Model Checking and Random Competition – A Study Using the Model Checking Framework MIC

Alexander K. Wißpeintner, Franz Huber and Jan Philipps

Institut für Informatik, Technische Universität München, 80290 München

E-mail: {wisspein | huberf | philipps}@in.tum.de

1 Introduction

Using verification techniques to prove the correctness of systems is becoming more and more important. Especially in the domain of embedded and distributed systems, verification techniques are increasingly being used for quality assurance today. *Model checking* is a verification technique that allows the automatic verification of systems with a finite state space. Developers create a formal specification of the system and define properties that need to be satisfied by the system description. The model checking program then uses the state transition graph of the system to prove whether the specification fulfills the properties.

Two different approaches to model checking are used in practice today. The *symbolic model checking* [3] variant stores the state transition graph within a *binary decision diagram* (BDD). Symbolic methods are used to prove the validity of the required properties. The most popular symbolic model checking program is SMV [9].

The second model checking variant is called *on-the-fly model checking* [6]. This technique explicitly builds up the state transition graph and searches for states that do not satisfy the specified properties. The possibly large number of states of a system can lead to very long run times and very high storage requirements. To cope with these problems, several efficient algorithms are used.

Parallel search algorithms [15] are used to reduce the run time of the model checking process. To reduce the storage requirements of on-the-fly model checking, techniques like *state compaction* [7, pages 283–284], *state caching* [14] and *bit-state hashing* [7, page 284] are used. Both storage requirements and run time can be reduced by partial search techniques like *partial order reduction* [7, pages 282–823]. A popular on-the-fly model checking program is SPIN [7].

MODELS IN CHECK (MIC) [16] is a framework for the realization of on-the-fly model checking programs. It has been developed in the research group for Software & Systems Engineering (Prof. Broy) at the Technische Universität München. MIC is an experimental platform to study different algorithms and strategies of the model checking process. To speed up the model checking process it is obviously desirable to use parallel search algorithms. A very simple approach to parallel computation is *random competition* [4]. We have used the MIC framework to study the effects of random competition for model checking.

This paper introduces the MIC framework. In section 2 we show the architecture of MIC. To study the model checking algorithms, example system specifications are needed. We describe one example, a central locking system, in more detail in section 3. In section 4 the results of the study about random competition for model checking are shown.

2 The MIC Framework

The MODELS IN CHECK (MIC) [16] framework is an object-oriented programming library. It is implemented in the programming language JAVA. MIC can be used to rapidly realize own model checking programs by using prefabricated components. The framework is intended to serve as an experimental platform. It enables users to study different algorithms and strategies for model checking. The software architecture of MIC splits the model checking task into different subtasks. This way, users can exchange a single algorithm without changing the other parts of the whole system. MIC therefore can also be used to carry out comparative measurements between different algorithms under the same basic conditions.

Users can choose between different search algorithms, state codings and strategies for identifying already inspected system states. The current version of MIC can verify system specifications written in the synchronous language ESTEREL [2]. The framework supports the composition of different ESTEREL modules in a hierarchical structure. Properties can be declared using predicates over the system state and the system outputs. The search algorithms *Breadth-First Search* (BFS), *Depth-First Search* (DFS) and *Depth-First Iterative Deepening* (DFID) [10] are available in the framework. DFID is a variant of the DFS algorithm where the search depth is limited. The limit allows to find short execution paths to incorrect system states. The search is performed in iterations with increasing search depth limits. Due to the iterative search the run times of DFID are longer compared to DFS.

Furthermore, MIC offers two different state coding strategies. States can be stored without any compression as a single system state vector. The second state coding strategy is similar to the *state compaction method* of SPIN [7, pages 283–284]. The system state vector consists of references to the component state vectors. By storing a component state vector only once for several system states, memory usage can be decreased. The framework stores inspected states in the main memory. The stored states are used to identify already visited states using a hashing technique.

