, 2], can be enhanced with symmetry reduction. The present paper puts this work to practice: a prototype of Uppaal with symmetry reduction has been implemented. The symmetric data type *scalarset*, which was introduced in the Murφ model checker [10], was added to Uppaal's system description language to support the easy static detection of symmetries. Furthermore, the *state swaps* described and proven sound in [14] are *optimally* used to reduce the space and time consumption of the model checking algorithm. Run-time data is reported for the examples mentioned above, showing that symmetry reduction in a timed setting can be very effective.

Related work. Symmetry reduction is a well-known technique to reduce the resource requirements for model checking algorithms, and it has been successfully implemented in model checkers such as Murφ [10, 19], SMV [22], and Spin [17, 6]. As far as we know, the only model checker for timed systems that exploits symmetry is Red [25, 26]. The symmetry reduction technique used in Red, however, gives an over approximation of the reachable state space (this is called the *anomaly of image false reachability* by the authors). Therefore, Red can only be used to ensure that a state is *not* reachable when it is run with symmetry reduction, whereas symmetry enhanced Uppaal can be used to ensure that a state is reachable, or that it is not reachable.

Contribution. We have added symmetry reduction as used within Murφ, a well-established technique to combat the state space explosion problem, to the real-time model checking tool Uppaal. For researchers familiar with model checking it will come as no surprise that this combination can be made and indeed leads to a significant gain in performance. Still, the effort required to actually add symmetry reduction to Uppaal turned out to be substantial.

The soundness of the symmetry reduction technique that we developed for Uppaal does not follow trivially from the work of Ip and Dill [19] since the description languages of Uppaal and Murφ, from which symmetries are extracted

```
$passed := \emptyset$
$waiting := Q_0$
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 $passed$
        $waiting := waiting \cup \{ q' \in Q \mid (q, q') \in \Delta \}$
    fi
od
return NO
```

Fig. 1. A general forward reachability analysis algorithm

automatically, are quite different. In fact, the proof that symmetry reduction for Uppaal is sound takes up more than 20 pages in [14].

The main theoretical contribution of our work is an efficient algorithm for the computation of a canonical representative. This is not trivial due to Uppaal's symbolic representation of sets of clock valuations.

Many timed systems exhibit symmetries that can be exploited by our methods. For all examples that we experimented with, we obtained a drastic reduction of both computation time and memory usage, exponential in the size of the scalar sets used.

Outline. Section 2 presents a very brief summary of model checking and symmetry reduction in general, while Sections 3 and 4 introduce symmetry reduction for the Uppaal model checker in particular. In Section 5, we present run-time data of Uppaal's performance with and without symmetry reduction, and Section 6 summarizes and draws conclusions.

A full version of the present paper including proofs of lemma 1 and of theorem 2 is available as [15].

2 Model Checking and Symmetry Reduction

This section briefly summarizes the theory of symmetry presented in [19], which is reused in a timed setting since (i) it has proven to be quite successful, and (ii) it is designed for reachability analysis, which is the main purpose of the Uppaal model checker. We simplify (and in fact generalize) the presentation of [19] using the concept of bisimulations.

In general, a transition system is a tuple (Q, Q_0, Δ), where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, and $\Delta \in Q \times Q$ is a transition relation between states. Figure 1 depicts a general forward reachability algorithm which, under the assumption that Q is finite, computes whether there exists a reachable state q that satisfies some given property ϕ (denoted by $q \models \phi$).

Due to the state space explosion problem, the number of states of a transition system frequently gets too big for the above algorithm to be practical. We would like to exploit structural properties of transition systems (in particular symmetries) to improve its performance. Here the well-known notion of bisimulation comes in naturally:

Definition 1 (Bisimulation). *A* bisimulation *on some transition system, say* (Q, Q_0, Δ), *is a relation* $R \subseteq Q \times Q$ *such that, for all* $(q, q') \in R$,

1. $q \in Q_0$ *if and only if* $q' \in Q_0$,
2. *if* $(q, r) \in \Delta$ *then there exists an* r' *such that* $(q', r') \in \Delta$ *and* $(r, r') \in R$,
3. *if* $(q', r') \in \Delta$ *then there exists an* r *such that* $(q, r) \in \Delta$ *and* $(r, r') \in R$.

