Search Algorithms for Automated Validation

Tom A.N. Engels, Jan Friso Groote,
Muck J. van Weerdenburg, and Tim A.C. Willemse

Design and Analysis of Systems Group,
Eindhoven University of Technology – The Netherlands

Abstract. A novel search technique called *highway search* is introduced. The search technique relies on a *highway simulation* which takes several homogeneous walks through a (possibly infinite) state space. Furthermore, we provide a memory-efficient algorithm that approximates a highway search and we prove that, under particular conditions, they coincide. The effectiveness of highway search is compared to two mainstream search techniques, viz. *random search* and *depth-first search*. Our results demonstrate that, when applied to small but complex substructures in isolation, depth-first search and highway search are orthogonal, but have equal merits, whereas overall performance of random search is poor. In contrast, experiments conducted using state spaces that are based on real systems, highway search ranks highest. The next best choice is a random search, while depth-first search is always outperformed.

1 Introduction

Random simulation is used in many engineering disciplines as a technique to validate the correctness of a design and get acquainted with its properties. Its popularity is explained largely from the fact that it is push-button technology, limiting user input to a bare minimum. Adding to its popularity are the appeal to intuition that is behind the method, and the sense that a random simulation reflects the typical behaviours of a system.

While random simulation is a proper tool for some designs, its use in a non-deterministic setting is certainly less obvious. This is because the results that are obtained by means of a random simulation are often a poor reflection of the overall behaviour of the design. This is amplified in non-deterministic designs with huge or even infinite state spaces, in which a single simulation visits an insignificant part of the entire state space. In fact, Pelánek *et al* [7] demonstrated that for random simulation, the frequency of visits of states has the power law distribution rather than a uniform distribution.

Still, random simulation is often useful when the primary verification tools such as model-checking tools (see e.g. [4]) and theorem proving are incapable of dealing with the problem at hand due to the infamous phenomenon known as "state-space explosion". In such cases, random simulation can be used to search for particularly interesting states or events, leading to a *random search*.

In this paper, we demonstrate that there is ample room for improvement in random simulation. We investigate a technique called *highway simulation* (with the associated search technique called *highway search*), which can be implemented straightforwardly.

Furthermore, we provide an algorithm for conducting an approximation of a highway simulation, and we study its properties. Experiments indicate that in practice, an approximate highway search and an ideal highway search do not differ significantly.

Highway simulation is akin to a *restricted breadth-first exploration* [8], which provides a graceful degradation from breadth-first exploration to random simulation, and *parallel random simulation* [7]. The simulation technique (and its associated search technique) remains push-button technology, requiring little to no human intervention. The effectiveness of the search method is compared with standard search methods such as *depth-first search* [4] and random search.

This paper is structured as follows: in Section 2, we provide a brief exposition of our formal framework which we use to explain the simulation and search methods. In Section 3 the search methods are introduced and in Section 4 we describe several experiments that we used to study the various search techniques. Related work is discussed in Section 5 and we present our conclusions in Section 6.

2 Framework

We present our techniques in the setting of Transition Systems (TSs). We stress that the notions that we subsequently develop are easily recast into richer frameworks such as *Labelled Transition Systems*.

Definition 1. *A* Transition System *is a three-tuple $\langle S, s_0, \rightarrow \rangle$, where S is a possibly infinite set of states, referred to as the* state space*, $s_0 \in S$ is the initial state and $\rightarrow \subseteq S \times S$ is the transition relation. We write $s \rightarrow t$ instead of $(s,t) \in \rightarrow$.*

Throughout this section, we assume an arbitrary TS $\mathcal{S} = \langle S, s_0, \rightarrow \rangle$. Let $s \in S$ be an arbitrary state. The set of *successor states* of s is defined as $\mathsf{succ}(s) = \{s' \mid s \rightarrow s'\}$; generalised to a set of states S', we have $\mathsf{succ}(S') = \bigcup_{s' \in S'} \mathsf{succ}(s')$. The TSs that we consider in this paper are *finitely branching*, i.e. for each state $s \in S$, we require $|\mathsf{succ}(s)| \in \mathbb{N}$. A *path* starting in s is a sequence of states $\sigma \equiv t_0\ t_1\ \ldots\ t_n$ for $n \geq 0$ satisfying $t_0 = s$ and $t_{i+1} \in \mathsf{succ}(t_i)$. The state s is *reachable* iff there is a path starting in s_0 ending in s; the set of reachable states of a finite-state TS can be found by means of an exhaustive depth-first or breadth-first search. For TSs with an infinite state space, or a finite but extremely large state space, an exhaustive search is not feasible because of memory and/or time restrictions. Therefore, some approaches resort to *probabilistic methods*. The most intuitive and straightforward probabilistic state-space exploration method is that of random simulation.

Definition 2. *A* random simulation *of the state space is a path σ starting in s_0, for which each s_{i+1} is obtained by a random draw from the set $\mathsf{succ}(s_i)$.*

Random simulation suffers from the problem of a poor coverage of the state space (see e.g. [8, 7]): in the presence of loops and non-determinism, a random simulation may exhibit revisits of states, leading to a power law distribution for the frequency of state visits. Clearly, unnecessary revisits of states of a state space do not add to the coverage of the state space, even though such revisits sometimes have a surprisingly rewarding effect (see Section 4.2).

3 Random and Highway Search

We define random search. Here we denote the set of *visited states* of a path σ by $V(\sigma)$, i.e. $V(\sigma)$ contains all states that appear in σ. Both random and highway search consider all successor states in choosing transitions. The set of all *considered states* of a path σ is denoted by $C(\sigma)$, where for all paths σ' and states s, $C(\sigma' \cdot s) = V(\sigma' \cdot s) \cup \mathsf{succ}(V(\sigma'))$. Note that the successors of the final state of a simulation are not considered as no transition needs to be chosen from such a state.

Definition 3. *The result of a* random search *for a set of states $T \subseteq S$ is defined as the set of states $T \cap C(\sigma)$ that is obtained by means of a single random simulation σ.*

Clearly, a random search has drawbacks that can be linked to the drawbacks of the random simulation. Thus, random search as a tool for validating a design seems to be far from optimal. We therefore focus on a hybrid breed of *breadth-first exploration* and random simulation. The resulting simulation method is dubbed *highway simulation*.

Definition 4. *Let S and T be arbitrary sets and let $N > 0$ be an arbitrary natural number. We say that T is N-*close to *S iff $T \subseteq S$ and $|T| = \mathsf{min}(N, |S|)$.*

Next, we generalise the notion of visited states as follows. Let $\boldsymbol{Q} \in (2^S)^*$ be a sequence of sets of states. The set of visited states of $\boldsymbol{Q}$ is denoted $V(\boldsymbol{Q})$, and is defined as the union of all Q that occur in $\boldsymbol{Q}$. The set of considered states, denoted $C(\boldsymbol{Q})$, where for all sequences of sets of states $\boldsymbol{Q}$ and sets of states Q, $C(\boldsymbol{Q} \cdot Q) = V(\boldsymbol{Q} \cdot Q) \cup \mathsf{succ}(V(\boldsymbol{Q}))$.

