

Program Checking with Certificates: Separating Correctness-Critical Code

Sabine Glesner

Institut für Programmstrukturen und Datenorganisation,
Universität Karlsruhe, 76128 Karlsruhe, Germany,
<http://www.info.uni-karlsruhe.de/~glesner>

Abstract. We introduce program checking with certificates by extending the traditional notion of black-box program checking. Moreover, we establish program checking with certificates as a safety-scalable and practical method to ensure the correctness of real-scale applications. We motivate our extension of program checking with concepts of computational complexity theory and show its practical implication on the implementation and verification of checkers. Furthermore, we present an iterative method to construct checkers which is able to deal with the practically relevant problem of incomplete or missing specifications of software. In our case study, we have considered compilers and their generators, in particular code generators based on rewrite systems.

Keywords: program checking, certificates, correctness, validation, verification, safety-scalability, real-scale applications.

1 Just Let Me Double-Check!

Formal correctness of software is a desirable, yet expensive property. Program checking aims at reducing this cost. Instead of verifying a piece of software, one only verifies its result. In this paper, we present the notion of program checking with certificates which extends the established version of black-box program checking. Furthermore, we show that program checking with certificates is a practical and safety-scalable method which has proved itself in real-scale applications. To be practical and safety-scalable, we require a method to fulfill the following criteria: Its reliability should be scalable wrt. safety, ranging from “only” validated up to formally verified software. In safety-critical applications, one needs formally verified software. Hence, on its top level of reliability, the desired method should formally ensure the correctness of the computed results. But in many situations, one can already be satisfied with a sufficiently increased confidence that the software does indeed exactly what is defined in its specification. Furthermore, we require the method to have a manageable effort in practice. In particular, we want to avoid to verify the software in its entirety. This is an important requirement as many parts of a given piece of software do not influence the correctness of its results. E.g. optimizations are intended not only to compute a correct but also an optimal result. In many cases, its correctness

can be established independently of its quality. Hence, we do not want to spend verification effort for the quality but only for the correctness of the result. There are even more parts of software which do not influence the correctness of the result. Nearly all software evolved over a certain amount of time and maintained by several programmers contains code which is not used any more. Programmers tend to leave over old code instead of deleting it. A practical method for ensuring the correctness of software should be able to deal with such scenarios. Moreover, it should be applicable for generated software. Consequently, the involved generators should be extendable to generate not only the desired software pieces but also the proofs for their correctness. As a sacrifice, we do not require total but only partial correctness. It is sufficient if the method proves the correctness of each result individually. If it cannot verify the correctness of a result, it simply says ‘no’, giving us only partial formal correctness.

With our investigations we show that program checking with certificates can cope with these requirements. Originally, program checking [BK95] assumes that an implementation P computing a function f is used as a black box. An independent program result checker for f checks for a particular input x if $P(x) = f(x)$. We extend the notion of a program result checker such that it is allowed to observe the implementation program P during its computation of the result. In particular, the program P might pass a certificate to the checker, telling the checker how P has computed the solution. This extension is natural as in many cases one has access to the implementation code. Moreover, it fits to the nature of search and optimization problems, in particular to **NP**-complete problems. Such problems are characterized by the fact that proofs for the correctness of a solution can be checked easily while the computation of solutions is believed to be significantly harder. We argue that a certificate is such a correctness proof. This gives us evidence that for many problems, checkers are significantly simpler than the actual implementations. In theoretical considerations (e.g. the checkers for numerical problems in [BLR93]), the specification of the checker is clear. It is supposed to check if a certain function f has been computed correctly for one specific argument x . Hence, the precondition of the checker states that x and $P(x)$ are admissible inputs. Its postcondition states that the output is ‘yes’ iff $P(x) = f(x)$. In practical situations, the specification of a complex system is not that obvious, making subsequent correctness proofs more difficult. Since software evolves over time, its specification might also change. We show that program checking can be deployed to determine pre- and postconditions in an iterative process. Therefore we start with a preliminary specification, implement a checker checking this specification, and use the checker for typical pairs of input and output values. Whenever the checker cannot establish the correctness of the result, we either have found a mistake in the implementation or the checker specification needs to be revised. This proceeding is continued until we are sufficiently assured that the specification of the checker reflects the connection between input and output values. In contrast to pure testing, the checker will say ‘no’ for all cases not included in the specification, instead of allowing the implementation to do something unpredictable. On the highest level of re-

liability, we require a formal proof for the correctness of the checker with an automated theorem prover. If this level of security is not necessary, we can still use the checker without a formal proof, giving us a safety-scalable method. In our experiments, we found that program checking emerges as a novel combination of existing validation and verification techniques.

As a real-scale test case, we have chosen compiler backend generators and the thereby generated compiler backends. In particular, we have focused on designing and implementing a checker for the CGGG system [Boe98]. This system generates compiler backends which transform intermediate SSA (static single assignment) representations into native machine code based on BURS (bottom-up rewrite systems). CGGG has been utilized to generate a compiler in the AJACS (Applying Java to Automotive Control Systems) project with industrial partners [Gau02, GKC01]. Compiler backend generators are well-suited as a test case due to several reasons. Typically, compiler generators are maintained for a long period of time. Several developers with changing design objectives and optimization goals modify the generators by adapting them to the constantly changing concepts in source programming languages or target machine architectures. This results in “naturally grown software” with a large variety of functionality, much of which is not used in the generation of one particular backend. Possibly, there are even combinations of different functionalities which are never chosen and which, if used together, may even be incorrect. A formal verification aiming for total correctness could fail due to such scenarios. Nevertheless, each actually generated backend could still be correct. Hence, a notion of partial correctness and, in turn, program checking is adequate for the correctness of compiler backends.