Suppose that, before starting the reachability analysis of a transition system, we know that a certain equivalence relation $\approx$ is a bisimulation and respects the predicate ϕ in the sense that either all states in an equivalence class satisfy ϕ or none of them does. Then, when doing reachability analysis, it suffices to store and explore only a single element of each equivalence class. To implement the state space exploration, a *representative function* θ may be used that converts a state to a representative of the equivalence class of that state:

$$\forall_{q \in Q} \ (q \approx \theta(q)) \tag{1}$$

Using θ, we may improve the algorithm in Figure 1 by replacing lines 2 and 8, respectively, by:

$$(2) \ \textit{waiting} := \{\, \theta(q) \mid q \in Q_0 \,\}$$

$$(8) \ \textit{waiting} := \textit{waiting} \cup \{\, \theta(q') \mid (q, q') \in \Delta \,\}$$

It can easily be shown that the adjusted algorithm remains correct: for all (finite) transition systems the outcomes of the original and the adjusted algorithm are equal. If the representative function is "good", which means that many equivalent states are projected onto the same representative, then the number of states to explore, and consequently the size of the passed set, may decrease dramatically. However, in order to apply the approach, the following two problems need to be solved:

- A suitable bisimulation equivalence that respects ϕ needs to be statically derived from the system description.
- An appropriate representative function θ needs to be constructed that satisfies formula (1). Ideally, θ satisfies $q \approx q' \Rightarrow \theta(q) = \theta(q')$, in which case it is called *canonical*.

In this paper, we use symmetries to solve these problems. As in [19], the notion of *automorphism* is used to characterize symmetry within a transition system. This is a bijection on the set of states that (viewed as a relation) is a bisimulation. Phrased alternatively:

Definition 2 (Automorphism). *An* automorphism *on a transition system* (Q, Q_0, Δ) *is a bijection* $h : Q \to Q$ *such that*

1. *$q \in Q_0$ if and only if $h(q) \in Q_0$ for all $q \in Q$, and*
2. *$(q, q') \in \Delta$ if and only if $(h(q), h(q')) \in \Delta$ for all $q, q' \in Q$.*

Let H be a set of automorphisms, let **id** be the identity function on states, and let $G(H)$ be the closure of $H \cup \{\mathbf{id}\}$ under inverse and composition. It can be shown that $G(H)$ is a group, and it induces a bisimulation equivalence relation $\approx$ on the set of states as follows:

$$q \approx q' \iff \exists_{h \in G(H)} \, (h(q) = q') \tag{2}$$

We introduce a symmetric data type to let the user explicitly point out the symmetries in the model. Simple static checks can ensure that the symmetry that is pointed out is not broken. Our approach to the second problem of coming up with good representative functions consists of "sorting the state" w.r.t. some ordering relation on states using the automorphisms. For instance, given a state q and a set of automorphisms, find the smallest state q' that can be obtained by repeatedly applying automorphisms and their inverses to q. It is clear that such a θ satisfies the correctness formula (1), since it is constructed from the automorphisms only.

3 Adding Scalarsets to Uppaal

The tool Uppaal is a model checker for networks of timed automata extended with discrete variables (bounded integers, arrays) and blocking, binary synchronization as well as non-blocking broadcast communication (see for instance [21]). In the remainder of this section we illustrate by an example Uppaal's description language extended with a *scalarset* type constructor allowing symmetric data types to be syntactically indicated. Our extension is based on the notion of scalarset first introduced by Ip and Dill in the finite-state model checking tool Murφ [10, 19]. Also our extension is based on the C-like syntax to be introduced in the forthcoming version 4.0 of Uppaal.

To illustrate our symmetry extension of Uppaal we consider Fischer's mutual exclusion protocol. This protocol consists of n process identical up to their unique process identifiers. The purpose of the protocol is to insure mutual exclusion on the critical sections of the processes. This is accomplished by letting each process write its identifier (`pid`) in a global variable (`id`) before entering its critical section. If after some given lower time bound (say 2) `id` still contains the `pid` of the process, then it may enter its critical section.

A scalarset of size n may be considered as the subrange $\{0, 1, \ldots, n-1\}$ of the natural numbers. Thus, the n process identifiers in the protocol can be modeled using a scalarset with size n. In addition to the global variable `id`, we use the array `active` to keep track of all active locations of the processes[1]. Global declarations are the following:

[1] This array is actually redundant and not present in the standard formulations of the protocol. However, it is useful for showing important aspects of our extension.