Definition 5. *A* highway simulation *of width $N > 0$ is defined as a sequence of sets of states $\boldsymbol{Q} \in (2^S)^*$ where $\boldsymbol{Q} = Q_0 \ \ldots \ Q_d$ satisfies:*

- *$Q_0 = \{s_0\}$ and*
- *each Q_j (for $j > 0$) is obtained by a random draw from the set P_j:*

$$P_j := \{S' \mid S' \text{ is } N\text{-close to } \mathsf{succ}(Q_{j-1}) \setminus \bigcup_{k<j} Q_k\}$$

The result of a highway search *of width $N > 0$ for a set of states T is defined as the set of states $T \cap C(\boldsymbol{Q})$ that is obtained by means of a single highway simulation $\boldsymbol{Q}$ of width N.*

Intuitively, a highway simulation simultaneously explores several *lanes* (set of paths, in our formal terminology), which at each instance can merge or split into new lanes; *exits* (outgoing transition of a state) of a lane are only considered if these lead to new *places* (states, in our formal terminology). Note that the latter requirement may lead to the (premature) termination of a path. The *highway* can be thought of as a collection of lanes. Figure 1 visualises a typical highway simulation of width 2. The places are indicated by bullets, and the lanes are represented by the small ovals. The large ovals visualise the reachable places from a preceding lane. The exits that are ignored are printed by a dotted line (such exits lead to already visited places and are therefore not *eligible*) or a dashed line (these exits are ignored by chance), whereas the actually taken exits are represented by a solid line.

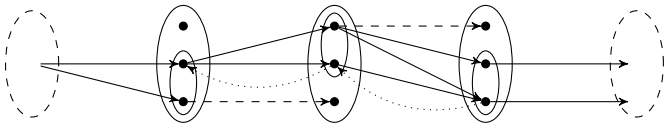

Fig. 1. Typical paths through a transition system.

Property 1. Let $\boldsymbol{Q} = Q_0 \dots Q_d$ be a highway simulation of width $N > 0$.

- For all i, j, $(i \neq j)$, $Q_i \cap Q_j = \emptyset$,
- The number of paths starting in s_0, implicitly defined by $\boldsymbol{Q}$ is bound from below by $\sum_{i=1}^{d} |Q_i|$.

Highway simulation (and, hence also a highway search) can be implemented straightforwardly. Such an implementation essentially requires:

1. Memorising the places that have been visited along a lane,
2. Computing the set of eligible reachable places from the current lanes and taking a homogeneous N-close selection of this set.

In case of a large highway width N and a large branching degree of the places in the current lane, first computing the eligible reachable places in step 2, and subsequently taking a selection of these places can draw significantly on memory resources and computation times.

The noted problem can be avoided using a probabilistic construction for computing the next set of places in an *on-the-fly* fashion. Algorithm 1 implements this solution. Theorem 1 presented below formalises the assumptions that guarantee that Alg. 1 correctly implements a highway simulation.

Theorem 1. *Let Q_d be an arbitrary set of places obtained at iteration d in Algorithm 1. Let $V = \bigcup_{i \leq d} Q_i$ be the set of places that have been visited and $C = \mathsf{succ}(Q_d) \setminus V$. If for all different places $s_1, s_2 \in Q_d$, $\mathsf{succ}(s_1) \cap \mathsf{succ}(s_2) \subseteq V$, then for all $s \in C$:*

1. *if $|C| \leq N$ then $P(s \in Q_{d+1}) = 1$.*
2. *if $|C| > N$ then $P(s \in Q_{d+1}) = \frac{N}{|C|}$.*

Proof. The first case is trivially satisfied, as in line 8 all non-revisited successor places are added to Q_{d+1} so long as the set Q_{d+1} is not full (i.e. $|Q_{d+1}| < N$). The second case is settled by means of an inductive proof. In particular, we prove that the addition of place t in line 11 should be done with probability $\frac{N}{c}$ and at random in the set. With induction on the difference $c - N$ we show that doing so leads to a uniform distribution (i.e. every place has probability $\frac{N}{|C|}$ of being in the set):

1. Base case: $c - N = 0$. This is trivial.
2. Inductive step: $c - N = k + 1$ for some k. We added the last place, t, with probability $\frac{N}{c}$, so we need only consider the other places. Before adding place t they are in the set with probability $\frac{N}{c-1}$ (by induction) and when the last place has been considered,

Algorithm 1 Approximate Highway simulation

```
1: V, d, Q_0 = {s_0}, 0, {s_0};
2: while Q_d ≠ ∅ do
3:     c, Q_{d+1} = 0, ∅;
4:     for s ∈ Q_d do
5:         for t ∈ succ(s) do
6:             if t ∉ V ∪ Q_{d+1} then
7:                 c = c + 1;
8:                 if c ≤ N then Q_{d+1} = Q_{d+1} ∪ {t};
9:                 else with probability N/c
10:                    select randomly u ∈ Q_{d+1};
11:                    Q_{d+1} = (Q_{d+1} \ {u}) ∪ {t};
12:                end if
13:            end if
14:        end for
15:    end for
16:    V, d = V ∪ Q_{d+1}, d + 1;
17: end while
```

they are still in the set afterwards with probability $\frac{N}{c}\ (1 - \frac{1}{N}) + (1 - \frac{N}{c})$. That is, if the new element is added, then each currently selected place has chance $\frac{1}{N}$ of being removed (or $1 - \frac{1}{N}$ of not being removed). If t is not added, then the selected elements obviously remain selected. It is straightforward to derive that

$$\frac{N}{c-1}\ (\frac{N}{c}\ (1 - \frac{1}{N}) + (1 - \frac{N}{c})) = \frac{N}{c}$$

Indeed, Alg. 1 correctly implements a highway simulation whenever two places in a lane only lead to the same place if that place was already visited; all other places that can be reached should be unique.

In practice, this property is difficult to assess upfront. In degenerate cases, the selection mechanism for the next places that can be reached by the current lanes in Alg. 1 strongly favours *funnels*: eligible reachable places that are shared among a significant amount of places in the current lanes. This is confirmed by the following analysis.

Property 2. Let Q be the selection at some point of some depth d of Alg. 1 for width N. Further, let c be the number of places actually considered for addition up to that moment. We analyse the probability of a place to end up in a lane of the highway simulation when that place is considered for addition multiple times.

Let $s \notin Q$ be a place. Suppose that we successively try to add s, K ($K > 0$) times. The probability that s is in Q afterwards is obtained as follows. Let probabilistic variable X_k denote the conditional probability of adding place s to Q after K attempts when s fails to be added for the first $k \leq K$ attempts.

$$P(X_k) = \begin{cases} 0 & \text{if } k = K \\ \frac{N}{c+k+1} + (1 - \frac{N}{c+k+1})P(X_{k+1}) & \text{if } 0 \leq k < K \end{cases} \tag{1}$$

The above expression can be simplified, yielding the following probability function:

$$P(X_k) = 1 - \frac{(c+k)!(c+K-N)!}{(c+K)!(c+k-N)!} \tag{2}$$

For $k = 0$, we then obtain $P(X_0) = 1 - \frac{c!(c+K-N)!}{(c+K)!(c-N)!}$.