Figure 1 on the following page shows the software architecture of MIC. The `Main Controller` component realizes the check cycle of the model checking process. It controls the collaboration between the different subsystems of the framework. All communications between the subsystems take place via the `Main Controller` component. The `System` component acts as an adaptor between the system description and the other parts of MIC. This component offers functionality to execute the system description provided by the user. Furthermore, getting and setting the system state are elementary tasks of the `System` com-

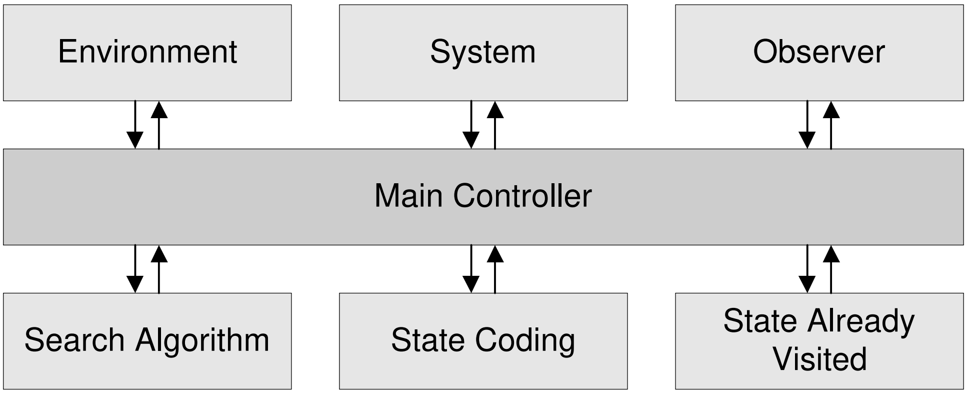

Figure 1: The MIC Architecture—Data Flow Diagram

ponent. This part of MIC has to be adapted to add support for a new system description language.

The `Observer` component monitors the correctness of the specified properties during the model checking process. It realizes an adaptor between the properties specified by the user and the MIC system. The `Environment` produces system inputs from the system environment. By choosing a specific `Search Algorithm` subsystem, the search algorithm to be used for model checking is fixed. The components `State Coding` and `State Already Visited` specify the kind of state coding algorithm and the strategy to identify already inspected states.

An additional statistics component can be used to record measured values during single model checking runs. The run time, the number of check cycles, the number of found system states and the current search depth are written to a log file. Furthermore, the memory usage of the search data structure and the table for identifying already inspected states are logged during the model checking run.

Using the existing components, a complete model checking program can be realized within only a few lines of code. The only task to perform is to choose between the alternative algorithms provided by MIC and register them at the `Main Controller` component. The architecture of MIC supports extensibility. Users can easily extend the framework by implementing new algorithms. Even support for further system description languages and property languages can be realized. Every MIC component has a well-defined interface. This interface has to be implemented to realize a new variant of the component.

3 Example System Specifications

We need different system specifications to make quantitative measurements and compare different algorithms and strategies. For that purpose, we used three example systems. A traffic light control system [11] was checked by MIC. The traffic light controller on purpose contained a specification error resulting in unfairness. Furthermore, we verified a communication protocol realizing mutual exclusion (mutex) [13]. Here, a specification error broke the rule of exclusive access. The third example was a central locking control system, which we now describe in more detail.

The central locking system is adopted from an example used in [12]. This system was originally specified using *Statecharts* [5]. We transferred the system into the *Syncccharts* [1] description technique. Syncccharts are very similar to Statecharts and allow the automatic translation into an ESTEREL program. The system controls two door locks of a car. The doors can be locked or unlocked by turning a key. Furthermore, the car has a crash sensor. If the car has an accident, the central locking system must immediately unlock all doors.

Figure 2 on the next page shows the Synccchart defining the normal operation of the system. The signal `open` represents the user interaction of turning the key to unlock the doors. It is used to branch into the sub-Synccchart `Unlocking` to perform the unlocking operation. A `close` signal is used to enter the `Locking` Synccchart. Figure 3 on the following page shows the `Unlocking` Synccchart. When entering this sub-Synccchart, the signals `left_up` and `right_up` are sent to both door locks to open them. The control system waits until both door locks acknowledge the execution of the unlocking operation by the signals `left_ready` and `right_ready`. The sub-Synccchart `Locking` is analogous to `Unlocking`.

To implement the requirement to unlock the doors during a crash, another Synccchart is used (Figure 4). Whenever a crash is detected and the ignition of the car is turned on, the central locking system is deactivated by branching into the `Crash` state. The signals `left_up` and `right_up` are sent to directly unlock the doors. The Synccchart specifications of the door locks are not shown in this paper.

To verify the central locking system, we define a temporal safety property. If a crash is detected while the ignition of the car is on, the doors must be unlocked after five clock cycles. There is a specification error hidden in this system description. If an accident happens while the driver locks the doors, the doors cannot react on the signals to unlock the doors. The doors stay locked.