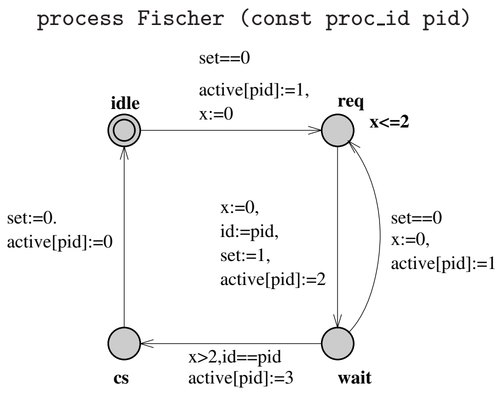

Fig. 2. The template for Fischer's protocol

```
typedef scalarset[3] proc_id;    // a scalarset type with size 3
proc_id id;                      // declaration of a proc_id
                                 //   variable
bool set;                        // declaration of a boolean
int active[proc_id];             // declaration of an array
                                 //   indexed by proc_id
```

The first line defines `proc_id` to be a scalarset type of size 3, and the second line declares `id` to be a variable over this type. Thus `scalarset` is in our extension viewed as a type constructor. In the last line we show a declaration of an array indexed by elements of the scalarset `proc_id`.

At this point the only thing missing is the declaration of the actual processes in the system. In the description language of Uppaal, processes are obtained as instances of parameterized process templates. In general, templates may contain several different parameters (e.g. bounded integers, clocks, and channels). In our extension we allow in addition the use of scalarsets as parameters. In the case of Fischer's protocol the processes of the system are given as instances of the template depicted in Figure 2. The template has one local clock, `x`, and no local variables. Note that the header of the template defines a (constant) scalarset parameter `pid` of type `proc_id`. Access to the critical section `cs` is governed by suitable updates and tests of the global scalarset variable `id` together with upper and lower bound time constraints on when to proceed from requesting access (`req`) respectively proceed from waiting for access (`wait`). Note that all transitions update the array `active` to reflect the current active location of the process. The instantiation of this template and declaration of all three process in the system can be done as follows:

```
FischerProcs = forall i in proc_id : Fischer(i);
system FischerProcs;
```

The `forall` construct iterates over all elements of a declared scalarset type. In this case the iteration is over `proc_id` and a set of instances of the template `Fischer` is constructed and bound to `FischerProcs`. In the second line the final system is defined to be precisely this set.

4 Using Scalarsets for Symmetry Reduction

As a preliminary to this section we briefly mention the state representation of Uppaal. A state is a tuple $(\boldsymbol{l}, v, Z)$, where $\boldsymbol{l}$ is the location vector, v is the integer variable valuation, and Z is a zone, which is a convex set of clock valuations that can efficiently be represented by a *difference bounded matrix* (DBM) [5, 9].

4.1 Extraction of Automorphisms

This subsection is a very brief summary of [14], to which we refer for further details. The new syntax described in the previous section enables us to derive the following information from a system description:

- A set Ω of scalarset types.
- For each $\alpha \in \Omega$: (i) a set V_α of variables of type α, and (ii) a set D_α of pairs (a, n) where a is an array and n is a dimension of a that must be indexed by variables of type α to ensure soundness. We assume that arrays that are indexed by scalarsets do not contain elements of scalarsets. The reason is that this would make computation of a canonical representative as hard as testing for graph isomorphism.
- A partial mapping $\gamma : P \times \Omega \hookrightarrow \mathbb{N}$ that gives for each process p and scalarset α the element of α with which p is instantiated. This mapping is defined by quantification over scalarsets in the process definition section.

This information enables us to derive so-called *state swaps*. Let Q be the set of states of some Uppaal model, and let α be a scalarset type in the model with size n. A state swap $swap^{\alpha}_{i,j} : Q \to Q$ can be defined for all $0 \leq i < j < n$, and consists of two parts:

- The *multiple process swap* swaps the contributions to the state of all pairs of processes p and p' if they originate from the same template and $\gamma(p, \alpha) = i$, $\gamma(p', \alpha) = j$ and $\gamma(p, \beta) = \gamma(p', \beta)$ for all $\beta \neq \alpha \in \Omega$. Swapping such a pair of symmetric processes consists of interchanging the active locations and the values of the local variables and clocks (note that this is not a problem since the processes originate from the same template).
- The *data swap* swaps array entries i and j of all dimensions that are indexed by scalarset α (these are given by the set D_α). Moreover, it swaps the value i with the value j for all variables in V_α.