To illustrate the effect for an approximate highway simulation of width $N = 1$ we have that the probability that s will be selected is $\frac{K}{c+K}$ (as opposed to the more desirable $\frac{1}{c+1}$).

We can see that K and N have a positive relation to the probability of duplicates being chosen and that c has a negative relation to it.

Next, observe that a place has the greatest probability of being selected if all of its occurrences are considered last. This implies that the probability that a place that occurs K times in a sequence of size M consisting of U ($U > N$) unique elements is selected with a probability in the interval

$$\left[\frac{N}{M},\ 1 - \frac{(U-1)!((U-1)+K-N)!}{((U-1)+K)!((U-1)-N)!}\right] \tag{3}$$

From the above analysis, it follows that in degenerate cases, funnels are more likely to appear in the highway simulation than ordinary places.

Property 3. When every place t occurs an equal amount of times as successor of a place $s \in Q_d$ and all successors are treated in a uniformly distributed way, then the probability of place t appearing in Q_{d+1} approaches $\frac{N}{U}$, where U is the number of unique successors.

While the preceding analysis suggests a highway simulation and an approximate highway simulation behave very differently, this is not supported by the experiments that we conducted in Section 4. In these experiments, the difference turns out to be insignificant for most applications, suggesting that the degenerate cases really are less likely to occur than the optimal cases. We expect that this is due to the small probability of funnels to be present in small highway widths.

4 Experiments

In practice, it is difficult to assess which simulation/search technique best fits a given problem. We conducted a number of experiments on both classical and novel problems using three search techniques: random search, depth-first search and approximate highway search. In the only case where approximate highway search differs significantly from the *ideal* highway search (see Section 4.2), we added the latter in our exposition of the particular experiment.

Apart from analysing the effectiveness of the search techniques on problems taken from practice, we first study the impact of commonly found substructures on the behaviour of the search techniques. The latter serves a better understanding of the intricacies of the search techniques.

4.1 Complex (Sub)structures

Most transition systems stemming from real applications are of moderate overall structural complexity, but contain substructures that are particularly difficult to search through. Typical examples are *diamond structures*, that arise due to the interleaving semantics that is given to parallel components, and *back-loops* that appear due to recovery mechanisms or resets, *etcetera*.

Diamond Structure. Diamond-structured state spaces are a known source of complexity for random-based search techniques [7]. The probability to enter into the outer ends of a diamond structure is virtually zero: the outermost states in a diamond of width n can only be reached via 1 of the 2^{n-1} paths of length $n-1$; in general, about 95% of the paths of length $n-1$ lead to just over 40% of the states at depth n.

Setup and Analysis Our experiments involve a diamond structure of width 10, shown in Fig. 2. The search objective is any state numbered 1 through 10 (indicated in Fig. 2 by the states 1 and 10 and all 8 states in between). In our experiments, we analyse the probability of hitting one of the numbered states in the state space. The resulting distributions of random search and highway searches of width 2,4,6,8 and 10 are depicted in Fig. 2. Depth-first search yields a consistent 100% for all values, as does a highway search of width 10, but compared to a depth-first search, the latter requires only half as many states to be explored

The effect of the probabilistic approaches used in a random search and in highway searches of small width is clearly visible in Fig. 2. As expected, a random search covers fewer states than a highway search and highway searches of increasing width cover more states more often.

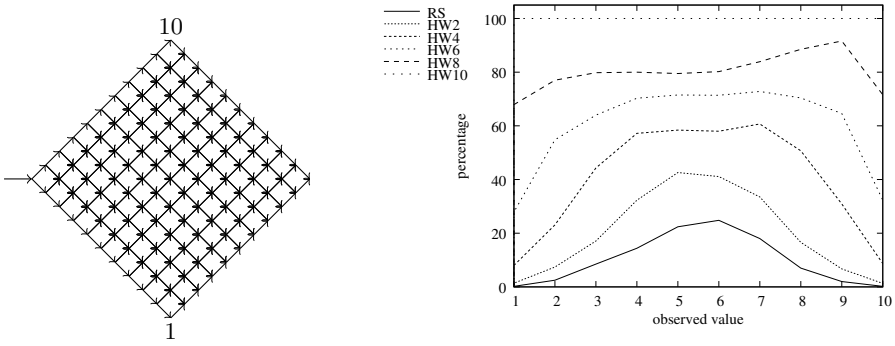

Fig. 2. Diamond structure of width 10 (left) and search results (right).

Back-loops Back-loops occur frequently due to *system resets* and *recovery mechanisms*. A random search tends to revisit states instead of exploring new states in such situations.

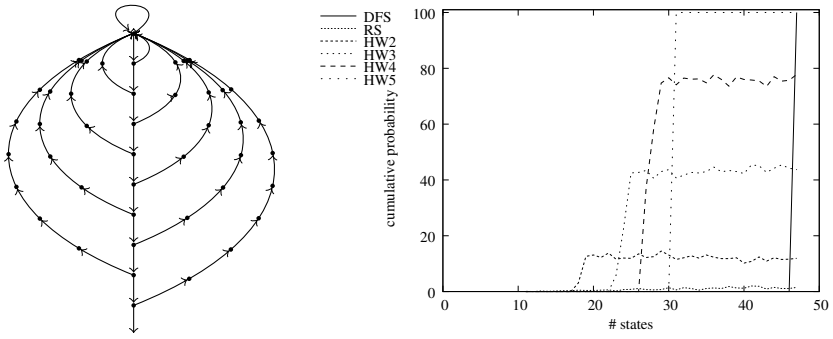

Fig. 3. Back-loop system of 47 states (left) and search results (right).

Setup and Analysis We used a back-loop system consisting of 47 states (see Fig. 3) to demonstrate the efficacy of all search techniques in finding the (single) deadlock in the system. The plots in Fig. 3 show the probability of each search method of finding the deadlock against the number of states that have been visited.

Note that depth-first search performs quite poorly. Its performance cannot be improved upon by repeated runs due to its deterministic character, which happens to select the least efficient strategy for reaching the deadlocking state repeatedly. Note that the probability of finding the deadlock this late (but also as early as 11 steps) is quite slim.

The results of random search confirm its revisiting behaviour; even after 50 steps it cannot find the deadlock with a significant probability. knowledge. A highway search of width 1 (not shown) is outperformed by all searches due to the inability to revisit states. The results of highway searches show that each width has an optimum that can be achieved; exploring more states does not lead to better detection rates.

Strongly Connected Components Substructures that contain a *strongly connected component* "absorb" a probability-based search: the search is confined to a single SCC. The behaviour of a highway search is, due to a more balanced search strategy that allows it to investigate multiple SCCs, subtly different.

