INDIAN JOURNAL OF
ENGINEERING

18(50), 2021

Logical and software engineering challenges in the verification of optimizing compilers

Seyfali Mahini

To Cite:
Mahini S. Logical and software engineering challenges in the verification of optimizing compilers. *Indian Journal of Engineering*, 2021, 18(50), 277-284

Author Affiliation:
Islamic Azad University, Khoy Branch, Khoy, Iran
Email: my1341pos@tyahoo.com

Peer-Review History
Received: 08 June 2021
Reviewed & Revised: 09/June/2021 to 05/July/2021
Accepted: 08 July 2021
Published: July 2021

Peer-Review Model
External peer-review was done through double-blind method.

ABSTRACT

The correctness of compilers is a necessary prerequisite for the correctness of the software translated with them. Optimizing compilers, in particular, are often flawed. In this paper, after an overview of the state of research, we present our new work on the verification of optimizing compilers. On the one hand, we discuss which logical problems arise in the formal verification of translation algorithms in compilers using theorem provers and which solutions we have developed for them. On the other hand, we show how the correctness of real optimizing compilers with a considerable implementation scope can be ensured. Our results thus contribute to the correctness of compilers, which are important tools in software engineering. In this way, we also develop methods that can also be used in other application areas for software verification.

Keywords: Compiler, Correctness, Optimizing, Verification, Translation, Implementation

1. INTRODUCTION

The correctness of compilers is necessary to ensure the correctness of the software translated with them. Even if one generally trusts compilers, these software tools still have their errors, such as the error messages from common compilers [Bo02] or e.g. the particularly obvious compiler error discussed in [Ne01]. Who does not know the phenomenon that you are desperately looking for errors in your own program and these errors disappear as soon as you switch off optimization levels?.

In our work we solve the question of how optimizing compilers, which turn out to be particularly error-prone, can be verified. We apply the following criteria to our solution: Correctness should be proven formally, i.e. strictly in a logical system by means of a machine theorem prover. In addition, our methods should be applicable to realistic compilers. From a research perspective, this problem is interesting for three reasons: Compilers are relatively large software systems, so that one can see from them how well the verification methods used and developed are scale. This problem is also interesting from a semantic-logical point of view, because not only refining,

but in particular also structurally changing, optimizing transformations are considered, which are not possible with the verification methods that have hitherto been used can be treated. And finally, many of the methods developed in the field of verification of compilers can be used in other software and hardware areas. In this paper we give a summary of the state of the art an overview of our new work on the verification of optimizing translators.

2. CORRECTNESS OF COMPILERS: STATE OF RESEARCH AND TECHNOLOGY

When it comes to the correctness of translators, a distinction is made between two different questions: On the one hand, one examines whether a given translation algorithm is correct, i.e. whether it contains the meaning, the semantics of the transformed programs. On the other hand, the question arises as to whether a possibly previously verified translation algorithm is also correctly implemented in an existing compiler. The first correctness term, which relates to the semantic correctness of the translation algorithms, is referred to as translation correctness, the second, which considers the correctness of the implementations in compilers, is called implementation correctness. Implementation correctness was considered for the first time in [Po81, CM86]. In the following we present the state of research and technology with regard to these two terms.

Translation correctness: In the literature, verifications of refinement transformations are mostly considered. These are translations in which the structure of the programs is not changed, but rather how the individual calculations are to be carried out is determined more and more precisely during the translation. For example, when translating high-level programming languages into machine code, you would determine how complex data structures are mapped into the memory hierarchy of the processor. This means in particular that programs can be translated locally according to a Divide et Impera principle and then the translations can be put together again. The corresponding proofs of correctness follow this principle: correctness can be shown locally and global correctness follows from it. Verifications according to this scheme can be found in [DvHG03, SA97], among others.

Implementation correctness: The question of the implementation correctness of compilers is of great importance from a software point of view, especially when you consider that translators nowadays are not written by hand, but generated from suitable specifications by means of generators. You could now try to verify these generators yourself. In view of the size of these systems, this option is not applicable, especially if you want to use machine proofs in theorem provers. With the verification methods available today, it is not (yet) possible to get a grip on software of this size. Instead, you could try to verify the generated compilers. This possibility is also omitted, for the same reason.

