# PARALLEL CONSISTENCY CHECKING OF AUTOMOTIVE PRODUCT DATA

WOLFGANG BLOCHINGER, CARSTEN SINZ AND WOLFGANG KÜCHLIN

*Symbolic Computation Group, WSI for Computer Science,*
*Universität Tübingen, 72076 Tübingen, Germany*
*http://www-sr.informatik.uni-tuebingen.de*

This paper deals with a parallel approach to the verification of consistency aspects of an industrial product configuration data base. The data base we analyze is used by DaimlerChrysler to check the orders for cars and commercial vehicles of their Mercedes lines. By formalizing the ordering process and employing techniques from symbolic computation we could establish a set of tools that allow the automatic execution of huge series of consistency checks, thereby ultimately enhancing the quality of the product data. However, occasional occurrences of computation intensive checks are a limiting factor for the usability of the tools. Therefore, a prototypical parallel re-implementation using our Distributed Object-Oriented Threads System (DOTS) was carried out. Performance measurements on a heterogeneous cluster of shared-memory multiprocessor Unix workstations and standard Windows PCs revealed considerable speed-ups and substantially reduced the average waiting time for individual checks. We thus arrive at a noticeable improvement in usability of the consistency checking tools.

## 1 Introduction

Today's automotive industry manages to supply customers with highly individualized products by configuring each vehicle individually from a very large set of possible options. E.g., the Mercedes C-class of passenger cars allows far more than a thousand options, and on the average more than 30,000 cars will be manufactured before an order is repeated identically. Heavy commercial trucks are even more individualized, and every truck configuration is built only very few times on average. The space of possible variations is so great that the validity of each order needs to be checked electronically against a product data base which encodes the constraints governing legal combinations of options[1]. But the maintenance of a data base with thousands of logical rules is error-prone, especially since it is under constant change due to the phasing in and out of models. Every fault in the data base may lead to a valid order rejected, or an invalid (non constructible) order accepted which may ultimately result in the assembly line to be stopped. Therefore, reaching correctness of the product data base is a high priority goal.

DaimlerChrysler employs the electronic product data management (EPDM) system DIALOG for the configuration of their Mercedes lines. Within this system, a customer's order consists of a basic model class selection together with a set of further equipment codes describing additional features. Each equipment code is represented by a Boolean variable, and choosing some piece of equipment is reflected by setting the corresponding variable to *true*. An order is processed in three major steps, as depicted in Figure 1. All of these steps are controlled by logical rules of the EPDM system:

1. *Order completion:* Supplement the customer's order by additional (implied) codes.
2. *Constructibility check:* Are all constraints on constructible models fulfilled by this order?
3. *Parts list generation:* Transform the (possibly supplemented) order into a list of parts (a bill of materials).

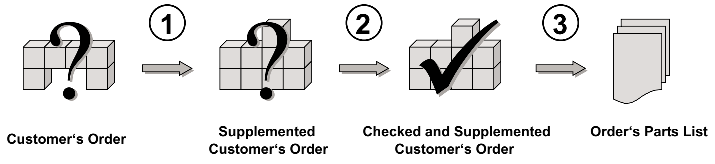
1
2
3
Customer's Order
Supplemented Customer's Order
Checked and Supplemented Customer's Order
Order's Parts List

Figure 1. Processing a customer's order.

In order to systematically detect defects in the rule system, we developed a formal model of the ordering process. Thus, we are able to apply an automatic theorem prover to check certain consistency criteria of the rule base as a whole:

**Necessary and inadmissible codes:** Are there codes which must invariably appear in each constructible order? Are there codes which cannot possibly appear in any constructible order?

**Superfluous parts:** Are there parts which cannot occur in any constructible order?

Both criteria can be formulated as propositional logic satisfiability (SAT) problems[2], and our interactive consistency support tool BIS[3] contains an implementation of a Davis-Putnam-style[4] propositional prover to verify them.

To completely check the above mentioned criteria for only one model class, up to 10,000 prover runs have to be performed. Most of these automatic proofs are completed in a few seconds, but there remains a small fraction that requires comparatively high run-times of up to several hours. Unfortunately, there is no known method to estimate the run-times in advance, and in an interactive system like BIS, users hardly accept long and unpredictable waiting times. It may happen that one of the first in a sequence of proofs requires a very long run-time, which causes a delay in the presentation of the results of all the other proofs: The user does not get any result until the first proof is completed.