Setup and Analysis The state space we study consists of five reachable strongly connected components with single exits, see Fig. 4. The impact of the number of states to be explored on each search method's capability of finding one of the numbered states is also depicted in Fig. 4.

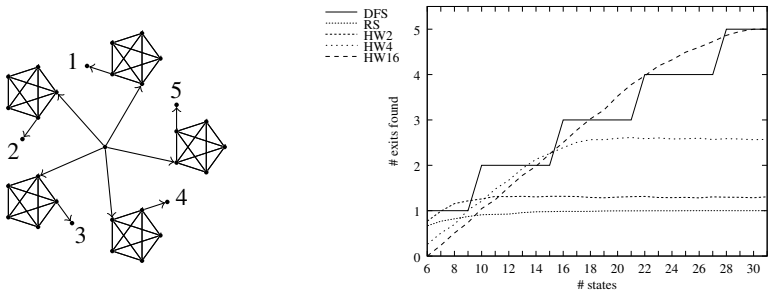

Fig. 4. Five Strongly Connected Components (left) and search results (right).

Depth-first search, being an exhaustive search method, examines each connected component only once (but exhaustively) before turning its attention to another component. This causes the stepwise behaviour as shown in Fig. 4. A highway search of width 1 slightly outperforms a random search as it will always find (precisely) one exit while random search sometimes misses one because of revisits. At wider widths, a highway search starts to approach depth-first search. The numbered states that are found using a highway search are not always the same, whereas a depth-first search consistently finds the same numbered states.

4.2 Applications

The results of previous section do not straightforwardly transfer to practical situations, although they do provide additional intuition behind each search method's capabilities. In this section, we study the effectiveness of the search techniques on documented cases which are either inherently large or can be scaled up to become too large for model checking purposes.

A Network of Buffers The analysis of system performance is computationally quite costly or even infeasible. A point in case is the performance analysis of a network of buffers. Variabilities such as buffer capacity, channel speeds, mutual exclusion protocols, *etcetera* affect throughput in ways that are hard to predict.

Setup and Analysis The setup depicted in Fig. 5 is a simplification of a proprietary industrial protocol [2] for quickly collecting data from a large number of distributed receivers; each buffer and channel in the network introduces its own delay, and a mutual exclusion protocol is used to enforce singular communications between the buffer and the receiver. We search for extreme propagation times of 100 data units through the network configuration of Fig. 5.

Our model of the network protocol terminates after 100 data units have been received. Since the protocol does not contain loops or cycles, each path in the protocol is finite and equally long (202 states). We set an upper bound of $202 \times N$ on the number of

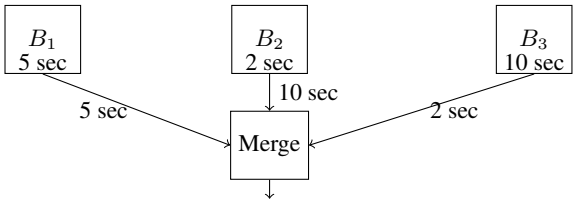

Fig. 5. A network configuration of depth 2 and width 3, with propagation times and channel delays in seconds.

states that are allowed to be explored by each search method (N depends on a highway search width).

In Fig. 6 we report the minimal and maximal propagation times that are found by runs of each search method (limited to $202 \times N$ states). Comparing the intervals of propagation times that are reported by each search method, we find that a highway search of width 2 is comparable to a random search, and outperformed by a depth-first search. For highway searches of width 1,000 and 2,000, we see that highway search ranks highest, followed by random search and finally depth-first search. This suggests that if one has little resources, a depth-first search is preferable, whereas when resources are less scarce, highway search can be the preferred choice.

method → ↓ N(width)	RS	DFS	HS$_N$
1	1129	1000	1124
2	1065 – 1075	994 – 1020	1142 – 1147
1000	1014 – 1194	973 – 1085	1010 – 1265
2000	1016 – 1196	970 – 1100	998 – 1284

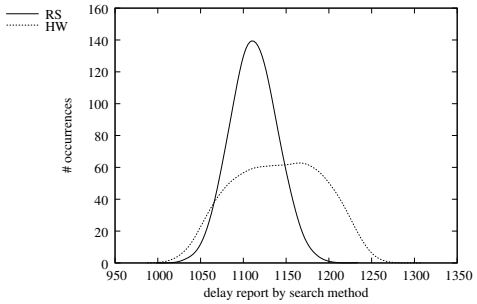

Fig. 6. Minimal and maximal propagation times found using Random Search, Depth-First Search and Highway Search of widths 1, 2, 1,000 and 2,000 (left), and typical probabilities for finding propagation times using Random Search and Highway Search of width 1,000.

Note that a single run is not necessarily representative for the extremities that can be found using random search or highway search. Fig. 6 shows the average probability of finding a particular propagation time for 100 highway searches of width 1,000 compared to a 100,000 random searches. It shows that a random search is less likely to find extreme values than a highway search.

A Faulty Sliding Window Protocol The *Sliding Window Protocol* is a complex data communications protocol; the complexity of protocols of this type quickly leads to conceptual flaws, which is testified by the nasty livelock that is present in a description of the protocol in early versions of [9] (i.e. the protocol using a selective repetition, first, second and third edition).

We have used a formalisation of the protocol [1] to search for the livelock. The protocol's state space for a window size greater than 1 quickly grows beyond verifyable proportions, which makes the protocol an excellent benchmark for alternative validation techniques.

Setup and Analysis We used all three search methods to find the livelock in the protocol with window size 2. As a measure for success, we have used the number of states and the length of the simulation that are needed to observe the first occurrence of the livelock, respectively. The absence of a plot for depth-first search is due to the fact that we found

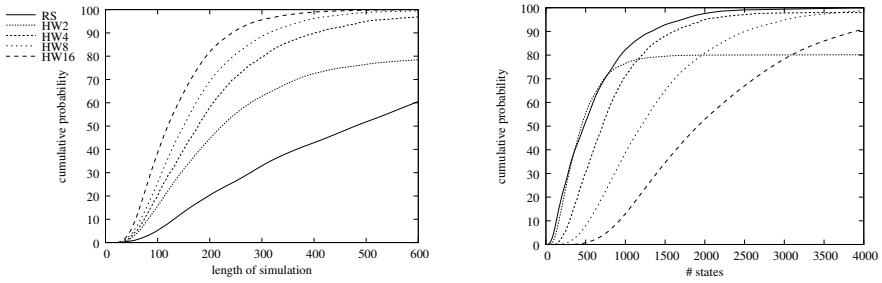

Fig. 7. Percentage of flaws found by random search and highway searches of various widths for the faulty Sliding Window Protocol.

this livelock only after examining 8,255 states.

Figure 7 clearly illustrates that a random search is outperformed by highway searches of width 2 and wider when considering the simulation length that is required to find the livelock. However, when we look at the number of states that are required to find the livelock, random search beats all highway searches. This means that in case one wishes short witnesses to the search objective, a highway search of greater width is more appropriate, whereas if one wishes to minimise memory usage, a random search is best.