This paper is organized as follows. In section 2, we introduce our approach of program checking with certificates. In section 3, we discuss the nature of program checking in practice and derive a general safety-scalable validation and verification method. In section 4, we describe our case study regarding compiler generators. Our experimental results are explained in section 5. We discuss related work in section 6 and conclude in section 7.

2 Program Checking: Trust Is Good, Control Is Better!

In this section, we introduce the notion of program checking with certificates. It modifies the classical concept of black-box program checking [BK95] which is summarized in subsection 2.1. In subsection 2.2, we show how it can be extended to check the correctness of solutions for optimization problems by using certificates. Moreover, in subsection 2.3, we argue why it is unlikely that we might be able to construct efficient checkers for the optimality of solutions.

2.1 Classical Black-Box Program Checking

Program checking [BK95] has been introduced as a method to improve the reliability of programs. It assumes that there exists a black box implementation P computing a function f . A *program result checker* for f checks for a particular

input x if $P(x) = f(x)$. Assume that $f : X \rightarrow Y$ maps from X to Y . Then the checker *checker* has two inputs, x and y , whereby x is the same input as the input of the implementation P and y is its result on input x . The checker has an auxiliary function *f-ok* that takes x and y as inputs and checks if $y = f(x)$ holds.

```

proc checker( $x : X, y : Y$ ) : BOOL
  if f-ok( $x, y$ ) then return True
  else return False
end proc
    
```

Note that the checker does not depend on the implementation P . Hence, it can be used for any program P' implementing f . Thereby the checker should be simpler than the implementation of f itself and, a stronger requirement, simpler than any implementation P' computing f . Simple checkers would have

the advantage of potentially having fewer bugs than the implementation P . Since there is no reasonable way to define the notion of being simpler formally, [BK95] states a definition for being *quantifiably different*. The intention is to force the checker to do something different than the implementation P . A checker is forced to do something different if it has fewer resources than the implementation. This would imply that bugs in the implementation and in the checker are independent and unlikely to interact so that bugs in the program will be caught more likely. Formally, a checker is quantifiably different than the implementation if its running time is asymptotically smaller than the running time of the fastest known algorithm. However, for many interesting problems, the fastest algorithm is not known. As a weaker requirement, one can consider *efficient* checkers whose running time is linear in the running time of the checked implementation and linear in the input size. [BLR93] presents checking methods for a variety of numerical problems.

2.2 Program Checking with Certificates

We introduce the method of program checking with certificates by extending the notion of black-box program checking: We allow the checker to observe the implementation program P during its computation of the result. In our setting, the program P might tell the checker how it has computed its solution.

To motivate our idea theoretically, let us take a look at the common definition for problems in **NP**. **NP** is the union of all problems that can be solved by a nondeterministic polynomial time Turing machine. An alternative equivalent definition for **NP** states the following: Assume that a language L is in **NP**. Then there exists a polynomial time Turing machine M such that for all $x \in L$, there exists y , $|y| \leq poly(x)$ such that M accepts (x, y) . Hence, for any language L in **NP**, there is a simple proof checker (the polynomial time Turing machine M) and a short proof (y) for every string $x \in L$. Given the proof y and the string x , the proof checker M can decide if the proof is valid. Clearly, the two definitions are equivalent: Com-

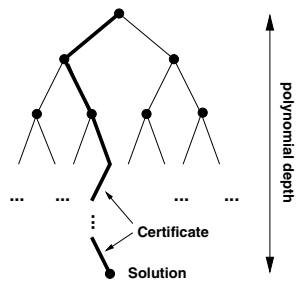


Fig. 1. Computation Spaces in **NP**

putations within nondeterministic polynomial time can be thought of as a search tree with polynomial depth. Each node represents the choice which the nondeterministic Turing machine has at any one time during computation. A proof for membership in the language L is a path to a solution through this search tree. Since the tree has polynomial depth, there always exists a proof of polynomial length. Such a proof is also called a *certificate*, cf. [Pap94]. **NP**-complete problems have the tendency to have very natural certificates.

When solving optimization problems, huge search spaces need to be searched for an optimal or at least acceptable solution. When we want to check its correctness, we do not care about its optimality. Hence, we can use the certificate to recompute the result. In particular for the optimization variants of **NP**-complete problems, we have the well-founded hope that the checker code is much easier to implement, and in turn to verify, than the implementation itself. Our checking scenario with certificates is summarized in Fig. 2.