As a way out of this dilemma, program checking has established itself as the method of choice in recent years, also known in the literature as translation validation [PSS98] or program checking [GGZ98]. Instead of verifying the translator, one only verifies his result. One proceeds in such a way that the compiler is enriched with an independent checker.

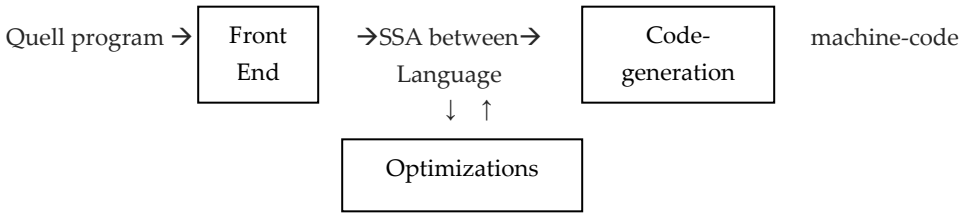


Figure 1: Compiler architecture

The checker receives the source program and the target program generated from it as input and checks whether the two programs are semantically equivalent. In the event of a positive decision, we have proof that a correct translation has actually taken place. In the negative case you don't know anything. From a theoretical point of view, one cannot expect that the checker can always decide whether a correct transformation has taken place, because program equivalence is an undecidable problem. From a practical point of view, however, this approach has proven to be very suitable. If you continue to formalize the checker with regard to a suitable specification, you get a positive one Decision of the checker a formally verified translation result. This method for verifying the correctness of the implementation has proven to be very suitable in all phases of the front ends of compilers (lexical, syntactic and semantic analysis), especially because the results in these phases of the translation are unambiguous [HGG + 99, Gl03b, GFJ04]. In the syntactic analysis, for example, one must first check whether the syntax tree calculated by the compiler actually matches the context-free grammar of the programming language, which can be done in a top-down run through the syntax

ISSN 2319–7757 EISSN 2319–7765 | OPEN ACCESS

tree by adding X_0 and its successor nodes $X_1, \ldots, X_n$ it is checked whether there is a production $X_0 :: = X_1 \cdot \cdot \cdot X_n$. Second, one must ensure that the syntax tree is a correct derivation of the original program, which is checked by ensuring that the text that is created by chaining the pages in the syntax tree matches the original program.

Problems that have not yet been resolved: It has not yet been clarified how the correctness of the translation of structure-changing transformations can be proven. We will deal with this problem in Section 3. We will consider the generation of code from SSA (static single assignment) intermediate languages and discuss two alternative proof possibilities. Furthermore, previous work has not clarified how for optimizing transformations in which there are several, possibly even many correct solutions, implementation correctness can be ensured. In Section 4 we present a solution to this.

3. COMPILER OPTIMIZING THE TRANSLATION CORRECTNESS

In this section we consider the generation of code in compilers and show how the correctness of the translation for this phase can be demonstrated automatically with the help of a theorem prover. Figure 1 shows the rough architecture of compilers which first transform a source program with the front end (consisting of lexical, syntactic and semantic analysis) into an internal intermediate representation. This intermediate representation can then be optimized with machine-independent transformations. Executable machine code is then generated in code generation. We are assuming an SSA (static single assignment) -based intermediate representation [CFR + 91], because the essential data dependencies are represented directly in programs and optimizations are particularly possible. We first introduce SSA intermediate languages in Section 3.1 and then present two ways of proving the correctness of the code generation in Sections 3.2 and 3.3. We discuss the advantages and disadvantages of the two alternatives in Section 3.4. In our work we have used the generic theorem prover Isabelle [NPW02], which can be instantiated with different logics. We use the HOL (higher order logic) instantiation, because this has already been preserved during the formalization of a significant subset of Java [KN03].