Automated Parking Garage In [6] an architecture for an automated parking garage is developed and verified. The main obstacles in verification of the design is its very large state space. Instead of applying abstraction to reduce the state space to manageable proportions (as was done in [6]), we analyse the original specification, which contains two known faults. A complicating factor in this design is that the search space contains many loops and cycles.

Setup and Analysis We are interested in the (cumulative) probability of finding the error upon having examined a given number of states, and a given simulation length, respectively.

The results are depicted in Fig. 8. In line with the observations for the Sliding Window Protocol, depth-first search is not visualised since we did not find the error with it until 886,946 states had been explored (using 4.5GB of memory). On average, a random simulation visiting 4,000 states is needed to find the error. In contrast, a highway search of width 1 outperforms random search significantly, requiring a simulation that visits less than 400 states to reach the error.

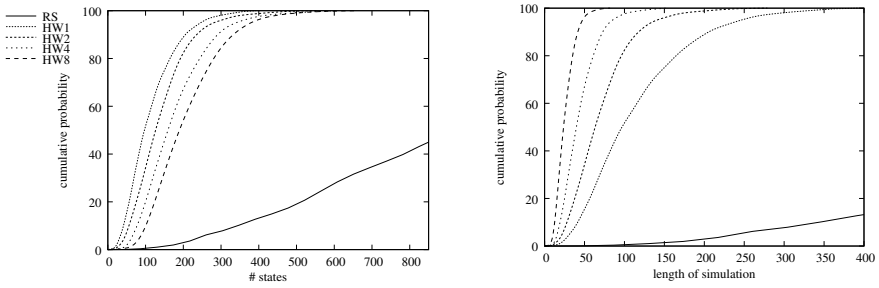

Fig. 8. Search results for the Automated Parking Garage.

When we look at the simulation lengths that are required to find an error, a different picture emerges: while random search still performs quite poorly, compared to a highway search of width 16, a width 1 search is significantly worse at finding the error quickly, meaning that a witness that leads to the error is less complex at increasing widths.

Dining Philosophers The *dining philosophers* problem is a problem that clearly illustrates the subtleties and complexity of synchronisation in concurrency. An advantage of the problem is that it is highly scalable, yet always exhibits a single deadlocking situation. We have examined two instances of the dining philosophers problem: an instance with 5 and an instance with 17 philosophers.

Setup and Analysis The deadlock search problem in the instance of the dining philosophers problem with 17 philosophers is inspired by [3]. The results of a highway search of width 2 through 5 and a random search are depicted in Fig. 9. For completeness sake, we mention that we found the deadlock with a depth-first search after having visited 15,481,578 states, which required up-to 25 GB.

Surprisingly, a random search outperforms all highway searches. Even more surprising is the observation that highway searches of greater width perform poorer than highway searches of smaller width, meaning that the witnesses leading to the deadlock states become more complex at increasing highway search widths.

We use the instance of 5 philosophers to further illustrate these findings. In Fig. 9, the width of a highway search (horizontal axis) is plotted against the total probability of detecting the deadlock. The probability is obtained as an average of 4,000 experiments for each highway search width. Note that each search terminates either due to revisits of states, as the total state space is finite.

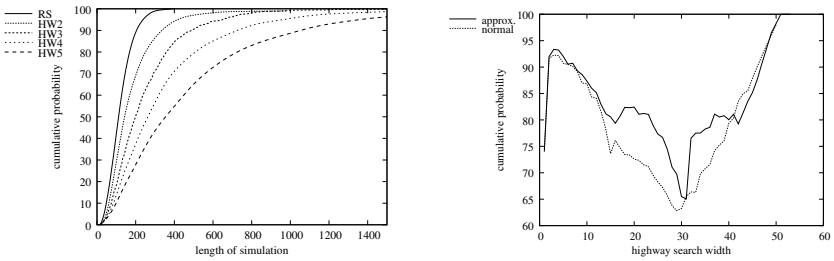

Fig. 9. Cumulative probability of finding the deadlock in the dining philosophers (instance 17) using various search methods (left), and total probability of finding the deadlock in the dining philosopher's problem (instance 5) at increasing highway widths and approximate highway widths.

An explanation for this remarkable situation can be found in the unusual structure of the state space which has features of the *back-loops* problem of the previous section. Increasing the width of a highway search increases the probability that paths to the deadlocking state gets disabled. Since one can only return to a path to a deadlock if one revisits a state earlier on this path (which is impossible in a highway search), the total probability of reaching a deadlock decreases.

5 Related Work

A general overview of different search methods is provided in [7]. Our diamond example of Section 4.1 is taken from this paper. The authors show by means of experiments that the probability of visiting a state in an arbitrary state space using a random simulation has the power law distribution. This means that a random simulation "spends most of the time repeatedly visiting just a few states". While most of our studies indicate that this indeed hampers the search efficacy, the *dining philosopher* problem of Section 4.2 illustrates that this property can be useful.

Godefroid and Kurshid [3] introduce a search technique based on *genetic algorithms*. Fitness functions (heuristics) that are tailored to the property that is searched for are used for mutation and selection. Their experiments conducted on an instance of the dining philosophers problem with 17 philosophers shows that their technique outperforms random search, which yields no successes in 8 hours when restricted to runs of 68 states. Unfortunately, our analysis of the same problem in Section 4.2 cannot confirm these findings; on the contrary: a random search limited to runs of 68 states

on the same instance of 17 philosophers finds the deadlock in 20% of the cases within seconds. It is unclear to us what causes these differences in observations. A separate analysis conducted on our side confirms that our random search is as random as can possibly be achieved.

Beam-search [10] is another method of efficiently searching through a state space. A beam-search, however, is guided by semantic knowledge about the state space, essentially *guiding* the search to the places of interest. In this respect, it requires significantly more effort to conduct a beam-search (and, in fact, also to conduct a search using genetic algorithms) than to conduct any of the searches that we studied in this paper.

6 Conclusion

We studied a novel search technique, called *highway search* for validating large system designs. We compared its efficacy with two other standard search methods, viz. *depth-first search* and *random search*. Their performance is assessed by means of searches through small but complex state spaces (see Section 4.1) and large state spaces that come from practice or appeared in the literature (see Section 4.2).

In order to reduce memory requirements we presented a memory-efficient algorithm approximating a highway simulation (which underlies a highway search). We prove that under specific conditions, it faithfully implements a highway simulation, and that its worst-case behaviour is significantly different from the ideal highway simulation. In practice, this difference is hardly ever experienced, and, in the only case in which we found a significant difference, we find that approximate highway search outperforms ideal highway search.

Summarising the results of our experiments, we find that on the complex substructures, a random search is often outperformed by depth-first search and highway search, whereas quantitatively, highway searches and depth-first search perform comparable. Qualitatively, a highway search and a depth-first search behave quite differently, due to the random nature of a highway search and the deterministic nature of a depth-first search. The former typically yields different results in successive searches, whereas the latter's results remains constant. It depends on the application domain which one is to be preferred. Note that while executing the experiments we have also experienced that the performance of depth-first search highly depends on the structure of the input; it is only deterministic for a given input and a given version of the tooling.