Consider the instance of Fischer's mutual exclusion protocol (as described in the previous section) with three processes. There are three swap functions:

$swap^{\mathtt{proc\_id}}_{0,1}$, $swap^{\mathtt{proc\_id}}_{0,2}$ and $swap^{\mathtt{proc\_id}}_{1,2}$. Now consider the following state of the model (the active location of the i-th process is given by l_i and the local clock of this process is given by x_i):

$$\begin{array}{ll}
\boldsymbol{l} & : l_0 = \mathtt{idle},\ l_1 = \mathtt{wait},\ l_2 = \mathtt{cs} \\
v & : \mathtt{id} = 2,\ \mathtt{set} = 1 \\
Z & : x_0 = 4,\ x_1 = 3,\ x_2 = 2.5 \\
\mathtt{active} & ; \mathtt{active}[0] = 0,\ \mathtt{active}[1] = 2,\ \mathtt{active}[2] = 3
\end{array}$$

When we apply $swap^{\mathtt{proc\_id}}_{0,2}$ to this state, the result is the following state:

$$\begin{array}{ll}
\boldsymbol{l} & : l_0 = \mathtt{cs},\ l_1 = \mathtt{wait},\ l_2 = \mathtt{idle} \\
v & : \mathtt{id} = 0,\ \mathtt{set} = 1 \\
Z & : x_0 = 2.5,\ x_1 = 3,\ x_2 = 4 \\
\mathtt{active} & ; \mathtt{active}[0] = 3,\ \mathtt{active}[1] = 2,\ \mathtt{active}[2] = 0
\end{array}$$

The process swap swaps l_0 with l_2, and x_0 with x_2. The data swap first changes the value of the variable id from 2 to 0, since $\mathtt{id} \in V_{\mathtt{proc\_id}}$, and then swaps the values of active[0] and active[2]. Applying $swap^{\mathtt{proc\_id}}_{1,2}$ to this state gives the following state:

$$\begin{array}{ll}
\boldsymbol{l} & : l_0 = \mathtt{cs},\ l_1 = \mathtt{idle},\ l_2 = \mathtt{wait} \\
v & : \mathtt{id} = 0,\ \mathtt{set} = 1 \\
Z & : x_0 = 2.5,\ x_1 = 4,\ x_2 = 3 \\
\mathtt{active} & ; \mathtt{active}[0] = 3,\ \mathtt{active}[1] = 0,\ \mathtt{active}[2] = 2
\end{array}$$

Note that this swap does not change the value of id, since the scalarset elements 1 and 2 are interchanged and id contains scalarset element 0.

A number of syntactic checks have been identified that ensure that the symmetry suggested by the scalarsets is not broken. These checks are very similar to those originally identified for the Murφ verification system [19]. For instance, it is not allowed to use variables of a scalarset type for arithmetical operations such as addition. The next soundness theorem has been proven in [14]:

Theorem 1 (Soundness). *Every state swap is an automorphism.*

As a result, the representative function θ can be implemented by minimization of the state using the state swaps. Note that every state swap resembles a transposition of the state. Hence, the equivalence classes induced by the state swaps originating from a scalarset with size n consist of at most $n!$ states. The maximal theoretical gain that can be achieved using this set of automorphisms is therefore in the order of a factor $n!$.