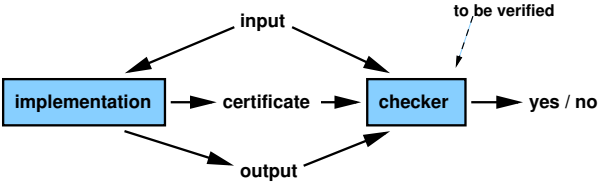


Fig. 2. Checker Scenario with Certificates

But what if the implementation is malicious and gives us a buggy certificate? The answer is simple: If the checker manages to compute a correct solution with this erroneous certificate and if, furthermore, this correct solution is identical with the solution of the implementation, then the checker has managed to verify the correctness of the computed solution. It does not matter how the implementation has computed the solution or the certificate as long as the checker is able to reconstruct the solution via its verified implementation.

```

proc checker( $x : X, y : Y, Certificate$ ) : BOOL
  if  $f\_ok(x, y, Certificate)$  then return True
  else return False
end proc
  
```

We describe the checker functionality as follows. Let P be the implementation of a function f with input $x \in X$ and the two output values $y \in Y$ and $Certificate$ such that y is supposed to be equal to

$f(x)$ and $Certificate$ a description how the solution has been computed. The checker has an auxiliary function f_ok that takes x, y , and $Certificate$ as inputs and checks whether $y = f(x)$ holds.

2.3 Can We Check the Optimality of Solutions?

Our notion of checking with certificates makes sure that a solution is correct but does not consider checking its quality. In this subsection, we argue that due to

certain widely-accepted assumptions in complexity theory, we cannot hope to construct efficient checkers which check if a solution is optimal.

Problems in **NP** are always decision problems, asking if a certain instance belongs to a given language (e.g.: Is there a Hamiltonian path? Does the travelling salesman have a tour of at most length n ?). These problems are characterized by their property that each positive instance has a proof of polynomial length, a *certificate*. E.g. the Hamiltonian path itself or a tour for the travelling salesman of length n or smaller would be such certificates. Conversely, the class **coNP** is defined as containing all those languages whose negative instances have a proof of non-membership, a *disqualification*, of polynomial length. E.g. the language containing all valid propositional formulas is such a language. A non-satisfying assignment for a formula proves that this formula does not belong to the language of valid formulas. Hence, this non-satisfying assignment is a disqualification. To prove that a solution is not only correct but also optimal, one would need a positive proof in the spirit of **NP**-proofs and a negative proof as in the case of **coNP**-proofs. The positive proof states that there is a solution at least as good as the specified one. The negative proof would state that there is no better solution. Complexity theory [Pap94] has studied this situation and defined the class **DP**. **DP** is the set of all languages that are the intersection of a language in **NP** and a language in **coNP**. One can think of **DP** as the class of all languages that can be decided by a Turing machine allowed to ask a satisfiability oracle twice. This machine accepts iff the first answer was ‘yes’ (e.g. stating that the optimal solution is at least as good as the specified one) and the second ‘no’ (stating that the optimal solution is at most as good as the specified one). It is a very hard question to decide whether an optimization problem lies in **DP**. The current belief in complexity theory is that **NP**-complete problems are not contained in **coNP**, implying that conceivably they do not have polynomial disqualifications. So if we design a checker for a problem being at least **NP**-complete, it does not surprise that we are not able to announce a polynomial checker also for the optimality of a solution, since such an announcement would solve a few very interesting questions in complexity theory.

3 Program Checking in Practice

In practical applications, program checking poses problems which do not appear in the theoretical setting discussed in the previous section. We discuss these problems in subsection 3.1. In subsection 3.2, we propose an iterative method to cope with them.

3.1 Problems in Practice

No Explicit Input-Output-Mapping: In many software systems, there is not one single function to be computed but rather a (finite) sequence of functions. Even though their concatenation could theoretically be expressed within one single function, this is not practical. It would result in a clumsy formulation.

A typical example for this problem are compilers which transform a source program by applying a sequence of compilation steps. The transformation is split up for efficiency reasons. Hence, the checker approach must be able to deal with such situations as they are typical for many software systems. In consequence, one needs to check a sequence of results. Some of these results are only available in main memory (otherwise the implementation would be too inefficient). This implies that the checker needs to run together with the implementation program so that the necessary checks can be done on the fly. A tight interlocking between checker and implementation program is necessary. Therefore it is absolutely necessary to have access to the implementation code.

Incomplete or Nonexistent Specifications: As a major practical problem, complete specifications do only rarely exist. As software has evolved over time, the most prominent functionality is known. But a complete specification defining results for special cases does not exist and might be very hard to set up. E.g. in our case study of compiler generation tools, we have the typical situation that specification and implementation of compiler passes are not fully separated. It is a common situation that the specification for a compiler generator might contain implementation code. As well-known examples, consider the Unix tools Lex and Yacc for the generation of the lexical and syntactic analysis. Input specifications for them can contain C code which will become part of the generated compiler pass. This observation holds for basically all compiler generators, in particular for the CGGG system considered in our case study.

3.2 Iterative Checking Method

We propose an iterative method to deal with the problem of non-existing specifications. Therefore, we postulate a preliminary specification which is revised iteratively. To deal with the existence of intermediate results and the necessity to check them, we use a multistage process, cf. also Fig. 3. We assume that we are given a hand-written or generated program without a formal specification. The first step is to guess the specification by assuming postulates, consisting of a pre- and a postcondition and of assertions describing intermediate states or results reached during the run of the program. This is the truly creative part of program checking in practice and requires code inspection. In the second step, checkers are to be implemented that check whether the postulates hold. Then we need to make sure that our assumed specification meets reality. Therefore we need to make tests which ensure that the specification covers the relevant pairs of input and output values. If such tests fail, we need to revise the specification (or correct bugs in the implementation), adapt the checkers, and start again with tests. We are done whenever we are confident that the specification and the respective checkers cover all practically interesting pairs of input and output values. In contrast to pure testing, the checker will say ‘no’ for all cases not included in the specification.

