Formal Verification in an Industrial Context

Carsten Sinz
Symbolic Computation Group, WSI for Computer Science,
University of Tuebingen, 72076 Tuebingen, Germany
sinz@informatik.uni-tuebingen.de

February 18, 2002

Abstract

We present two case studies employing formal verification in an industrial context. Our first example deals with product configuration for the automotive industry, the second one examines a rule-based expert system controlling IBM's high-availability System Automation software. We identify common requirements to both the logical encoding and the decision procedures for the purpose of verification. Moreover we summarize experiences gained during these projects.

1 Introduction

Formal verification has attracted a lot of attention in the realm of hardware verification (see [4] for a survey). Industrial application outside this area has remained rare, though. In this short report, we want to summarize our experiences collected during two industrial projects employing verification technology. In both cases we experimented with Binary Decision Diagrams (BDDs) and propositional SAT checkers to solve the resulting decision problems.

Although the problems underlying the two studies stem from quite different fields and vary considerably in their characteristics, we could observe a surprisingly high degree of similarity.

2 Case Studies

We now want to give a short introduction to the two applications. Details about each of them can be found elsewhere [5, 8].

2.1 Configuration of DaimlerChrysler's Mercedes Lines

Today's industry manages to supply customers with highly individualized products by personalizing them using a large set of configuration options. E.g., in the automotive industry, the Mercedes C-class of passenger cars allows far more than a thousand

equipment options. 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 [2]. But the maintenance of a data base with thousands of logical rules is error-prone in itself, especially since it is under constant change due to the phasing in and out of models. Therefore, configuration systems are employed which do the validity checking of individual orders. Such systems automatically checks whether or not each order fulfills a set of constraints formulated in Boolean logic. Some of these constraints represent rules about valid combinations of sales options, others modify a customer's order by adding implied options (*order completion*), again others express conditions under which a part is included into the parts list, and therefore make up a transformation from the customer's view of sales options to the engineer's view of parts.

In our study, we checked the configuration data base for integrity, specified by a set of validation criteria that the knowledge base is supposed to fulfill:

1. All sales options can actually occur in a valid order, and no sales option is mandatory.
2. All parts can actually occur in a valid order, and no mutually exclusive parts are simultaneously selected on a product.
3. The order completion process does not invalidate constructible orders and does not depend on the (possibly accidental) order in which options are added.

The data sets we examined consisted of up to 1500 propositional variables, and 10000 rules. For all tests a propositional encoding $\mathcal{C}$ of the set of valid orders was constructed [5]. This set was used as the basic theory for our tests, to which small formulae were added expressing the integrity criteria mentioned above.

By using formula $\mathcal{C}$, we could refrain from expressing explicitly the computation states during order processing, thus avoiding full-blown model checking. We consider this an essential aspect, since we suppose the size of our formulae to exceed the limit of current model checkers.

2.2 IBM System Automation's Expert System

The purpose of IBM's System Automation software [8] is to monitor and control a large set of applications running on a cluster of mainframes (e.g. IBM's zSeries), including error state detection, and moving, starting and stopping of applications. During these actions dependencies between applications have to be considered.

The dependencies and the currently desired state of the applications (*Automation Goal*) is user-specified. Conversion of these goals into basic application control commands, as well as automatic reaction to possible errors, is performed by a rule-based expert system (*Automation Manager*). The expert system consists of a set of approximately one hundred rules (*Correlation Rules*) which compute the application control command to be issued next. Of course this computation should be terminating, so this was our main concern for verification.

The rules consist of situation-action pairs (`WHEN-THEN`) and employ a finite domain logic. An example of such a rule is

```
when     status/compound NOT E {Satisfactory}
     AND ( ( status/observed E {Available, WasAvailable}
               AND status/desired E {Available}
               AND status/automation E {Idle, Internal} )
           OR
           ( status/observed E {SoftDown, StandBy}
             AND status/desired E {Unavailable} )
     )
then SetVariable status/compound = Satisfactory
```

We specified the computations performed by the expert system in an extension of propositional dynamic logic (ΔPDL), and generated instances of the termination property by converting the ΔPDL formula to propositional logic [8]. This conversion then allowed application of SAT checkers and BDDs.

3 Logical Requirements

We now want to summarize some observations concerning the logical requirements that we have encountered in our application-oriented setting.