4.2 Computation of Representatives

The representative of a state is defined as the minimal element of the symmetry class of that state w.r.t. a total order $\prec$ on the symmetry class. In general, the DBM representation of zones renders an efficient *canonical* minimization algorithm impossible, since minimization of a general DBM for any given total order using state swaps is at least as difficult as testing for graph isomorphism for strongly regular graphs [14]. If we assume, however, that the timed automaton that is analyzed resets its clocks to zero only, then the zones (DBMs) that are generated by the forward state space exploration satisfy the nice *diagonal property*. This property informally means that the individual clocks can always be ordered using the order in which they were reset. To formalize this, three binary relations on the set of clocks parameterized by a zone Z are defined:

$$x \preccurlyeq_Z y \iff \forall_{\nu \in Z}\ \nu(x) \leq \nu(y) \tag{3}$$

$$x \approx_Z y \iff \forall_{\nu \in Z}\ \nu(x) = \nu(y) \tag{4}$$

$$x \prec_Z y \iff (x \preccurlyeq_Z y \wedge \neg(x \approx_Z y)) \tag{5}$$

The diagonal property is then defined as follows.

Lemma 1 (Diagonal Property). *Consider the state space exploration algorithm described in figure* 6 *of [21]. Assume that the clocks are reset to the value 0 only. For all states* $(\boldsymbol{l}, v, Z)$ *stored in the waiting and passed list and for all clocks* x *and* y *holds that either* $x \prec_Z y$*, or* $x \approx_Z y$ *or* $y \prec_Z x$*.*

Using the reset order on clocks and the diagonal property, we can define a total order, say $\prec$, on all states within a symmetry class whose minimal element can be computed efficiently. To this end we first assume a fixed indexing of the set of clocks X: a bijection $\rho : X \to \{1, 2, \ldots, |X|\}$. Now note that $\approx_Z$ is an equivalence relation that partitions X in $P = \{X_1, X_2, \ldots, X_n\}$. We define a relation on the cells of P as follows:

$$X_i \leq X_j \iff \left(\forall_{x \in X_i, y \in X_j} x \preceq_Z y\right) \tag{6}$$

Clearly this is a total order on P. Let X_i be a cell of P. The *code* of X_i, denoted by $\mathcal{C}^*(X_i)$, then is the lexicographically sorted sequence of the indices of the clocks in X_i (the set $\{\rho(x) \mid x \in X_i\}$). The *zone code* of the zone which induced P is then defined as follows.

Definition 3 (Zone Code). *Let* Z *be a zone and let* $P = \{X_1, X_2, \ldots, X_n\}$ *be the partitioning of the set of clocks* X *under* $\approx_Z$ *such that* $i \leq j \Rightarrow X_i \leq X_j$ *(we can assume this since* $\leq$ *is a total order on* P*). The* zone code *of* Z*, denoted by* $\mathcal{C}(Z)$*, is the sequence* $(\mathcal{C}^*(X_1), \mathcal{C}^*(X_2), \ldots, \mathcal{C}^*(X_n))$*.*

Note that every zone has exactly one zone code since the indices of equivalent clocks are sorted. Moreover, zone codes can lexicographically be ordered, since they are sequences of number sequences. This order is then used in the following way to define a total order on the states in a symmetry class (the orders on the location vectors and variable valuations are just the lexicographical order on sequences of numbers):

```
(1) for all $\alpha \in \Omega$ do
(2)     for $i = 1$ to $|\alpha|$ do
(3)         for $j = 1$ to $|\alpha| - i$ do
(4)             if $swap^{\alpha}_{j-1,j}(q) \prec q$ then
(5)                 $q := swap^{\alpha}_{j-1,j}(q)$
(6)         od
(7)     od
(8) od
```

Fig. 3. Minimization of state q using the bubble-sort algorithm. The size of scalarset type α is denoted by $|\alpha|$

$$\begin{gathered}(\boldsymbol{l}, v, Z) \prec (\boldsymbol{l}', v', Z') \\ \iff \\ (\boldsymbol{l} < \boldsymbol{l}') \vee (\boldsymbol{l} = \boldsymbol{l}' \wedge v < v') \vee (\boldsymbol{l} = \boldsymbol{l}' \wedge v = v' \wedge \mathcal{C}(Z) < \mathcal{C}(Z'))\end{gathered} \tag{7}$$

We minimize the state w.r.t. the order of equation (7) using the state swaps by applying the bubble-sort algorithm to it, see Figure 3. It is clear that this representative computation satisfies the soundness equation (1), since states are transformed using the state swaps only, which are automorphisms by Theorem 1. We note that $swap^{\alpha}_{j-1,j}(q)$ is not computed explicitly for the comparison in the fourth line of the algorithm; using the statically derived γ, D_α and V_α (see section 4.1) we are able to tell whether swapping results in a smaller state.