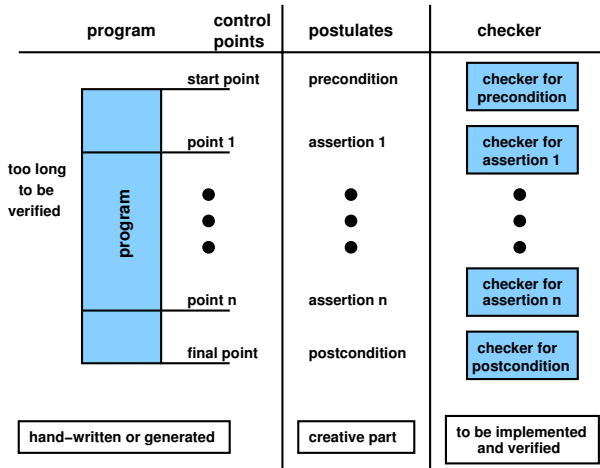


Fig. 3. Program Checking as a Validation and Verification Method

4 Case Study: Compilers and Compiler Generators

Compilers and their generators are good candidates for investigating the benefits of program checking in practical applications as they show the two major problems discussed in subsection 3.1. In our case study, we have considered compiler backend generators and the corresponding generated compiler backends.

The two major practical problems are the nonexistence of explicit input-output mappings and the absence of complete specifications. Both problems arise in the area of compiler construction. Compilers transform input programs consisting of a sequence of characters into target machine code. This translation consists of several steps, separated by explicit interfaces. The overall translation function as a concatenation of these individual steps is never stated explicitly as it would be much too unwieldy. Hence, we do not have an explicit input-output mapping for compilers, not even for their individual steps. As we argue below, for the code generation phase in a compiler backend, we could theoretically state the overall transformation function but simplify the translation and the checking task significantly if we regard the sequence of single transformations. In this context, we also have the problem that intermediate results of the translation are only available in main memory. Hence, we need a tight interlocking between compiler and checker. The second problem, namely incomplete specifications, also appears when using compiler generators. Typically, compiler generators are maintained over a long period of time. Many functionalities are implemented but the generation of one particular compiler backend needs only a few of them. It would be much too expensive to verify compiler backend generators. This verification task could even be impossible if there existed different functionalities whose combination would result in incorrect compilers, even though such a combination was never chosen in practice. Here program checking as a cheaper

alternative is a good compromise. It does not prove total correctness but gives us formal correctness proofs for particular instances, i.e. compiler runs.

The backend of a compiler transforms the intermediate representation into target machine code. This task involves code selection, register allocation, and instruction scheduling. These problems are optimization problems that do not have a unique solution but rather a variety of them, particularly distinguished by their quality. Many of these problems are **NP**-hard. Hence, algorithms for code generation are often search algorithms trying to find the optimal solution or at least a solution as good as possible wrt. certain time or space constraints. If one wants to prove the implementation correctness of such algorithms, it is not necessary to prove the quality of the computed solutions. It suffices to prove their correctness. (If the optimality of the solution also belonged to the specification, we would need to extend the checking task by additionally testing if the quality of the computed solution is good enough.) In our case study, we have designed and implemented checkers for the code generation phase in compiler backends. In subsection 4.1, we describe code generation based on bottom-up rewrite systems (BURS) and the CGGG system [Boe98] (compiler generator generator based on graphs). In subsection 4.2, we present a generic checking algorithm for BURS code generation and the necessary modifications of the CGGG system.

4.1 BURS and the CGGG System

Bottom-up rewrite systems (BURS) are a powerful method to generate target machine code from intermediate program representations. Conventional BURS systems allow for the specification of transformations between terms which are represented as trees. Rules associate tree patterns with a result pattern, a target-machine instruction, and a cost. If the tree pattern matches a subtree of the intermediate program representation, then this subtree can be replaced with the corresponding result pattern while simultaneously emitting the associated target-machine instruction. The code generation algorithm determines a sequence of rule applications which reduces the intermediate program tree into a single node by applying rules in a bottom-up order.

Traditionally, BURS has been implemented by code generation algorithms which compute the costs of all possible rewrite sequences. This enormous computation effort has been improved by employing dynamic programming. The work by Nymeyer and Katoen [NK97] enhances efficiency further on by coupling BURS with the heuristic search algorithm A^* . This search algorithm is directed by a cost heuristic. It considers the already encountered part of costs for selected code as well as the estimated part of costs for code which has still to be generated. A^* is an optimally efficient search algorithm. No other optimal algorithm is guaranteed to expand fewer nodes than A^* , cf. [DP85]. Using such an informed search algorithm offers the advantage that only those costs need to be computed that might contribute to an optimal rewrite sequence. [NK97] propose a two-pass algorithm to compute an optimal rewrite sequence for a given expression tree. The first bottom-up pass computes, for each node, the set of all possible local rewrite sequences, i.e. those rewrite sequences which might be applicable at that

node. This pass is called *decoration* and the result is referred to as *decorated tree*. The second top-down pass trims these rewrite sequences by removing all those local rewrite sequences that do not contribute for the reduction of the term.