3.1. SSA intermediate languages

Like most intermediate languages in compilers, SSA languages are also basic block-oriented, i.e. maximum sequences of non-branching instructions are arranged in basic blocks, and the control flow connects the basic blocks with one another. In addition, in the SSA representation, each variable is statically assigned a value only once, a representation that can be achieved, among other things, by appropriately duplicating and renaming the variables can always be achieved and by means of which one can recognize the essential data dependencies of a program particularly well. Within basic blocks are calculations exclusively data flow-driven, i.e. an operation can be carried out as soon as its input values are fixed. In this paper, for reasons of space, we only consider that (semantically more interesting) case of compiling basic blocks and neglecting all details that are unimportant for this purpose. Refer to for a detailed illustration [BG04]. In the context of this paper, basic blocks can be thought of as acyclic data flow graphs (DAGs for short).

3.2. Proof of the correctness of the translation

Formal semantics of the programming languages involved form the basis of every formal verification of translations. So we first need a formal semantics of the Basic SSA Blocks. As mentioned above, basic SSA blocks can be thought of as DAGs. The question of how to represent DAGs or graphs in general in a logical language such as HOL has not yet been clearly clarified in research and depends on the context of the verifications. We decided to use term graphs as a representation.

SSA basic blocks contain common partial expressions, the results of which are used in several places, only once. This is why basic SSA blocks are generally not terms, but DAGs. We formalize SSA basic blocks by transforming their DAGs into an equivalent set of terms and thereby duplicating common partial expressions, see Figure 2. In order to identify equivalent partial terms with one another, we assign and duplicate a unique identification number to each operation in the original SSA DAG this number whenever we duplicate a partial term in the DAG. This way we can convert a SSA-DAG into a set of Transform SSA terms. In Isabelle / HOL we have specified such SSA terms with the following definition, which is only shown here in a simplified manner:

ISSN 2319–7757 EISSN 2319–7765 | OPEN ACCESS

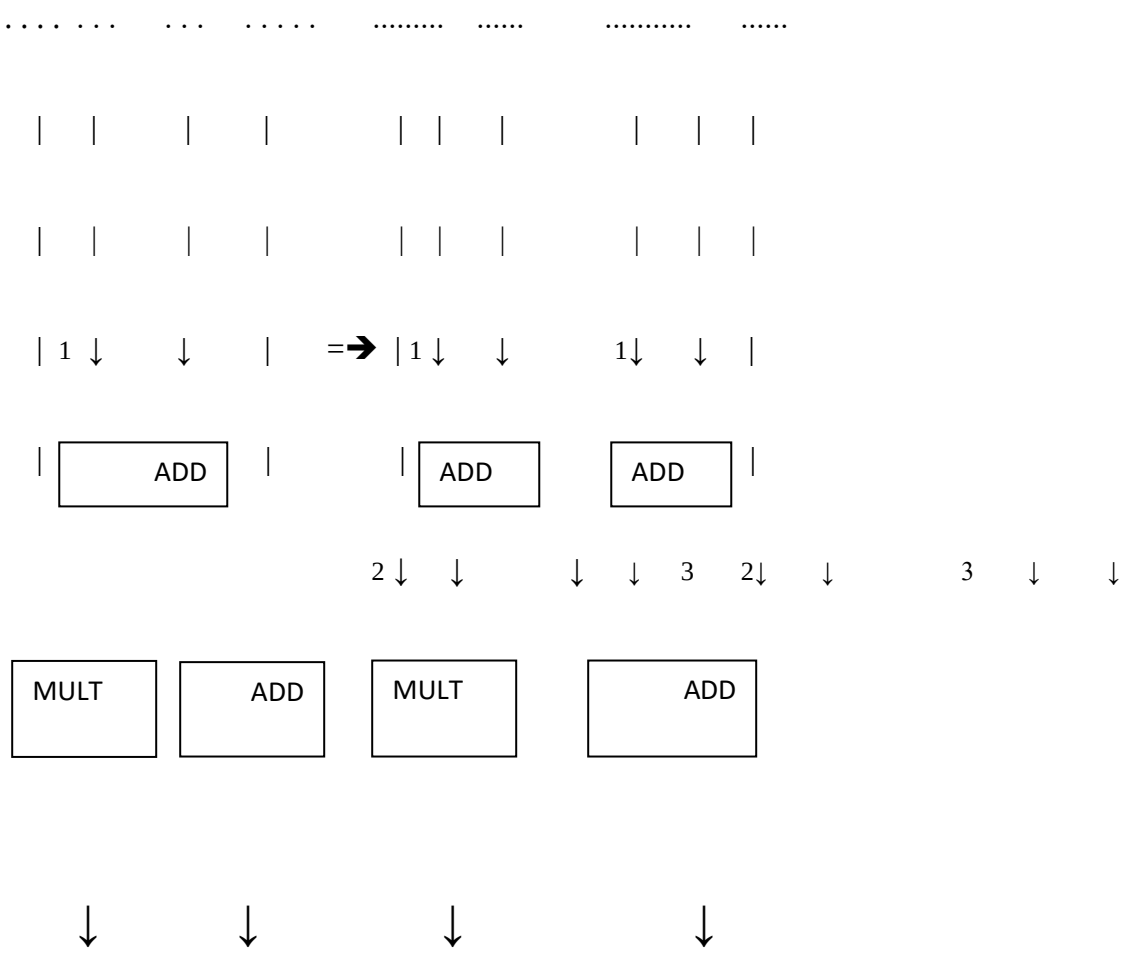


Figure 2: Transformation of SSA-DAGs into SSA trees

datatype *SSATree = CONST value ident | NODE operator SSATree SSATree value ident*

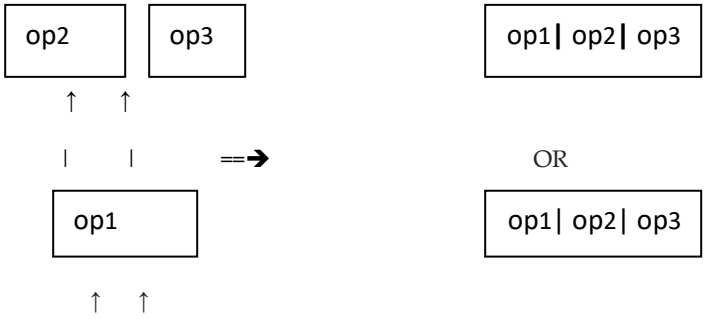
Without going into the Isabelle-specific details, one recognizes that an inductive data type SSATree is specified, the elements of which either consist of a leaf or are trees composed of two subtrees. Furthermore we have defines a function eval_tree that takes an SSA tree and evaluates it by assigning its result to each node, i.e. operation in the tree. The signature of this function is defined as follows:

consts *eval_tree ::" SSATree ⇒ SSATree"*

In order to prove that the generation of machine code is correct, we have to formalize the semantics of the target machine language as well as the mapping of the SSA basic blocks into this language. We have chosen a relatively simple machine language whose operations correspond directly to the corresponding operations of the SSA basic blocks and whose programs are a sequential list of machine instructions that are processed in sequence. For an exact definition of the corresponding semantics, we refer to [BG04]. When mapping SSA DAGs in machine code have some degrees of freedom. For example, when transforming the DAG shown on the right, you must first calculate the operation op1 and then you have the choice of whether to calculate op2 first and

ISSN 2319–7757 EISSN 2319–7765 | OPEN ACCESS

then op3 or first op3 and then op2. We were able to prove in Isabelle / HOL that every order that is a topological sorting of the SSA-DAG represents a valid code generation order.



With this proof we have the correctness of the mapping between the data flow-driven calculations in SSA-DAGs and the sequentially ordered instructions in the machine code shown. This mapping is correct if the data dependencies of the SSA basic block are retained. For this proof we needed 885 lines of proof code in Isabelle / HOL, which is relatively extensive, especially when you consider that proofs in Isabelle / HOL have to be carried out interactively by hand and cannot be generated. In addition, the verification seemed unnaturally complex to us in some places. These difficulties arose because we had different induction principles in the SSA tree and in the machine code list (induction over trees and induction over lists). We have taken these observations as an opportunity to create a completely new proof, which we report on in the following.