Our parallelization approach therefore is two-fold: we execute a set of proofs in parallel, and if we hit a long-running proof we additionally process this individual proof in parallel. Thus, the average waiting time for the result of an individual proof can be reduced considerably.

Before we describe our parallelization in more detail, we will give an overview of our software infrastructure DOTS used in this approach. We will then explain our method in terms of the DOTS system, before presenting experimental results, related work, and our conclusions.

## 2 Parallelization Infrastructure

As parallelization infrastructure, our Distributed Object-Oriented Threads System DOTS[5] is used. DOTS is a C++ parallel programming toolkit that integrates a wide range of different computing platforms into a single system environment for high performance computing.

### *2.1 The programming paradigm of DOTS*

The main idea of DOTS is to make the threads programming paradigm used on shared-memory machines available in a distributed memory environment. With DOTS, a hierarchical multiprocessor, consisting of a (heterogeneous) cluster of shared-memory multiprocessor systems, can be efficiently programmed using a single paradigm.

The DOTS API provides primitives for DOTS thread creation (dots_fork), synchronization with the results computed by other DOTS threads (dots_join), and DOTS thread cancellation (dots_cancel). All primitives can also be used in conjunction with so called *thread groups*. Thread groups are a means of representing related DOTS threads. When applied with thread groups, the semantics of each primitive is automatically changed to the appropriate group semantics. E.g. when using the join primitive with a thread group, *join-any* semantics will be applied.

### *2.2 The Architecture of DOTS*

Figure 2 gives an overview of the basic components of the architecture of DOTS. When a DOTS thread is created with the dots_fork primitive, a so called *thread object* is instantiated that represents the DOTS thread within the system during the complete execution process. It stores all information that is necessary to execute the DOTS thread.

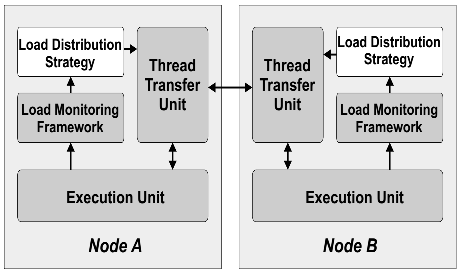
Load Distribution Strategy
Thread Transfer Unit
Load Monitoring Framework
Execution Unit
Node A
Thread Transfer Unit
Load Distribution Strategy
Load Monitoring Framework
Execution Unit
Node B

Figure 2. The DOTS Architecture

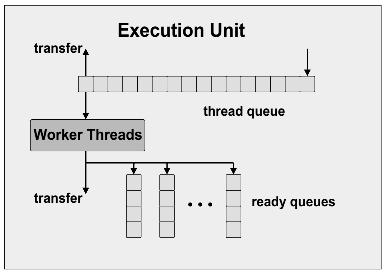

Figure 3. The Execution Unit

DOTS threads are executed within the *Execution Unit* (see Figure 3). It contains a thread queue in which newly created thread objects are enqueued. A pool of (OS native) worker threads dequeue thread objects from the queue and execute the corresponding DOTS threads. The number of worker threads can be determined by the programmer. Normally, for each node the number of available processors is chosen. After the execution of a DOTS thread is completed, its thread object is placed into a ready queue.

To support the execution of DOTS threads in a distributed environment, the DOTS architecture includes additional components. The *Thread Transfer Unit* transfers (serialized) thread objects between queues of execution units residing on different nodes. *The Load Monitoring Framework* traces all events concerning the execution of DOTS threads and provides status information like the current load or the current length of the thread queue. Based on the Load Monitoring Framework, different load distribution strategies can be implemented. A load distribution strategy is responsible for triggering the transfer of thread objects and selecting destination nodes according to a particular strategy.

## 3 Parallelization

The presented parallelization approach pursues two major goals. The first goal is to achieve a total speedup of the computation. A second important goal is to reduce the average waiting time for the result of a proof in order to improve the usability of the application.

In the subsequently described procedure, hard proofs are determined by setting a limit for the computation time for a proof. If the time for a proof has expired, it is considered as a hard proof and treated separately by executing it in parallel. Consequently, the parallel execution is organized in two phases:

- **Phase 1:** Concurrent execution of proofs.