Static single assignment (SSA) form [CFR⁺91, CF95] has become the preferred internal program representation for handling all kinds of program analyses and optimizing program transformations prior to code generation. Its main merits comprise the explicit representation of def-use-chains and, based on them, the ease by which further dataflow information can be derived. By definition SSA-form requires that a program be represented as a directed graph of elementary operations (jump, memory read/write, unary or binary operation) such that each “variable” is assigned exactly once in the program text. Only references to such variables may appear as operands in operations. Thus, an operand explicitly indicates the data dependency to its point of origin. The directed graph of an SSA-representation is an overlay of the control flow and the data flow graph of the program. If a variable x has several static predecessors x_1, \dots, x_n , one of which defines the value of x at runtime, this is expressed by $x := \phi(x_1, \dots, x_n)$. This value is a selection amongst the values x_1, \dots, x_n and represents the unique value assigned to variable x at runtime.

BURS theory in an extended form [Boe98] can handle SSA representations by a two-stage process [Boe98]. The first stage concerns the extension from terms, i.e. trees, to terms with common subexpressions, i.e. DAGs. This modification involves mostly technical details in the specification and implementation of the rewrite rules. The second stage deals with the extension from DAGs to potentially cyclic SSA graphs. SSA graphs might contain data and control flow cycles. There are only two kinds of nodes which might have backward edges, ϕ nodes and nodes guiding the control flow at the end of a basic block to the succeeding basic block. For these nodes, one can specify general rewrite rules which do not depend on the specific translation, i.e., which are independent from the target machine language. In a precalculation phase, rewrite sequences are computed for these nodes with backward edges. These rewrite sequences contain only the general rewrite rules. In the next step, the standard computation of the rewrite sequences for all nodes in the SSA graph is performed. Thereby, for each node with backward edges, the precalculated rewrite sequences are used.

The BURS code generation algorithm has been implemented in the code generator generator system CGGG (code generator generator based on graphs) [Boe98]. CGGG takes a specification consisting of BURS rewrite rules as input and generates a code generator which uses the BURS mechanism for rewriting SSA graphs, cf. Fig. 4. The produced code generators consist of three major parts. First the SSA graph is decorated by assigning each node the set of its local alternative rewrite sequences. Then the A^* -search looks for the optimal solution, namely the cheapest rewrite sequence. This search starts at the final node of the SSA graph marking the end of computation, by working up through the SSA graph until the start node is reached. Finally, the target machine code is generated by applying the computed rewrite sequence.

An example for a rule from a code generator specification is:

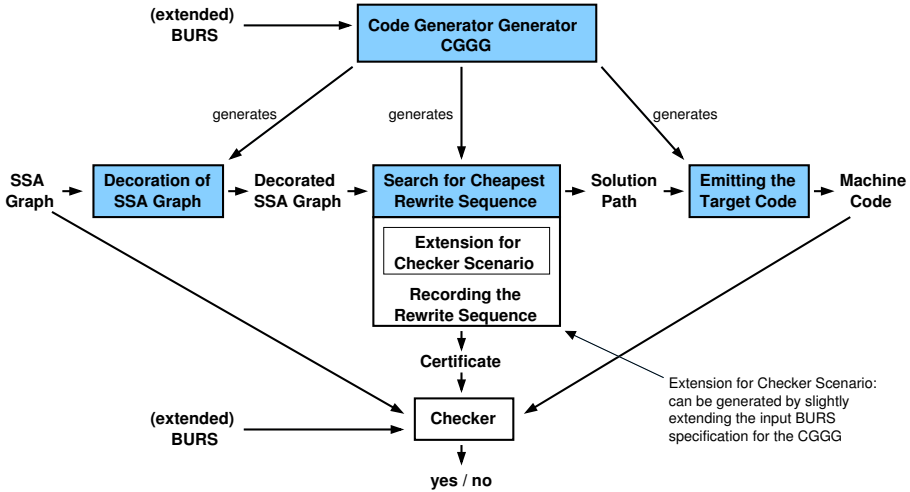


Fig. 4. Extended CGGG Architecture

```

RULE a:Add (b:Register b) -> s:Shl (d:Register c:Const);
RESULT d := b;
EVAL { ATTR(c, value) = 1; }
EMIT {}
    
```

This rule describes an addition of two operands. On the left hand side of the rule, the first operand is a register with short name *b*. The second operand is the first operand again, identified by the short name. Note that the left-hand side of this rule is a directed acyclic graph. If the code generator finds this pattern in the graph, it rewrites it with the right-hand side of the rule. In general, this could be a DAG again. Thereby the `EVAL` code is executed. This code places the constant 1 in the attribute field `value` of the `Const` node. The `RESULT` instruction informs the register allocator that the register *d* equals register *b*.

Optimal BURS code generation for SSA graphs is an **NP**-complete problem: In [AJU77], it is shown that code generation for expressions with common subexpressions is **NP**-complete. Each instance of such a code generation problem is also a BURS code generation problem for SSA graphs. Thus it follows directly that BURS code generation for SSA graphs is **NP**-complete.

4.2 A Generic Checker for Code Generators

The CGGG architecture can easily be extended for the program checking approach. Therefore we record which sequence of rewrite rules has been selected during the *A**-search for the cheapest solution. This sequence of rewrite rules is the certificate. The checker takes it and recomputes the result. This result is compared with the result of the code generator. Only if it is equal to that of the checker, the checker will output ‘yes’. If the checker outputs ‘no’, then it

has not been able to recompute the same result. Such a checker is generic in the sense that the respective BURS system is one of the checker inputs. Hence, the same generic checker can be used for all code generators generated by the CGGG system.