3.3. Alternative proof of the correctness of the translation

Based on the observation that code generation from SSA basic blocks is correct precisely when the data dependencies are retained, we have created a completely new specification of SSA basic blocks. We have consistently understood SSA basic blocks as partial orders on the operations contained in them. An operation o is smaller than another operation o', if o must be evaluated before o' because the result of o' depends directly or indirectly on the result of o. As usual in mathematical logic, we have represented a partial order O as a set of tuples (x, y) such that $(x, y) \in O$ and $(y, z) \in O$ also implies $(x , z) \in O$ (transitivity) and that from $(x, y) \in O$ and $(y, x) \in O$ it follows that $x = y$ applies (antisymmetry). We formalized code generation as a process that injects further dependencies into the partial order and thus turns the partial order into a total order. We were able to prove that code generation is correct if the original SSA order is contained in the order of the machine code. For details of the corresponding Isabelle proof, for which we unfortunately do not have space here, we refer to [BGLM04].

3.4. Discussion of the two alternatives

The correctness proofs for code generation from SSA basic blocks discussed in Sections 3.2 and 3.3 are both included in the theorem prover Isabelle / HOL and thus both show the same result that code generation is correct if the data dependencies of the SSA basic blocks are retained. If one is only interested in this result, both proofs are equivalent, especially because they hardly differ in their length. From a mathematical and logical point of view, however, they differ greatly. Behind this is an experience that mathematicians also have in their work. There is evidence that feels "good" that is properly intuitioned. You cannot give a formal definition of when a given proof actually feels good, but you usually know exactly as soon as you have it, see also [AZ04]. We have also had this experience. The second proof based on partial orders feels good. The individual proof steps fit together with our intuitive proof idea and formalize a general principle, namely that the transformation of a program preserves the data dependencies got to. Since we have based our proof on the general principle of “preservation of data dependencies”, we can also reuse the proof in the verification of further transformations. In current work we use it, for example, to verify the elimination of dead codes and to verify loop transformations, both of which are typical optimizations in modern compilers. Many other uses exist.

Formal software verification with the help of theorem provers such as Isabelle / HOL is possible nowadays for two reasons: First, the speed of processors has increased so that the search spaces of theorem provers can be traversed sufficiently quickly. Second, theorem provers have become so user-friendly that these systems are also used by working groups that were not involved in their development. Nevertheless, formal software verification is very expensive and requires complex user interactions. This effort can be reduced if we succeed in reusing evidence by building evidence on general principles for the correctness of system

ISSN 2319–7757 EISSN 2319–7765 I OPEN ACCESS

transformations (such as the preservation of data dependencies here). Our work on translation correctness presented here is an example of this.

4. IMPLEMENTATION CORRECTNESS OPTIMIZING COMPILERS

As with almost all problems, the code generation phase is to solve the optimization variant of an NP-complete problem in the backends of compilers. Problems in NP are characterized by the fact that their solutions can always be checked for correctness in polynomial time [Pa94]. This means that a non-deterministic Turing machine has to traverse a potentially exponentially large search space before it finds a solution, but the solution itself can always be found in polynomial depth, Figure 3. If you know the way to such a solution (also called a certificate in complexity theory), then you can calculate the solution quickly, i.e. in polynomial time. Typically, the certificates of NP complete problems are natural. In the case of the SAT problem (satisfiability of propositional logic Formulas) e.g. such a certificate contains the relevant assignment.