A timeout is set to suspend (long-running) hard proofs. Hard proofs are queued along with their current execution state.

- **Phase 2:** Parallel execution of the queued hard proofs.
  Each queued proof is treated individually in parallel, starting from its previously saved state computed in phase 1.

The execution of phase 1 is organized in a master-slave approach. For all proofs, the root thread creates corresponding DOTS threads that are executed concurrently. The computed results are joined by the root thread and displayed. In the case of a timed out hard proof, its current execution state is joined and queued for later execution in phase 2.

The realization of phase 2 requires more sophisticated techniques. We adopted the parallelization scheme for the Davis-Putnam algorithm presented by Zhang *et al.*[6]. Basically, we are dealing with the parallelization of a combinatorial search problem. This implies that the search space has to be divided into mutually disjoint portions to be treated in parallel. However, a (static) generation of balanced subproblems is not feasible, since it is impossible to predict in advance the extent of the problem reduction delivered by the Davis-Putnam procedure.

Instead, a dynamic search space splitting approach is carried out. To start the execution of a queued hard proof the root thread forks one DOTS thread that has the entire search space assigned. During the whole computation of phase 2, all DOTS threads periodically monitor the length of the local thread queue (see Section 2.2). If the thread queue is empty, a new DOTS thread is forked. The parent thread splits off a region of its search space and assigns it to the new DOTS thread. Details of the applied search space splitting heuristics can be found in Zhang *et al.*[6]. To prevent uncontrolled splitting actions, a predefined time interval has to be waited before the next split can be carried out by the DOTS thread. The newly created DOTS thread is queued and can be executed by another local worker thread or can be transferred to other nodes (see Section 3.1).

The described splitting procedure generates subproblems on demand. This ensures that new subproblems are generated during the initialization phase of the computation to exploit the available processing capacity and every time a subproblem has been completely processed without finding a solution.

After forking the initial DOTS thread, the root thread calls dots_join to wait for the created DOTS threads. All DOTS threads (except the initial one) are created with the dots_subfork primitive. This means that they can be joined by the root thread (and are not joined by their actual parent threads). The result of a DOTS thread indicates whether a problem solution was found within its assigned search region or not. The processing of a hard proof is completed either if all created DOTS threads have been joined without returning a solution, or when the first DOTS thread that has found a solution is joined. In the latter case, all remaining DOTS threads are immediately canceled to make all processing capacities available for the next queued hard proof.

### *3.1 Load Distribution*

As described in Section 2.2, new load distribution strategies can easily be integrated into DOTS using the Load Monitoring Framework. For the presented application, a customized load distribution strategy was realized that reflects the division of the computation in two phases.

In both phases, a work-stealing strategy is applied, i.e. when all worker threads on a node are idle, the distribution strategy tries to transfer thread objects from the thread queues of other nodes.

However, the phases are treated differently in the way how the victim node is chosen. Whereas in phase 1 always the master node is selected as victim (all DOTS threads are created by the master in phase 1), victims are chosen randomly in phase 2. It has been shown that applying a randomized work-stealing strategy to distribute the load in backtrack search algorithms is likely to yield a speedup within a constant factor from optimal (when all solution are required)[7].

Since this approach does not involve central components for load distribution, the scalability of the parallel application is improved.

## 4 Experimental Results

The parallel environment used for the presented performance measurements consisted of a cluster made up of the following components (all nodes were connected with 100 Mbps switched Fast-Ethernet).

- 2 Sun Ultra E450, each with 4 UltraSparcII processors (@400 MHz) and 1 GB of main memory, running under Solaris 7.

- 4 PCs, each with 1 PentiumII processor (@400MHz) and 128 MB of main memory, running under Windows NT 4.0.