For larger state spaces with practical value, the results are different. The efficacy of depth-first search completely disappears: its deterministic nature which is an advantage in small state spaces is the root cause. Depth-first search is in all our instances significantly outperformed by highway search and random search. Considering the number of states required to arrive at the search objective, no firm conclusions can be drawn: in case of the Automated Parking Garage, a highway search significantly outperforms a random search, whereas both perform equally good in the Sliding Window Protocol example. Taking the simulation length as a measure of success, highway search is clearly more suited, as suggested by the same examples. We remark that increasing the highway search width in general leads to shorter witnesses explaining the reachability of a particular state, providing a better understanding.

Finally, the experiments show that for obtaining a more homogeneous coverage of a state space, a highway search is also more valuable than a random search or a depth-first search (see the Network example).

As future work, it is interesting to extend the comparison of search techniques to also cover more involved techniques such as *beam-search* and searches using genetic algorithms, although the latter two require more human intellect than the techniques that we studied in this paper. Furthermore, we strive at extending the set of cases (both of theoretical substructures and of practical cases) for further assessment of the search methods.

References

1. J.J. Brunekreef. A formal specification of three sliding window protocols. Technical Report P9102, CWI, 1991.
2. T.A.N. Engels. Analysis of the irf digital network. Master's thesis, Eindhoven University of Technology, 2007.
3. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In J.-P. Katoen and P. Stevens, editors, *Proceedings of TACAS 2002*, volume 2280 of *LNCS*, pages 266–280. Springer-Verlag, 2002.
4. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
5. A. Mathijssen and A.J. Pretorius. Specification, analysis and verification of an automated parking garage. Technical report, Eindhoven University of Technology, 2005.
6. A. Mathijssen and A.J. Pretorius. Verified design of an automated parking garage. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2007.
7. R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing Random Walk State Space Exploration. In *Tenth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 05)*, pages 98–105, New York, NY, USA, Sept. 2005. ACM Press.
8. H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory checking. *ENTCS*, 89(1):51–67, 2003.
9. A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1988.
10. A.J. Wijs and B. Lisser. Distributed extended beam search for quantitative model checking. In *Proc. 4th Workshop on Model Checking and Artificial Intelligence (MoChArt'06)*, volume 4428 of *LNAI*, pages 165–182. Springer-Verlag, 2007.

A Specifications of Benchmark Systems

In order to be able to repeat the experiments conducted in previous sections, we list all specifications of the systems that were used. The larger specification are preceded by a short description of their behaviour or a reference to literature where the description can be found.

A.1 Diamond Structure

```
map N : Int;
eqn N = 9;  % diamond structure of 10 states wide

act report : Int;


proc X(i,j: Int) =
 (i+j < N) -> (tau. X(i+1,j) + tau. X(i,j+1)) <> delta
+
 (i+j >= N && i + j < 2*N+1) ->
   (
    (i + j == N ) -> report (j). X(i,j)
  +
    (i +j >= N && j < N) -> tau. X(i, j+1)
  +
    (i +j >= N && i < N) -> tau. X(i+1,j)
   ) <> delta;

init X(0,0);
```

A.2 Back-loops

```
% A system with a single trace to the end with backpointers to
% the root at all nodes in between.

map N : Int;
eqn N = 9;    % N+1 is the deadlocking node

act report:Int;

proc P(n : Int)
=
  (n < N) -> tau. P(n+1)
+ (n == N) -> tau. delta
+ (n > 0 && n <= N) -> Q(n-1,n-1)
+ (n == 0) -> tau. P(n);


proc Q(n: Int, m: Int)
=
```

```
  (n == 0) -> tau. P(0)
+ (n >= 1) -> tau. Q(n -1,m);


init P(0);
```

A.3 Strongly Connected Components

```
map K: Pos;
eqn K = 5;

map scc_num, scc_size: Pos;
eqn scc_num = K;
    scc_size = K;

act ini, scc, report: Pos;

proc P = sum id: Pos. (id <= scc_num) ->
                        ini(id).SCC(id,1,scc_size);

     SCC(id: Pos, pos: Pos, size: Pos) =
         sum new_pos: Pos.
           (new_pos <= size && new_pos != pos) ->
             scc(id).SCC(id,new_pos,size)
       + (pos == size) -> report(id).D(id);

     D(id: Pos) = delta;

init P;
```

A.4 Network of Buffers

```
% A small network protocol in which three buffers push a total
% of N messages through the network; each buffer has its own
% propagation time and an associated channel delay. The total
% time it takes to communicate N messages is memorised and
% reported.
% Description: Tom Engels

map N : Nat;
eqn N = 100;

sort TimedOffer = struct offer(t1: Nat, t2: Nat, t3: Nat);
sort Properties = struct prop(pt: Nat, dl: Nat);

map CHECK_PARAM: TimedOffer # Nat # Nat -> Bool;
var offer: TimedOffer;
    position, value: Nat;
eqn (position == 1) -> CHECK_PARAM(offer,position,value) =
```

```
                        (t1(offer) == value);
    (position == 2) -> CHECK_PARAM(offer,position,value) =
                        (t2(offer) == value);
    (position == 3) -> CHECK_PARAM(offer,position,value) =
                        (t3(offer) == value);

map MIN: TimedOffer -> Nat;
var offer: TimedOffer;
eqn MIN(offer) = min(min(t1(offer),t2(offer)),t3(offer));

map EXISTS_SMALLER: TimedOffer # Nat -> Bool;
var offer: TimedOffer;
    now: Nat;
eqn EXISTS_SMALLER(offer,now) =
    ((t1(offer) <= now) || (t2(offer) <= now) ||
    (t3(offer) <= now));

act send, receive, transfer: Nat # Nat # Nat;   % start time,
channel delay, finish time
    s_offer, r_offer, offer: TimedOffer;
    report: Nat;

init allow({transfer, offer, report},
  comm({send | receive -> transfer, s_offer | s_offer |
  s_offer | r_offer -> offer},
    X(1,0,prop(5,5)) || X(2,0,prop(2,10)) || X(3,0,prop(10,2))
    || Y(0,5,N)
  )
);

proc

  X(i, now: Nat, p: Properties) =
    sum offer: TimedOffer . (CHECK_PARAM(offer,i,now)) ->
      s_offer(offer) . (
        X(i, now, p) + sum t:Nat . send(now,dl(p),t) .
                                   X(i, t+pt(p), p));

  Y(now, pt, todo: Nat) =
    (todo > 0) -> (   %receive value from an X
    sum offer: TimedOffer .
      r_offer(offer) . (
        EXISTS_SMALLER(offer,now) -> (
          sum t,d:Nat. (t <= now) ->
              receive(t, d, now+d) .
              Y(now+d+pt, pt, Int2Nat(todo-1))
        )
        <> (
          sum t,d:Nat. (t == MIN(offer)) ->
            receive(t, d, t+d) . Y(t+d+pt, pt, Int2Nat(todo-1))
```

```
        )
      )
    ) <> ( report(now).delta );
```