4 Using Random Competition

Random competition [4] is a simple principle of parallel computation that requires little communication between processors. The calculations are done simultaneously on different processing units, each of them using a different calculation method. One of the processors is able to solve the problem first. This processor wins the competition and reports the result.

To adopt the principle of random competition for model checking, we use random variants of the DFS and DFID search algorithms. On each processor where the model checking process is started, the system states are explored in a different order. In case of a system specification that satisfies the defined properties, this method will not achieve speed-ups worth mentioning. Each processor has to search the whole state space of the system. But in case of an incorrect system description one processor will find the incorrect system state first. In this way, using several processors can speed up the model checking process.

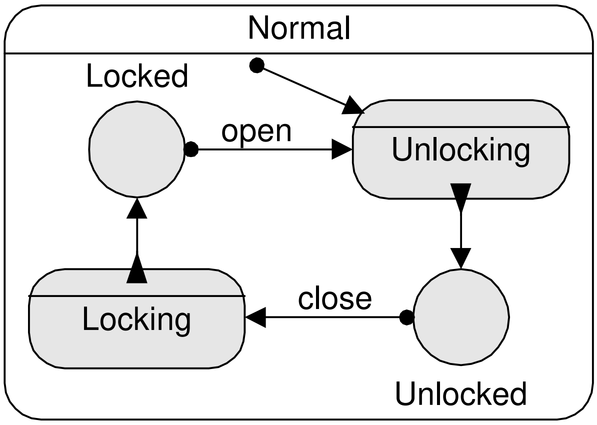

Figure 2: Syncchart Normal Operation—Central Locking System

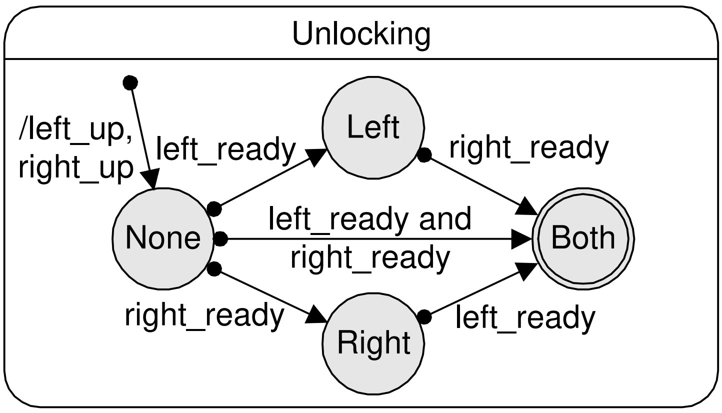

Figure 3: Syncchart Unlocking the Doors—Central Locking System

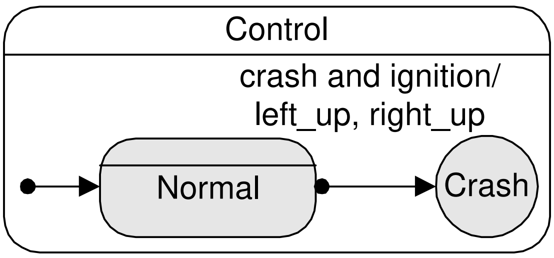

Figure 4: Syncchart Crash Detection—Central Locking System

We have used the MIC framework to study the effects of random competition for model checking. Incorrect versions of the example systems described in Section 3 were verified by a model checking program using the random search algorithms. The aim was to determine the expected speed-up when using multiple processors for model checking in contrast to one processor. The speed-up S is often defined as follows.

$$S = \frac{\textit{sequential run time } T_{seq}}{\textit{parallel run time } T_{par}} \tag{1}$$

The number of check cycles needed to find a specification error is almost proportional to the run time. To avoid inaccuracies in the run time measurements, we measure the number of check cycles instead. Our definition of the speed-up for k processing units S_k in terms of check cycles is given below. C_1 is the number of check cycles needed to find the error when using a single processor. C_k stands for the number of check cycles when using k processors.

$$S_k = \frac{C_1}{C_k} \tag{2}$$

The number C_k is the number of check cycles needed by the fastest of the k processors. It is the minimum of the check cycles c_1 to c_k that are measured for the k competing processors.

$$C_k = \min\{c_1, c_2, ..., c_k\} \tag{3}$$

To eliminate random effects caused by the random search algorithms, the expected values of C_1 and C_k are used.