We have applied the necessary and inadmissible codes checks as well as the superfluous parts checks on configuration data for the C-class and E-class limousines. Tables 1 and 2 show the measured run-times (wall-clock times) in seconds. A timeout of 20 seconds for detecting hard proof was chosen. The tables give sequential run-times of the prover executed on an E450, on a PC, and the corresponding weighted mean of the sequential run-times. Additionally, run-times of three parallel program runs on all processors are shown. The parallel execution of the prover can exhibit a

Table 1. Results for the Mercedes C-class.

| C-class limousines (code checks) | | |
|---|---|---|
| proofs: 520 | hard proofs: 4 | |
| | run-time | waiting time |
| seq (E450) | 1,417.0 | 654.6 |
| seq (PC) | 1,738.0 | 800.3 |
| seq (mean) | 1,524.0 | 703.2 |
| parallel (3 runs) | 158.0 | 66.8 |
| | 162.0 | 66.6 |
| | 164.0 | 67.1 |

| C-class limousines (part checks) | | |
|---|---|---|
| proofs: 512 | hard proofs: 6 | |
| | run-time | waiting time |
| seq (E450) | 10,447.0 | 5,459.5 |
| seq (PC) | 13,667.0 | 7,167.1 |
| seq (mean) | 11,520.3 | 6,028.7 |
| parallel (3 runs) | 1,026.0 | 100.6 |
| | 1,043.0 | 100.8 |
| | 1,039.0 | 100.8 |

Table 2. Results for the Mercedes E-class.

| E-class limousines (code checks) | | |
|---|---|---|
| proofs: 525 | hard proofs: 6 | |
| | run-time | waiting time |
| seq (E450) | 1,935.0 | 956.1 |
| seq (PC) | 2,281.0 | 1,123.9 |
| seq (mean) | 2,050.3 | 1,012.0 |
| parallel (3 runs) | 253.0 | 111.4 |
| | 245.0 | 110.6 |
| | 252.0 | 111.5 |

| E-class limousines (part checks) | | |
|---|---|---|
| proofs: 500 | hard proofs: 6 | |
| | run-time | waiting time |
| seq (E450) | 40,186.0 | 16,370.5 |
| seq (PC) | 51,769.0 | 21,500.4 |
| seq (mean) | 44,047.0 | 18,080.5 |
| parallel (3 runs) | 2,198.0 | 110.4 |
| | 2,174.0 | 109.3 |
| | 2,192.0 | 108.8 |

non-deterministic behavior, for this reason we don't give averaged times or speedup-values.
The average waiting times were calculated by

$$\sum_{i=1}^{\#proofs} \frac{waiting\_time(i)}{\#proofs} \quad ,$$

where $waiting\_time(i)$ is defined as the time from the start of the whole set of tests until the result of proof $i$ is reported.

Since some of the detected hard problems in the considered examples turned out to be satisfiable, the parallel execution of these proofs can lead to super-linear speedups and may consequently greatly reduce the total run-time as well as the mean waiting time for a proof.

## 5 Related Work and Conclusion

In the realm of industrial verification, Spreeuwenberg *et al.*[8] present a tool to verify knowledge bases built with Computer Associate's Aion[9]. Concerning parallel satisfiability checking, PSATO of Zhang *et al.*[6] is a distributed prover for propositional logic on a network of workstations. In contrast to our work, a master-slave model is applied, where a central master is responsible for the division of the search space and for assigning the subtasks to the slaves.

We see the main contribution of our paper in presenting an industrial real-world application of symbolic computation where employment of parallelization techniques greatly enhances usability by reducing the user's waiting time to an acceptable amount. Sophisticated parallel execution and scheduling methods almost manage to overcome the unpredictability of proof times.

## References

1. E. Freuder. The role of configuration knowledge in the business process. *IEEE Intelligent Systems*, 13(4):29–31, July/August 1998.
2. W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.
3. C. Sinz, A. Kaiser, and W. Küchlin. Detection of inconsistencies in complex product model data using extended propositional SAT-checking. In *FLAIRS'01*, 2001. To appear.
4. M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM*, volume 7, pages 201–215, 1960.
5. Wolfgang Blochinger, Wolfgang Küchlin, Christoph Ludwig, and Andreas Weber. An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.
6. H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.
7. R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.
8. S. Spreeuwenberg, R. Gerrits, and M. Boekenoogen. VALENS: A Knowledge Based Tool to Validate and Verify an Aion Knowledge Base. In *ECAI 2000, 14th European Conference on Artificial Intelligence*, pages 731–735. IOS Press, 2000.
9. S. Garone and N. Buck. *Capturing, Reusing, and Applying Knowledge for Competitive Advantage: Computer Associate's Aion*. International Data Corporation, 2000. IDC White Paper.