A.5 Sliding Window Protocol

```
% This file describes the third sliding window protocol in
% Computer Networks by A.S. Tanenbaum, Prentice Hall, 1988. In
% earlier editions of this book (up till the fourth edition) the
% most complex sliding window protocol (i.e. the protocol using
% selective repeat, section 4.4.3) exhibits a nasty livelock.
% This livelock occurs when the acknowledgement timer is switched
% off, after a first acknowledgement is lost in transmission.
% The comment in the code says in Tanenbaum's enjoyable writing
% style "no need for separate ack frame". If the protocol is used
% unidirectionally, i.e., when no piggy backing can be used, no
% acknowledgements are sent from this point, unless new data is
% transmitted and received. In the rare case that the timer is
% switched off when all data in the sender has been received by
% the receiving entity, retransmitted data by the sender is
% ignored. Therefore, no new data is received at the sender, and
% the acknowledgement timer is not switched on again. The result
% is that data is continuously being retransmitted from sender to
% receiver and never acknowledged.
%   Below the bug occurs when the action "error" can occur.
%   Actually, as the problem takes place when the sender's buffer
% is full, the receiver received all data the sender sent, and
% the timer is switched off, the error is a communication of
% three actions, error1, error2 and error3.  For larger values of
% MaxBuf, it is interesting to see how effective bughunting
% strategies are. For MaxBuf=1, the problem occurs after 9
% interactions, for MaxBuf=2 the problem occurs after 17 actions,
% etc.
%
% Jan Friso Groote, February 2007.  Translation is revised in May
% 2007, by removing an error in the translation and by making
% channels and lists finite, such that a finite state space can
% be generated.

sort CS=struct accept | deliver | error | lost;


% Data packets

sort DP = struct d1 | d2; % restricted to two data packets,
                          % for simulation.


% The constants MaxSeq en MaxBuf
```

```
map MaxSeq,MaxBuf,ChannelBufferSize : Pos;

eqn MaxBuf = 2;  % An arbitrary constant
    MaxSeq = MaxBuf + MaxBuf;
    ChannelBufferSize=1;


% The inWindow condition

map inWindow:Nat#Nat#Nat -> Bool;
    seq:Nat#Nat#Nat -> Bool;
    wrap:Nat#Nat#Nat -> Bool;

var fr,low,upp:Nat;
eqn inWindow(fr,low,upp) = seq(fr,low,upp) ||
                           wrap(fr,low,upp);
    seq(fr,low,upp) = upp>low && fr>=low && fr<upp;
    wrap(fr,low,upp) = upp<low && (fr>=low || fr<upp);

% The data type for buffers to store data in protocol
% entities that still have to be sent or received.

sort Buffer=List(struct item(packet:DP,Key:Nat));

map  in_table:Nat#Buffer->Bool;
     retr:Nat#Buffer->DP;
     del:Nat#Buffer->Buffer;
     insert:DP#Nat#Buffer->Buffer;
var  k,k':Nat;
     p,p':DP;
     b:Buffer;
eqn  b==[] -> in_table(k,b) = false;
     in_table(k,item(p,k')|>b) = (k==k')||in_table(k,b);
     b==[] -> del(k,b) = [];
     del(k,item(p,k')|>b) =
       if(k==k',del(k,b),item(p,k')|>del(k,b));
     retr(k,item(p,k')|>b) = if(k==k',p,retr(k,b));
     b==[] -> insert(p,k,b) = [item(p,k)];
     insert(p,k,item(p',k')|>b) =
       if(k<k',item(p,k)|>item(p',k')|>b,
               if(k==k',item(p,k)|>b,
                        item(p',k')|>insert(p,k,b)));

% TTAB contains a list of indices for which a timer is set

sort TTAB=List(Nat);
map  in_table:Nat#TTAB->Bool;
     del:Nat#TTAB->TTAB;
     insert:Nat#TTAB->TTAB;
```

```
var  n,n':Nat;
     t:TTAB;
eqn  t==[] -> in_table(n,t) = false;
     in_table(n,n'|>t) = (n==n')|| in_table(n,t);
     t==[] -> del(n,t) = [];
     del(n,n'|>t) = if(n==n',del(n,t),n'|>del(n,t));
     t==[] -> insert(n,t) = [n];
     insert(n,n'|>t) =
       if(n<n',n|>n'|>t,if(n==n',n|>t,n'|>insert(n,t)));

% The data type for Frames

sort Fid = struct dat | ack | nak;
     Frame = struct ce | defr | dframe(Fid,DP,Nat,Nat) |
                    naframe(Fid,Nat);


% Timer data types

sort Tis = struct start | stop;
     Act = struct act_;


% The description of a channel.

act skip,lost;
    rc1,sc2:Frame;

% This is a channel with limited capacity. If the channel
% is full, the newly entered element replaces the last element
% that was entered into the channel.

proc CH(cq:List(Frame),cs:CS) =
       sum fr:Frame.rc1(fr).((#cq<ChannelBufferSize) ->
           CH(fr|>cq,cs) <> CH(fr|>tail(cq),cs)) +
       (cs==accept && (#cq)!=0) -> (skip.CH(cq,deliver) +
       skip.CH(cq,lost) + skip.CH(cq,error))<>delta+
       (cs==deliver && #cq!=0) ->
             sc2(rhead(cq)).CH(rtail(cq),accept)<>delta+
       (cs==error && #cq!=0) ->
             sc2(ce).CH(rtail(cq),accept)<>delta+
       (cs==lost && #cq!=0) ->
           lost.CH(rtail(cq),accept)<>delta;


% The sliding window protocol entity

act ri1,si2  : DP;
    si3,ri4  : Frame;
    si5  : Tis # Nat;
```

```
    si5  : Tis # Act;
    ri6  : Nat;
    ri6  : Act;

proc IMP = IMP(0,0,0,[],[],false);

     IMP(fts,fta,ftr:Nat,sbuf,rbuf:Buffer,nfs:Bool) =
          ((fta+MaxBuf) mod MaxSeq==fts) ->
            error2(fts).IMP(fts,fta,ftr,sbuf,rbuf,nfs)<>delta+
          error3(ftr).IMP(fts,fta,ftr,sbuf,rbuf,nfs)+
          ((((fts+MaxBuf) mod MaxSeq)!=fta) ->
             ( sum d:DP.ri1(d).
                    si3(dframe(dat,d,fts,(ftr-1) mod MaxSeq)).
                    si5(start,fts).
                    IMP((fts+1) mod MaxSeq,fta,ftr,
                        insert(d,fts,sbuf),rbuf,nfs))<>delta)
               +

          sum e:DP,rfr:Nat,rac:Nat.
               ri4(dframe(dat,e,rfr,rac)).
               ((ftr==rfr)->DEL(fts,fta,ftr,rac,sbuf,
                                insert(e,rfr,rbuf),nfs)
                           <>(nfs -> TORB(fts,fta,ftr,rfr,rac,
                                          e,sbuf,rbuf,true)
                                   <> si3(naframe(nak,
                                            (ftr-1) mod MaxSeq)).
                                      si5(stop,act_).
                                      TORB(fts,fta,ftr,rfr,rac,e,
                                           sbuf,rbuf,true)))+

          sum rac:Nat.
            ri4(naframe(nak,rac)).
            (inWindow((rac+1) mod MaxSeq,fta,fts)
               -> si3(dframe(dat,retr((rac+1) mod MaxSeq,sbuf),
                       (rac+1) mod MaxSeq,(ftr-1) mod MaxSeq)).
                  si5(start,(rac+1) mod MaxSeq).
                  ACK(fts,fta,ftr,rac,sbuf,rbuf,nfs)
               <> ACK(fts,fta,ftr,rac,sbuf,rbuf,nfs)) +

          sum rac:Nat.ri4(naframe(ack,rac)).
                      ACK(fts,fta,ftr,rac,sbuf,rbuf,nfs)
          +
          ri4(ce).(nfs-> IMP(fts,fta,ftr,sbuf,rbuf,true)
                      <> si3(naframe(nak,(ftr-1) mod MaxSeq)).
                         si5(stop,act_).
                         IMP(fts,fta,ftr,sbuf,rbuf,true))+

          sum tfs:Nat.ri6(tfs).
                      si3(dframe(dat,retr(tfs,sbuf),tfs,
                                 (ftr-1) mod MaxSeq)).
```

```
                    si5(start,tfs).
                    IMP(fts,fta,ftr,sbuf,rbuf,nfs) +
          ri6(act_).si3(naframe(ack,(ftr-1) mod MaxSeq)).
          IMP(fts,fta,ftr,sbuf,rbuf,nfs) ;

    DEL(fts,fta,ftr,rac:Nat,sbuf,rbuf:Buffer,nfs:Bool) =
     in_table(ftr,rbuf) -> si2(retr(ftr,rbuf)).
                           DEL(fts,fta,(ftr+1) mod MaxSeq,rac,
                               sbuf,del(ftr,rbuf),false)
                         <> si5(start,act_).
                           ACK(fts,fta,ftr,rac,sbuf,rbuf,nfs);

    TORB(fts,fta,ftr,rfr,rac:Nat,e:DP,
         sbuf,rbuf:Buffer,nfs:Bool)
    =
      (inWindow(rfr,ftr,(ftr+MaxBuf) mod MaxSeq) &&
       !in_table(rfr,rbuf))
          -> ACK(fts,fta,ftr,rac,sbuf,insert(e,rfr,rbuf),nfs)
          <> ACK(fts,fta,ftr,rac,sbuf,rbuf,nfs);

    ACK(fts,fta,ftr,rac:Nat,sbuf,rbuf:Buffer,nfs:Bool) =
       inWindow(rac,fta,fts) -> si5(stop,fta).
                                ACK(fts,(fta+1) mod MaxSeq,
                                    ftr,rac,sbuf,rbuf,nfs)
                              <> IMP(fts,fta,ftr,sbuf,rbuf,nfs);


% The specification of a timer.

act  rt1 : Tis # Nat;
     rt1 : Tis # Act;
     st2 : Nat;
     st2 : Act;

proc TIM = TIM([],false);
     TIM(ttab:TTAB,atr:Bool) =
           sum fn1:Nat.rt1(start,fn1).
                       TIM(insert(fn1,ttab),false)
           +
           rt1(start,act_).TIM(ttab,true) +
           rt1(stop,act_).TIM(ttab,false) +
           sum fn2:Nat.in_table(fn2,ttab) ->
             (rt1(stop,fn2) +
              st2(fn2)).TIM(del(fn2,ttab),atr)
             <>delta +
           atr -> st2(act_).TIM(ttab,false)
               <> sum n:Nat.error1(n).TIM(ttab,atr);

% Below all the components above are combined into one
process.
```

```
act ra,rd,sb,sc:DP;
    s1,r1,c1:Frame;
    s2,r2,c2:Frame;
    s3,r3,c3:Tis#Nat;
    s3,r3,c3:Tis#Act;
    s4,r4,c4:Nat;
    s4,r4,c4:Act;
    s5,r5,c5:Frame;
    s6,r6,c6:Frame;
    s7,r7,c7:Tis#Nat;
    s7,r7,c7:Tis#Act;
    s8,r8,c8:Nat;
    s8,r8,c8:Act;
    error,error1,error2,error3:Nat;

proc IMPa=block({error3},
          rename({ri1->ra, si2->sb, si3->s1,
                  ri4->r2, si5->s3, ri6->r4},
                 IMP));
     IMPb=block({error2},
          rename({ri1->rd, si2->sc, si3->s6,
                  ri4->r5, si5->s7, ri6->r8},
                 IMP));
     CHab=rename({rc1->r1, sc2->s5},CH([],accept));
     CHba=rename({rc1->r6, sc2->s2},CH([],accept));
     TIMa=block({error1},rename({rt1->r3, st2->s4},TIM));
     TIMb=rename({rt1->r7, st2->s8},TIM);

     SWP3 =
      hide({c1,c2,c3,c4,c5,c6,c7,c8,skip,lost},
      allow({ra,rd,sb,sc,c1,c2,c3,c4,c5,c6,c7,c8,
             skip,lost,error},
      comm({ s1|r1 -> c1,
             s2|r2 -> c2,
             s3|r3 -> c3,
             s4|r4 -> c4,
             s5|r5 -> c5,
             s6|r6 -> c6,
             s7|r7 -> c7,
             s8|r8 -> c8,
             error1|error2|error3 -> error},
                IMPa || TIMa || CHab ||
                IMPb || TIMb || CHba)));

init SWP3;
```