The following theorem states the main technical contribution of our work. Informally, it means that the detected symmetries are optimally used.

Theorem 2 (Canonical Representative). *The algorithm in Figure 3 computes a canonical representative.*

Note that we assumed that arrays that are indexed by scalarsets do not contain elements of scalarsets. Otherwise, computation of a canonical representative is as hard as graph isomorphism, but this is entirely due to the discrete part of the model, and not to the clock part.

5 Experimental Results

This section presents and discusses experimental data that was obtained by the Uppaal prototype on a dual Athlon 2000+ machine with 3 GB of RAM. The measurements were done using the tool `memtime`, for which a link can be found at the Uppaal website `http://www.uppaal.com/`.

In order to demonstrate the effectiveness of symmetry reduction, the resource requirements for checking the correctness of Fischer's mutual exclusion protocol were measured as a function of the number of processes for both regular Uppaal and the prototype, see Figure 4. A conservative extrapolation of the data shows that the verification of the protocol for 20 processes without symmetry reduction would take 115 days and 1000 GB of memory, whereas this verification can be

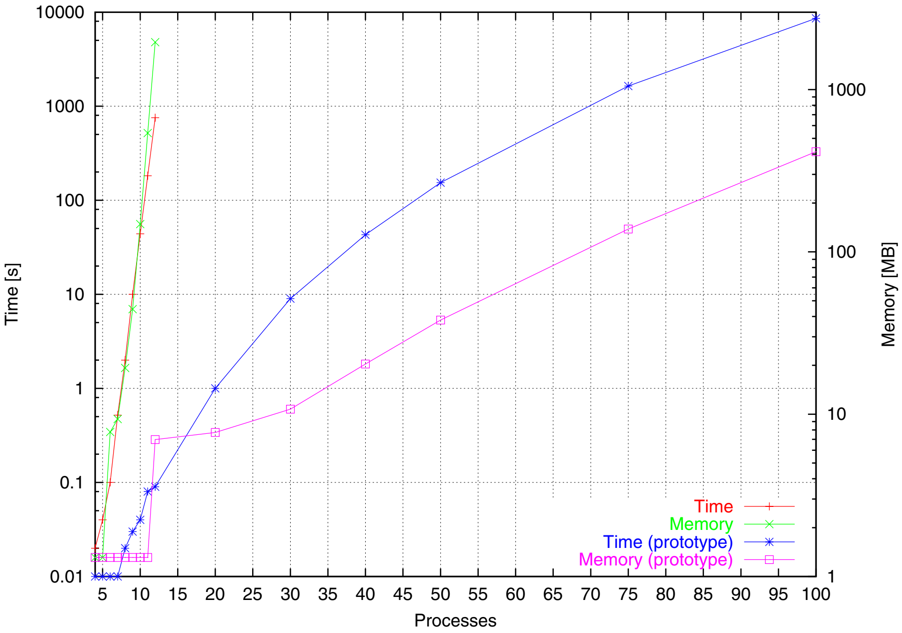

Fig. 4. Run-time data for Fischer's mutual exclusion protocol showing the enormous gain of symmetry reduction. The step in the graph of the memory usage is probably due to the the fact that Uppaal allocates memory in chunks of a few megabyte at a time

done within approximately one second using less than 10 MB of memory with symmetry reduction.

Similar results have been obtained for the CSMA/CD protocol ([24, 27]) and for the timeout task of a distributed agreement algorithm[2] [4]. To be more precise, regular Uppaal's limit for the CSMA/CD protocol is approximately 10 processes, while the prototype can easily handle 50 processes. Similarly, the prototype can easily handle 30 processes for the model of the timeout task, whereas regular Uppaal can only handle 6.