A characteristic of random competition is that the processing units do not need to communicate with each other during the working phase. The calculations done by the participating processors are independent of each other. This is why it is only necessary to measure single independent model checking runs. We can calculate the expected values of check cycles for k processors from the results measured on one processor. Therefore, we need the probability distribution $f_{C_k}(i)$. This discrete distribution defines the probability that the fastest of the k processors needs i check cycles to find the specification error. Details on calculating $f_{C_k}(i)$ from the measured values are given in [16, pages 106–109].

With the known $f_{C_k}(i)$ distribution we can calculate to the speed-up factor using the expected values.

$$S_k = \frac{E(f_{C_1}(i))}{E(f_{C_k}(i))} \tag{4}$$

A large number of measurements are needed to get stable expected values for the speed-up calculation. To calculate the expected speed-up using random DFS for the central locking and traffic lights systems we carried out 10,000 single runs each. For the simpler mutex example, 1,000 measured runs are sufficient to calculate the expected speed-up.

Figure 5 on the next page shows the histogram of the $f_{C_k}(i)$ distribution of the central locking system. The values for one to 20 processors are shown using a bucket interval of 5,000 check cycles. The probability distribution for one processor is the direct result of the measurements. The values for several processors have been calculated using the distribution for a single processor. A larger number of processors leads to a higher probability of finding the specification error in less time.

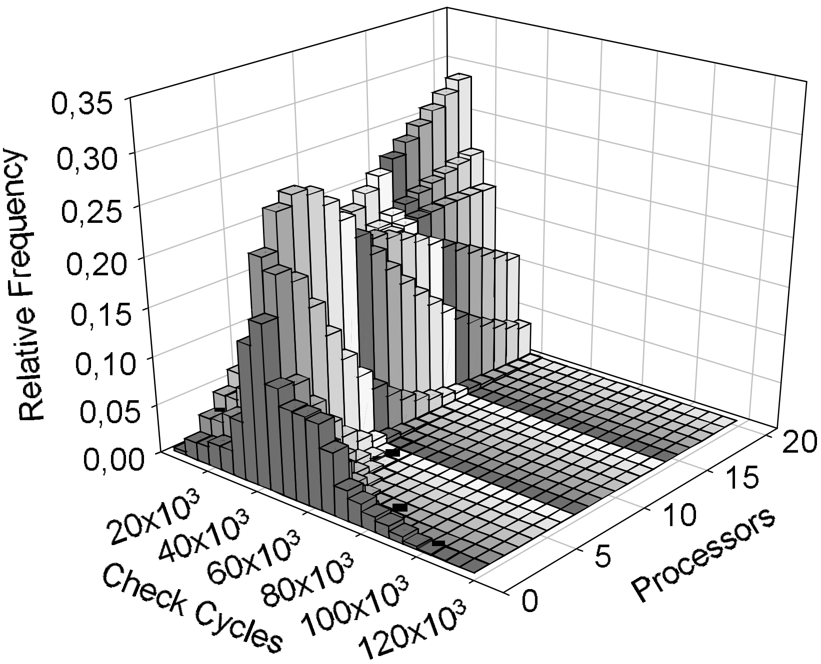

Figure 5: Histogram $f_{C_k}(i)$ Distribution—Random DFS—Central Locking System

The expected speed-ups using the random DFS algorithm on multiple processors are shown in Figure 6 on the following page. The run time for the verification of the central locking and mutex examples can be greatly decreased using the principle of random competition. Using 10 processors leads to a speed-up factor of 2.2 for the central locking system. The speed-up can be increased further by using more processing units. A speed-up factor of 6.6 is achieved by using 100 processors for verifying the central locking system. The mutex example achieves a speed-up of 5.1 with 10 processors. Due to the small size of this system specification, the speed-up factor can not be improved significantly by using more processors. The result of the traffic lights system using random DFS, however, is less promising. The speed-up factors of 1.7 with 10 and 2.5 with 100 processors are quite low.

The studies show substantial differences in the efficiency of random competition. General statements about the suitability of a system for model checking using random competition are difficult to find. The speed-ups that can be achieved strongly depend on the structure of the state space of the system to be verified. Good results can be achieved if the specification error is not obvious. If the error can only be found via a few long execution paths, then the expected value for the run time on one processor will be large. Nevertheless, using random DFS it is possible to find a solution in a short time. A large variance of the run times of the model checking process leads to a good speed-up when using random competition.