It is particularly easy to extend the CGGG architecture such that it outputs the certificate necessary to check the results of the generated code generators. We can extend the BURS specification such that not only the machine code is output but also, in a separate file, the applied rules. Therefore, we only need to extend the EMIT part of each rule. This part contains instructions which will be executed on application of the rule. We can place one more instruction there, namely a protocol function. This protocol function writes a tuple to the certificate file. This tuple contains the applied rule as well as the node identifier of the node where it has been applied. We have decided to take the address in main memory of each node as its unique identifier.

One might ask why it is not sufficient to check only the local decorations of the nodes on the solution path found during the A^* -search to ensure the correctness of the computed result. The answer concerns the sorts of the nodes in the SSA graph. Each node has a specific sort which might be changed on application of a rule. Hence, the correctness of a rule sequence can only be decided if one makes sure that the sorts of the nodes and the sorts required by the rules fit together in each rule application. Moreover, one needs to check that the rules are applied according to the bottom-up strategy of BURS. We do not know of any other checking method assuring well-sorting and bottom-up rewriting other than recomputing the solution. This problem is also an example for the first practical problem discussed in subsection 3.1. The transformation of an intermediate representation into its target code is not specified by a single input-output mapping but rather by a sequence of rule applications. For all theoretical as well as practical purposes, it would be much too complicated to express such rule-based transformations by a single function.

The exact checking algorithm is summarized in Fig. 5. The certificate *Certificate* is a list of tuples, each containing a rule and the node identifier *node_no*. This node identifier characterizes uniquely the node at which the rule has been applied. *BURS* is the extended rewrite system which the CGGG has taken as input. *SSA_Graph* is the intermediate representation or the intermediate results obtained during the rewrite process, resp. Finally, *Target_Code* is the result of the rewrite process, the machine instruction sequence. To keep the presentation of the checking algorithm as simple as possible, we did not give all details of the auxiliary procedures but only described them colloquially. Clearly, this checker is generic because the respective BURS system is not hardwired into its code but one of the input parameters.

Theorem 1 (Correctness of Checker). *If the checker outputs ‘yes’ (True) on input (BURS, SSA_Graph, Target_Code, Certificate), then the target code Target_Code has been produced correctly by transforming the intermediate representation SSA_Graph according to the rules of the BURS system BURS. \diamond*

```

proc CGGG_Cheker(BURS, SSA_Graph, Target_Code, Certificate) : Bool;
  var Checker_Code : list of strings;
  Checker_Code := [];
  while Certificate ≠ [] do
    (rule, node_no) := head(Certificate);
    Certificate := tail(Certificate);
    if rule ∉ BURS then return False;
    SSA_Graph := apply_and_check(rule, SSA_Graph, node_no);
    insert(code(rule), Checker_Code); od;
  return compare(Checker_Code, Target_Code) end
proc apply_and_check(rule, SSA_Graph, node_no) : Bool;
  node := find_node(SSA_Graph, node_no);
  if node = Nil then return False;
  if BURS.successors(SSA_Graph, node_no) ≠ ∅ then return False;
  if not match(lhs(rule), node, SSA_Graph) then return False;
  apply(rule, SSA_Graph, node_no); end;
proc find_node(SSA_Graph, node_no) :
  returns node in SSA_Graph with number node_no;
  if node does not exist, it returns Nil; end
proc BURS.successors(SSA_Graph, node_no) :
  returns set of nodes in SSA_Graph that have to be rewritten before node
  node_no if bottom-up rewrite strategy is used;
  control and data flow cycles are disconnected as in the A*–search; end
proc lhs(rule) : returns left-hand side of the rewrite rule rule; end
proc match(pattern, node, SSA_Graph) :
  checks if pattern matches subgraph located at node; end
proc apply(rule, SSA_Graph, node_no) : does the rewrite step; end
proc compare(Checker_Code, Target_Code) :
  checks if Checker_Code and Target_Code are identical; end
proc code(rule) : returns code associated with rule rule; end

```

Fig. 5. Generic Checker for Code Generation

Proof. The CGGG system is supposed to generate code generators implementing the respective input rewrite system BURS. To check if a code sequence produced by such a code generator is correct, we need to make sure that there is a sequence of rule applications conforming to the BURS rewrite method. Instead of testing if there is any such sequence, we check the weaker proposition if the certificate produced by the code generator is a BURS rewrite sequence. This is done successively by repeating each rule application, starting with the same SSA intermediate representation. In each step, it is tested that the node exists at which the rewrite step is supposed to take place. Then it is tested that the rewrite step conforms to the bottom-up strategy of BURS. Finally, the left-hand side of the rule must match the graph located at the respective node. If all three requirements are fulfilled, then the rewrite step is performed by the checker. If this recomputation of the target machine code results in exactly the same code sequence, then the result of the code generator has been validated. If we verify

the checker wrt. the requirements listed in this proof, then we have a formally verified correct result of the code generation phase. ■

5 Experimental Results

The computations performed in the checker for the CGGG do the same rewrite steps as the backend itself and return ‘False’ if an error occurs. The only difference between checker and backend lies in the search for the optimal solution. The checker gets it as input for granted while the backend needs to compute it by an extensive search. In this section, we explain why this observation seems to be general for a variety of optimization problems and how we have exploited it in our checker implementation. Then we state our experimental results. Thereby we also explain how we have dealt with the practical problems of only implicitly stated input-output mappings and incomplete or nonexistent specifications (cf. section 3) by the proposed iterative method of assuming, testing, extending and eventually verifying a specification.