Finite Domain Logic. In both examples we were either indirectly (Mercedes) or directly (IBM) confronted with finite domain logics. In the former case these finite domains (e.g. different engines, colors, countries) were already broken down into a set of propositional variables in the given rules. Explicitly using a finite domain logic for both constraint specification and automatic consistency checking would be advantageous, however, as that would allow for a more concise specification of, e.g., groups.

While it is hard to change a company's documentation method, finite domains can relatively easily be incorporated into a consistency checker. Our implementation, a variant of the Davis-Putnam (DP) procedure, takes this into account by offering a specialized *select-n-out-of-m*-construct [3], facilitating processing of groups of mutually exclusive variables.

Observations similar to ours were made by Béjar *et al.* [1].

Computational Complexity. Although the formulae to be checked were quite large—at least in the first of our case studies—and therefore in principle could reach or exceed the limits of today's prover technology, this turned out not to be the case. Satisfiability of formulae containing up to 1500 variables were solved by state-of-the-art SAT checkers [9, 6, 7, 3] without exception in under a second.

We do not yet fully understand the reasons for this, but our current hypothesis is that it is a problem-intrinsic phenomenon. We suppose that configuration constraints leading to invalid orders are always due to a very small subset of conflicting rules (which is also confirmed by first experiments). This would lead to the existence of short resolution proofs, and thus fast computations for the DP procedure. Further clarification of this behavior is not only of theoretical interest, as, e.g., SAP's product configurator uses only constraint propagation but does not perform a complete exhaustive search.

Our experiments with BDDs in the configuration domain were not very promising.

Explanation. We observed that a simple yes/no-answer as the result of a verification attempt is often not sufficient. During a considerable part of the verification phase—and even more in a validation setting as in our first case study—errors occur, and therefore the methodology is often mainly a debugging tool; so, a verification tool should give hints on where a possible error can be found.

For propositional (or modal) logic we see the need of explanation for both possible outcomes of a proof attempt: When a proof for some property is found, we want to know why it holds (*proof explanation*); when no proof is found, we are interested in an explanation consisting of a (complete) set of *counterexamples*. In our first case study, where error scenarios are characterized by successful proofs, we mainly need the first variant. We therefore integrated an explanation component into our prover [3] that is based on the computation of minimal unsatisfiable subsets (MUS). In our second case study faults correspond to failed proof attempts, so here the second setting is of relevance. We successfully used BDDs (and abstraction over irrelevant variables) to give a precise summary of the counter-models [8].

4 Practical Experiences

Besides the logical requirements there are more practical side-conditions that we consider important for the employment of verification in an industrial setting.

Push-button technology: The prover component and all unfamiliar logical language should be completely hidden from the user. The prover should be completely automatic, i.e., need no assistance in finding a proof. Interaction with the prover should be done in the terminology of the operating personnel.

Graphical user interface: The user should not be required to type in cryptic command lines. A graphical user interface is of great help, and boosts acceptance.

Short response times of the system: Especially when used as a debugging tool, short (and predictable) response times can be of importance to stabilize turn-around times.

Customized special checks: Consistency checks should be as specialized as possible. We observed a very poor user acceptance for a general-purpose proof option.

5 Summary

We presented two case studies of verification in industrial settings. As the experiences we made were similar in two quite different fields, we are confident that our observations hold in an even broader, more general context.

References

[1] R. Béjar, A. Cabiscol, C. Férnandez, F. Mànya, and C. Gomes. Extending the reach of SAT with many-valued logics. In *Workshop on Theory and Applications of Sat-*

isfiability Testing (SAT'2001), volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier Science, June 2001.

[2] E. Freuder. The role of configuration knowledge in the business process. *IEEE Intelligent Systems*, 13(4):29–31, July/August 1998.

[3] A. Kaiser. A SAT-based propositional prover for consistency checking of automotive product data. Technical report, WSI für Informatik, Universität Tübingen, 72076 Tübingen, Germany, 2001. Technical Report WSI-2001-16.

[4] C. Kern and M.R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[5] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Automated Reasoning*, 24(1–2):145–163, February 2000.

[6] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM, 2001.

[7] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In *Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier Science, June 2001.

[8] C. Sinz, T. Lumpp, and W. Küchlin. Towards a verification of the rule-based expert system of the IBM SA for OS/390 automation manager. In *Proc. 2nd Asia-Pacific Conf. on Quality Software (APAQS 2001)*, pages 367–374, Hong Kong, December 2001. IEEE Computer Society.

[9] H. Zhang. SATO: An efficient propositional prover. In *Proc. 14th Intl. Conf. on Automated Deduction (CADE-97)*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer-Verlag, 1997.