Besides the random DFS algorithms we have used the random DFID algorithm for parallel model checking. Figure 7 on the next page shows the speed-up for the central locking system calculated from 500 single model checking runs with the DFID search algorithm and a constant search depth increment of 10. Using 100 processors for model checking leads to a speed-up of 1.6. This is very low in comparison to the speed-up factor 6.6 when using the DFS algorithm to verify the same system description. There are two reasons for this effect.

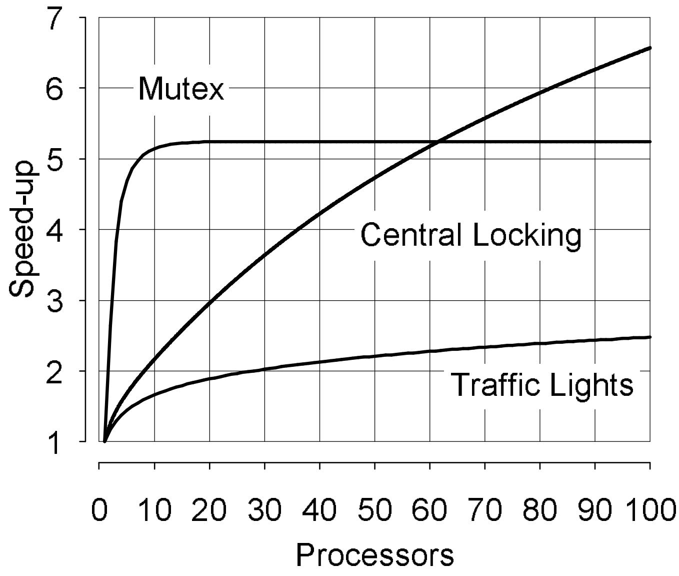

Figure 6: Speed-ups Using Random DFS

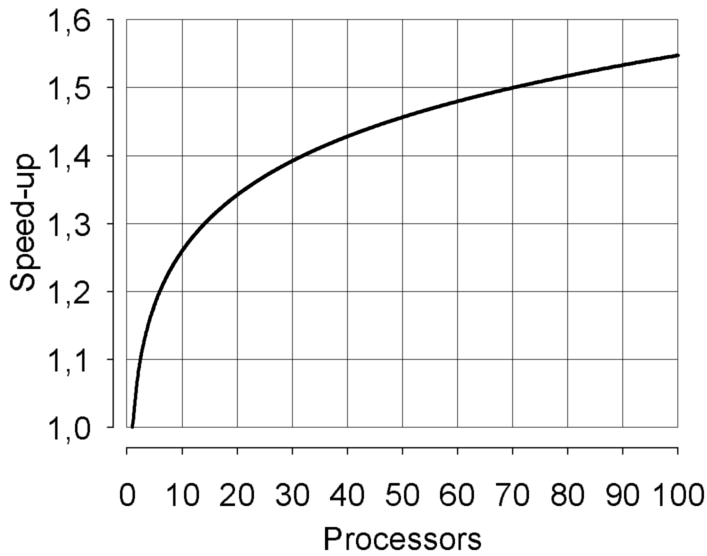

Figure 7: Speed-ups Using Random DFID—Central Locking System

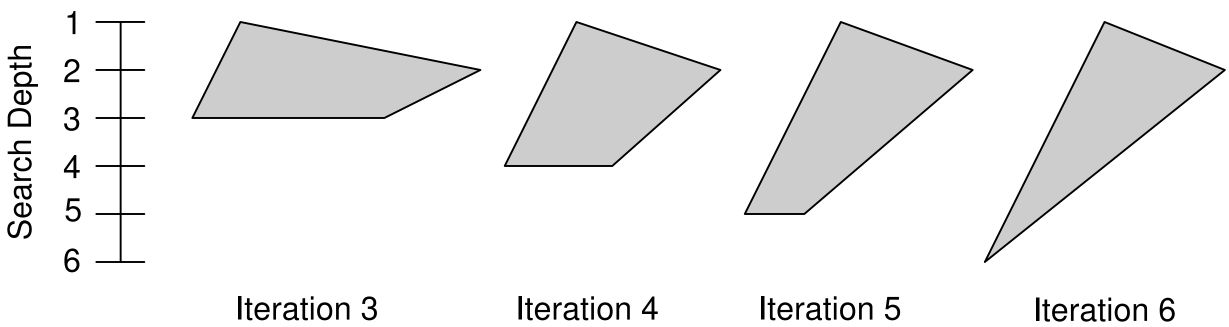

Figure 8: Search Trees of the DFID Algorithm during the Single Iterations

The specification error is found in the n-th iteration of the DFID algorithm. Therefore all processors must pass trough $n-1$ search iterations before the n-th iteration is reached. The run times for the $n-1$ iterations are almost equal for all processors. In these iterations each processor searches all states that are reached within the search depth limit. Only the n-th iteration of the search process is decisive for the achieved speed-up. The fixed run time for $n-1$ iterations therefore decreases the speed-up.

The second reason for the bad results is determined by the structure of common software systems. In most software systems states can be reached over short and long execution paths. Without any search depth limits DFS strategies tend to find new states in deep search depths. By limiting the search depth the system states are found via short search paths causing the search algorithm to span a less deeper but broader tree. Due to the search depth limit not all system states can be found. But even with low search depth limits very many states are reached by the search algorithm creating a broad search tree. By increasing the search depth limit the search tree gets more deeper and less broader. In the course of this the number of found states grows very slow. This effect leads to a non-exponential growth of the run time of the DFID iterations with increasing search depth limits. The run time of a DFID iteration is only a bit longer than the time of the previous iteration. This effect, together with the fact of the fixed run time for $n-1$ iterations, leads to very low achievable speed-ups. Figure 8 on the preceding page clarifies the described effect by an example.

5 Conclusion

In this paper a framework for on-the-fly model checking was introduced. The framework allows the rapid development of a model checker that can be used to study strategies and algorithms of the model checking verification technique. The MIC framework is focused on extensibility to allow the implementation of new algorithms for model checking. It is possible to perform comparative measurements between different algorithms during the model checking process.