A.6 Automated Parking Garage

The specification of the automated parking garage can be found in [5]; the entire specification is available online via `http://www.mcrl2.org/`. It is therefore omitted from this report.

A.7 Dining Philosophers

```
act get,put,up,down,lock,free: Pos#Pos;
    eat: Pos;

proc
  Phil(n:Pos) = get(n,n).get(n,if(n==17,1,n+1)).eat(n).
                put(n,n).put(n,if(n==17,1,n+1)).Phil(n);
  Fork(n:Pos) = sum m:Pos.up(m,n).down(m,n).Fork(n);

  ForkPhil(n:Pos) = Fork(n) || Phil(n);


init allow( { lock, free, eat },
       comm( { get|up->lock, put|down->free },
         ForkPhil(1) ||
         ForkPhil(2) ||
         ForkPhil(3) ||
         ForkPhil(4) ||
         ForkPhil(5) ||
         ForkPhil(6) ||
         ForkPhil(7) ||
         ForkPhil(8) ||
         ForkPhil(9) ||
         ForkPhil(10) ||
         ForkPhil(11) ||
         ForkPhil(12) ||
         ForkPhil(13) ||
         ForkPhil(14) ||
         ForkPhil(15) ||
         ForkPhil(16) ||
         ForkPhil(17)
     ));
```