5.1 The Nature of NP-Problem Checkers

Problems in **NP** are characterized by the fact that a proof for the correctness of a solution has polynomial length. This holds in particular for the decision variant of many optimization problems. In general, it is unknown how such a proof looks like. For many practical problems such proofs are natural, cf. section 2. This observation has direct implications concerning the implementation of checkers for such optimization problems with natural proofs. The actual implementation keeps track of its decisions and collects them in its certificate. This certificate is the proof for the correctness of the computed solution. Based on it, the checker recomputes the solution and compares it with the solution of the implementation. Only if both solutions are identical, the checker outputs ‘yes’. Speaking in the language of Turing machines, the problem implementation is a nondeterministic Turing machine that needs good random guesses to find a solution. The checker is a deterministic Turing machine that knows the good guesses (the certificate as its input) and just needs to recompute the solution. Hence, we can expect that the checker implementation is a part of the overall implementation of the optimization problem.

5.2 Experimental Results

For BURS code generation, this expectation has come true. We could extract most of the code for the checker implementation from the code generator implementation directly. This is an advantage since CGGG has been tested extensively, making sure that many obvious bugs have been eliminated from the (implementation and checker) code already in the forefront of our experiment. CGGG has been used during the last four years by many graduate students who tend to be very critical software testers. Moreover, the CGGG system has

been utilized to generate a compiler in the AJACS project (Applying Java to Automotive Control Systems) with industrial partners [Gau02, GKC01]. This compiler transforms restricted Java programs into low-level C code.

We can distinguish three different kinds of code in the CGGG system:

1. Code that does not have any influence on the correctness of the results at all. This comprises in particular all debugging functionalities. This code does not need to be verified.
2. Code that implements the search for the optimal solution. This code needs to be extended by the protocol function which sets up the certificate. This code does not need to be verified.
3. Code that computes the rewrite steps. In a slightly extended form, this code becomes part of the checker and needs to be verified in order to get formally correct results of code generation.

	Code Generator	Checker
lines of code in .h-Files	949	789
lines of code in .c-Files	20887	10572
total lines of code	21836	11361

Fig. 6. Size of Code Generator and Checker

In our case study, we implemented a checker for the AJACS compiler described above. The table in Fig. 6 compares the overall size of the AJACS code generator generated by the CGGG system with the size of its checker. Both implementations, CGGG and the code generator, are written in C.

If one wants to obtain formally verified solutions for the code generation phase, one needs to verify only the checker code. A first comparison between the size of the code generator and its checker shows that the verification effort for the checker seems to be half of that of the code generator. This comparison is only half the truth as the verification effort is even smaller. Much of the checker code is generated from the code generator specification. This is very simple code which just applies the rewrite rules. The verification conditions for the various rewrite rules are basically the same, simplifying the verification task considerably. In contrast, the code for the A^* -search is very complicated and would need much more verification effort. Luckily it does not belong to the checker. Up to now we have not formally verified the checker code. For this task it seems helpful to parameterize the rewrite routines with the respective rewrite rules. In doing so, it would suffice to only formally verify the parameterized rewrite routine.

We designed and implemented the checker iteratively, as described in section 3. We started with the assumption that it takes SSA representations and transforms them directly by applying the graph rewrite rules of the compiler backend specification. It turned out that this assumption was not detailed enough. Instead, the compiler first transforms the SSA form into a slightly different representation which removes some of the data flow freedom of the SSA form in order to simplify the graph rewrite process. This is a typical example for the practical

problem that the input-output mapping is not stated explicitly, cf. subsection 3.1. So we stated an intermediate assertion (cf. subsection 3.2 and in particular Fig. 3) about the result of this first auxiliary transformation. Then we proceeded by stating the next assertion that the result of the graph rewrite process, i.e. the target program, has been received by exclusively applying the rewrite rules of the specification. It turned out that this assumption was false. The CGGG system additionally uses two general rewrite rules to handle the data and control flow cycles in the SSA representations. These rules are not stated in the specification but hard-coded into the generated compiler backends because they are independent of the target language and needed in every backend. This is a classical example of an incomplete specification, the second major practical problem discussed in subsection 3.1. We extended the specification of our checker with these general rewrite rules and did not find any more inconsistencies.

6 Related Work

The original notion of program checking [BK95] assumes that an implementation is used as a black box and that an independent program result checker checks the correctness of each individual result. Black-box program checking has been applied to numerical problems [BLR93]. In our extended setting, we allow the checker to explicitly access the implementation code and to receive a certificate as input which records the run-time decisions of the implementation. Our checkers can use the certificate to recompute the solution. By employing concepts from computational complexity theory, we show that for many search and optimization problems, it is natural that the computations of the checker are a part of the computations of the implementation. So program checking with certificates becomes a method of separating the correctness-critical part of a given implementation. Our case study raises hope that the checker computations are those that can be verified more easily, cf. section 5.