We have shown the suitability of MIC in a study about using random competition for model checking. This simple principle of parallel computation can highly decrease the run time for model checking in case of specification errors. The excepted speed-up using random competition heavily depends on the system to be verified. Nevertheless system descriptions including complex specification errors are good candidates for the usage of random competition.

The random search algorithms DFS and DFID have been used to implement the approach of random competition. The results of the random DFID algorithm are less promising due to fundamental problems with regard to parallel computation. With the random DFS algorithm good speed-ups can be achieved. This search algorithm is suitable for model checking with random competition.

In the future, the MIC framework will support system descriptions created in the CASE tool AUTOFOCUS [8]. Furthermore, one of the next releases of MIC will support *temporal logic* to define system properties. Our future work will be focused on studying additional approaches to parallel model checking.

References

[1] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical report, Laboratoire I3S, UNSA/CNRS, Nice, France, 1995.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] Wolfgang Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Influenzsystemen*. PhD thesis, Technische Universität München, 1992.

[5] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] Gerard J. Holzmann. *Design and Validation of Computer Protocols*, chapter 11–14, pages 214–350. Prentice Hall, 1991.

[7] Gerard J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[8] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In *FTRTFT'96, LNCS 1135*, pages 467–470. Springer, 1996.

[9] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[10] Pearl and Korf. Search techniques. *ANNREVCS: Annual Review of Computer Science*, 2, 1987.

[11] Jan Philipps and Alexander Schmidt. Traffic flow by data flow. Technical Report TUM-I9718, Institut für Informatik, Technische Universität München, 1997.

[12] Jan Philipps and Peter Scholz. Formal verification of statecharts with instantaneous chain reactions. In *TACAS'97, LNCS 1217*, pages 224–238. Springer, 1997.

[13] C. Puchol. *The TempEst Program Verfication Toolkit – Example Slides*. AT&T Bell Laboratories, 1997.

[14] U. Stern and D.L. Dill. Combining state space caching and hash compaction. In Bernd Straube and Jens Schoenherr, editors, *4. GI/ITG/GME Workshop zur Methoden des Entwurfs und der Verifikation Digitaler Systeme*, Berichte aus der Informatik, pages 81–90, Kreischa, March 1996. GI/ITG/GME, Shaker Verlag, Aachen.

[15] U. Stern and D.L. Dill. Parallelizing the murφ verifier. In *9th International Conference on Computer Aided Verification*, 1997.

[16] Alexander K. Wißpeintner. Model-Checking Strategien mit MIC. Master's thesis, Technische Universität München, November 1999.