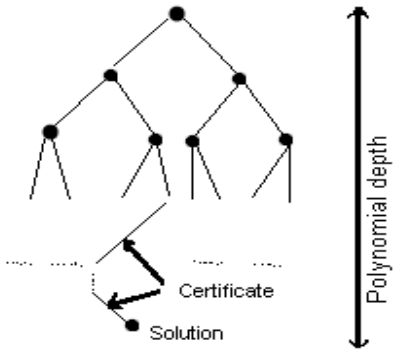


Figure 3: NP-Calculation

We take advantage of this connection when examining the correctness of the implementation for the code generation. We are modifying the previous checker scenario in such a way that the compiler not only generates the target program but also a log in which it is recorded how the solution was calculated. The checker receives this log as the third input and proceeds as follows: It calculates a target program from the input program with the aid of the certificate, compares the target program it has calculated itself with the one calculated by the compiler, and gives in the case if they match a "yes" answer, in the negative case the result "don't know". We refer to this testing method as *program testing with certificates*, Figure 4.

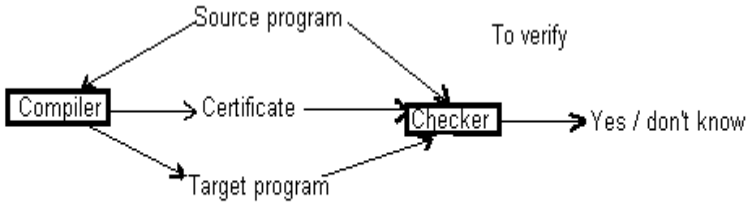


Figure 4: Program checks with certificates

One might wonder what happens when the compiler issues a bad certificate. If the checker calculates a correct result with such an incorrect certificate and if this result continues to agree with the result calculated by the compiler, then the checker has managed to verify the result calculated by the compiler. It doesn't matter how the compiler determined its result, as long as the checker was able to reconstruct with its (verified) implementation.

Let us take another look at the role of the code generator and the checker. The code generator behaves like a non-deterministic Turing machine in that it searches for and calculates the solution. The checker, on the other hand, acts like a deterministic Turing machine and calculates the solution in the same way as the code generator with the aid of the certificate. This means that the implementation of the checker can be expected to agree with part of the implementation of the code generator, namely with the part

ISSN 2319–7757 EISSN 2319–7765 I OPEN ACCESS

that computes the solution. We have found this expectation confirmed in our experiments. At first glance, this may seem astonishing because checkers are originally used to be able to carry out a test for correctness that is independent of the implementation. The testing paradigm has changed here. We do not use the checker for such an independent test, but rather as a method to separate the correctness-critical part of an implementation. The calculation of the solution is critical for correctness, the search for a good solution is not, because the goodness of a solution does not influence its correctness.

	Code geberator	Checker
Lines of code in .h-Files	949	789
Lines of code in .c-Files	20887	10572
Total lines of code	21836	11361

The table below shows our experimental results in figures. We have a program checker for a code generator using the method of program checking with certificates designed and implemented in an industrial project [Gl03a, Gl03b]. The code generator covers more than 20,000 loc, the checker oly about half. You can already see from this that we were able to significantly reduce the verification effort. If you also consider that the error-prone and laboriously verifiable search for an optimal or good one in the checker Solution is not included, you can see that we were able to significantly reduce the verification effort with the method of program checking with certificates.

5. CONCLUSION

When verifying optimizing compilers, both logical and software problems have to be solved. On the one hand it must be verified that the applied transformation algorithms preserve the semantics of the translated programs (translation correctness). On the other hand, it must be ensured that these algorithms are also correctly implemented in a compiler (implementation correctness). We have in our work showed how correctness of translation for optimizing code generation based on SSA-based intermediate languages, a modern intermediate representation in optimizing compilers, and have presented and compared two alternative proof approaches. We also have the method we developed of program auditing with certificates, with which we ensure the correctness of solutions in optimization problems. With our results, we have not only contributed to optimizing compilers, which are an important tool in the software technology to be verified, but methods are also developed that can be used in general for the transformation of hardware and software systems. For example, with the Model Driven Architecture (MDA) approach, the system specification is specified independently of the system implementation. In the transformation and implementation of such a system specification, methods of compiler construction are also used, in particular the verification techniques presented, with which one can ensure that the implementation of a system is correct with regard to its specification.

Funding

This study has not received any external funding.

Conflict of Interest

The author declares that there are no conflicts of interests.