Besides the three models discussed above, we also investigated the gain of symmetry reduction for two more complex models. First, we experimented with the previously mentioned agreement algorithm, of which we are unable to verify an interesting instance even with symmetry reduction due to the size of the state space. Nevertheless, symmetry reduction showed a very significant improvement. Second, we experimented with a model of Bang & Olufsen's audio/video protocol [13]. The mentioned paper describes how Uppaal is used to find a bug in the protocol, and it describes the verification of the corrected protocol for two (symmetric) senders. Naturally, we added another sender – verification of the model

[2] Models of the agreement algorithm and its timeout task are available through the URL `http://www.cs.kun.nl/∼martijnh/`

Table 1. Comparing the time and memory consumption of the relations for the agreement algorithm and for Bang & Olufsen's audio/video protocol with two and three senders. The exact parameters of the agreement model are the following: $n = 2$, $f = 1$, $ones = 0$, $c_1 = 1$, $c_2 = 2$ and d varied (the value is written between the brackets). Furthermore, the measurements were done for the verification of the agreement invariant only. Three verification runs were measured for each model and the best one w.r.t. time is shown

Model	**Time [s]**		**Memory [MB]**	
	No reduction	Reduction	No reduction	Reduction
Agreement (0)	1	3	33	45
Agreement (1)	21	16	294	180
Agreement (2)	80	23	905	245
Agreement (3)	231	32	2126	321
B&O (2)	2	1	16	10
B&O (3)	265	36	1109	181

for three senders was impossible at the time of the first verification attempt – and we found another bug, whose source and implications we are investigating at the time of this writing. Table 1 shows run-time data for these models.

6 Conclusions

The results we obtained with our prototype are clearly quite promising: with relatively limited changes/extensions of the Uppaal code we obtain a rather drastic improvement of performance for systems with symmetry that can be expressed using scalarsets.

An obvious next step is to do experiments concerning profiling where computation time is spent, and in particular how much time is spent on computing representatives. In the tool Design/CPN [18, 20, 11] (where symmetry reduction is a main reduction mechanism) there have been interesting prototype experiments with an implementation in which the (expensive) computations of representatives were launched as tasks to be solved in parallel with the main exploration algorithm.

The scalarset approach that we follow in this paper only allows one to express total symmetries. An obvious direction for future research will be to study how other types of symmetry (for instance as we see it in a token ring) can be exploited.

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. 47

[2] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993. 47

[3] 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. 47

[4] 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. 47, 56

[5] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957. 52

[6] D. Bosnacki, D. Dams, and L. Holenderski. A heuristic for symmetry reductions with scalarsets. In J.N. Oliveira and P. Zave, editors, *FME 2001*, number 2021 in LNCS, pages 518–533. Springer–Verlag, 2001. 47

[7] 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. 47

[8] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 2000. 46

[9] 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. 52

[10] 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. 46, 47, 50

[11] L. Elgaard. *The Symmetry Method for Coloured Petri Nets - Theory, Tools, and Practical Use*. PhD thesis, Department of Computing Science, University of Aarhus, Denmark, July 2002. 57

[12] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *CAV'93*, number 697 in LNCS. Springer–Verlag, 1993. 47

[13] K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *18th IEEE Real-Time Systems Symposium*, pages 2–13, 1997. 47, 56

[14] M. Hendriks. Enhancing UPPAAL by exploiting symmetry. Technical Report NIII-R0208, NIII, University of Nijmegen, October 2002. 47, 48, 52, 53, 54

[15] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to UPPAAL. Technical Report NIII-R03xx, NIII, University of Nijmegen, 2003. To appear. 48

[16] T. A. Henzinger, P. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997. 46

[17] G. J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 46, 47

[18] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level petri nets. *Theoretical Computer Science*, 45(3):261–292, 1986. 47, 57

[19] 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. Journal version appeared in *Formal Methods in System Design*, 9(1/2):41–75, 1996. 47, 48, 49, 50, 53

[20] K. Jensen. Condensed state spaces for symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996. 47, 57

[21] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, pages 134–152, 1998. 46, 47, 50, 54

[22] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1992. 46, 47

[23] P.H. Starke. Reachability analysis of petri nets using symmetries. *Syst. Anal. Model. Simul./5*, 8(4):293–303, 1991. 47

[24] A. S. Tanenbaum. *Computer Networks*. Prentice–Hall, 1996. 47, 56

[25] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In S. Graf and M. Schwartzbach, editors, *TACAS'00*, number 1785 in LNCS, pages 157–171. Springer–Verlag, 2000. 46, 47

[26] F. Wang and K. Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In D.A. Peled and M.Y. Vardi, editors, *FORTE'02*, number 2529 in LNCS, pages 50–64. Springer–Verlag, 2002. 47

[27] S. Yovine. Kronos: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(2), 1997. 46, 47, 56