Data and materials availability

All data associated with this study are present in the paper.

REFERENCES AND NOTES

1. [AZ04] Aigner, M. und Ziegler, G. M.: *Proofs from THE BOOK*. Springer-Verlag. 2004.
2. [BG04] Blech, J. O. und Glesner, S.: A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In: *34. Jahrestagung der GI*. LNI. 2004.
3. [BGLM04] Blech, J. O., Glesner, S., Leitner, J., und M¨ulling, S. Some Theorems on Data Dependencies using Partial Orders. 2004. Internal Report, University of Karlsruhe.
4. [BK95] Blum, M. und Kannan, S.: Designing Programs that Check Their Work. *JACM*. 1995.
5. [Bo02] Borland/Inprise: *Official Borland/Inprise Delphi-5 Compiler Bug List*. http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html. 2002.
6. [CFR+91] Cytron, Ferrante, Rosen, Wegman, und Zadeck: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM TOPLAS*. 13(4). 1991.

7. [CM86] Chirica, L. M. und Martin, D. F.: Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*. 8(2):185–214. 1986.
8. [DvHG03] Dold, A., von Henke, F.W., und Goerigk,W.: A Completely Verified Realistic Bootstrap Compiler. *Int'l Journal of Foundations of Computer Science*. 14(4):659–680. 2003.
9. [GD04] Große, D. und Drechsler, R.: Checkers for SystemC Designs. *Proc. 2nd ACM & IEEE Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE'2004)*. 2004.
10. [GFJ04] Glesner, S., Forster, S., und J¨ager, M.: A Program Result Checker for the Lexical Analysis of the GNU C Compiler. 2004. Elsevier, Elec. Notes in Theor. Comp. Sc. (ENTCS).
11. [GGZ98] Goerigk, W., Gaul, T., und Zimmermann, W.: Correct Programs without Proof? On Checker-Based Program Verification. In: *ATOOLS'98*. 1998. Springer.
12. [GGZ04] Glesner, S., Goos, G., und Zimmermann, W.: Verifix: Konstruktion und Architektur verifizierender U¨ bersetzer. *it - Information Technology*. 46:265–276. 2004.
13. [Gl03a] Glesner, S.: Program Checking with Certificates: Separating Correctness-Critical Code. In: *12th Int'l FME Symposium (Formal Methods Europe)*. 2003. Springer, LNCS 2805.
14. [Gl03b] Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Comp. Sc. (J.UCS)*. 9(3):191–222. March 2003.
15. [HGG+99] Heberle, A., Gaul, T., Goerigk, W., Goos, G., und Zimmermann, W.: Construction of Verified Compiler Front-Ends with Program-Checking. In: *Perspectives of System Informatics, Third Int'l A. Ershov Memorial Conf.*. 1999. Springer, LNCS 1755.
16. [KN03] Klein, G. und Nipkow, T.: Verified Bytecode Verifiers. *TCS*. 298:583–626. 2003.
17. [Mo89] Moore, J. S.: A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*. 5(4):461–492. 1989.
18. [Ne97] Necula, G. C.: Proof-Carrying Code. In: *POPL'97*. 1997. ACM.
19. [Ne01] Newsticker, H.: *Rotstich durch Fehler in Intels C++ Compiler*. http://www.heise.de/newsticker/data/hes-11.11.01-000/. 2001.
20. [NPW02] Nipkow, T., Paulson, L. C., und Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Lecture Notes in Computer Science, Vol. 2283. 2002.
21. [Pa94] Papadimitriou, C. H.: *Computational Complexity*. Addison-Wesley. 1994.
22. [Po81] Polak, W.: *Compiler Specification and Verification*. Springer, LNCS 124. 1981.
23. [PSS98] Pnueli, A., Siegel, M., und Singerman, E.: Translation validation. In: *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*. 1998. Springer, LNCS 1384.
24. [SA97] Schellhorn, G. und Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*. 3(4):377–413. 1997.

ISSN 2319–7757 EISSN 2319–7765 | OPEN ACCESS