Safety-scalable program checking is a good middle course between formal program verification and pure testing. Program verification is often too expensive since proofs are often longer than the program itself. Small changes in the program code require the proof to be redone. In contrast, testing does not give full reliability as it does not say anything about special inputs not included in the test suites. It is very difficult to decide if a test sample distribution is sufficient to predict that no errors will occur at run time. In our approach of safety-scalable program checking, we do not lose the advantages of program testing, which help in understanding existing software and in finding early mistakes.

In our case study, we have experienced safety-scalable program checking as a novel combination of existing methods. It combines code inspection, design by contract, testing, and formal program verification, cf. section 3.2. Code inspection is necessary to create the postulates. We employ design by contract [Mey97] by postulating assertions for certain points during program execution, cf. the assertions in Eiffel programs to be checked during runtime. To ensure the formal correctness of the implemented checkers, we need formal program verification.

Program checking has been used in the construction of correct compilers, most prominently in the Verifix project [GDG⁺96]. It has proposed program checking to ensure the correctness of compiler implementations. Program checking has been successfully applied in the context of frontend verification [HGG⁺99]. [GHZG99, GZG00] propose program checking to ensure the correctness of backend implementations but do not have a checking algorithm. The program checking approach has also been used in further projects aiming to implement correct compilers. [Nec00] shows how some backend optimizations of the GCC can be checked. Proof-carrying code [NL97, NL98, CLN⁺00] is another weaker approach to the construction of correct compilers which guarantees that the generated code fulfills certain necessary correctness conditions. During the translation, a correctness proof for these conditions is constructed and delivered together with the generated code. A user may reconstruct the correctness proof by using a simple proof checking method. In recent work [NR01], a variant of proof-carrying code has been proposed which is related to our notion of program checking with certificates. In this setting, trusted inference rules are represented as a higher-order logic program, the proof checker is replaced by a nondeterministic higher-order logic interpreter and the proof by an oracle implemented as a stream of bits that resolve the nondeterministic choices. This proof directly corresponds to our notion of certificate as it helps in resolving the nondeterminism in the same way as in our setting. Nevertheless, this work does not draw the same conclusion as we do, namely that checking with certificates isolates the correctness-critical part of an implementation. In [PSS98b, PSS98a], the problem of constructing correct compilers is also addressed, but only for very limited applications. Only those programs consisting of a single loop with loop-free body are considered and translated without the usual optimizations of compiler construction. Those programs are translated correctly such that certain safety and liveness properties of reactive systems are sustained. In more recent work [ZPL01], a theory for validating optimizing compilers is proposed similar to the method developed in the Verifix project. The main difference to our work is that these approaches do not assume to have access to the implementation of the compiler or its generator. This access gives us the freedom to modify the implementation to get a certificate used in the checker.

7 Conclusions

We have shown that program checking with certificates is a safety-scalable method for real-scale practical applications, i.e., its reliability is scalable wrt. safety. We deal with the practically relevant problem of incomplete or missing specifications of software with an iterative process. We postulate a specification, implement checkers for it, and iteratively improve it by testing it with typical input and output values. Since the checkers always reject results not fitting to the specification, we never get incorrect results. This gives us a significant advantage over pure testing and an increased confidence in the correctness of a

given piece of software. If we need reliability on the highest possible level, we need to verify the checker implementation with an automated theorem prover. Such a verification ensures the formally proved correctness of the results.

We have extended the classical notion of black-box program checking to program checking with certificates which allows the checker to access the implementation code. The checker might observe the implementation and might receive a certificate recording the computation steps of the implementation. The checker might use this certificate to check the computed solution, typically by recomputing it. This scenario is especially suited for search and optimization problems, in particular for **NP**-complete problems. Thereby we use the property of these problems that results can be checked in polynomial time whereas the computation of results is believed to take much more time. As a practical consequence, the checker code is nearly identical with some parts of the implementation code: Many search and optimization problems can be solved by algorithms looking for an optimal solution. These algorithms have typically a fraction searching for an optimum and a fraction computing the respective solution. The checker is not concerned with the quality of the solution and only needs to recompute it via a trusted (i.e. validated or verified) implementation. Hence, program checking with certificates arises as a method to separate the correctness-critical part of a given implementation. We have tested our method with a system consisting of approximately 20.000 lines of code. The size of our checker is about half as much. It remains an open question how this ratio scales up for larger software.

In our case study, we have considered compilers and their generators, in particular code generators based on rewrite systems. For this problem, we could separate the search and the computation part. The computation part becomes the major part of the checker and eventually needs to be formally verified. We are convinced that the implementations for most optimization problems can be partitioned in the same way. In our experiments, safety-scalable program checking has appeared as a novel combination of well-known techniques as e.g. code inspection, design by contract, testing, and program verification. It is applicable also in many other software engineering areas. Compiler technology is a core methodology for automatically handling all kinds of program and data transformations. Hence, there are many practical problems which can be treated with compiler technology, e.g. XML processing, in general the adaptation of data formats, design patterns in software engineering, software maintenance, software components, component adaptation, meta programming, etc. In future work we want to apply safety-scalable program checking to these areas.

Acknowledgment

The author would like to thank Gerhard Goos, Wolf Zimmermann, Boris Boesler, Götz Lindenmaier, and Florian Liekweg for many helpful discussions. Moreover, thanks to Jan Olaf Blech for implementing the checker for the AJACS compiler. Finally, thanks to the anonymous reviewers for many helpful comments.

References

- AJU77. A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.
- BK95. Manuel Blum and Sampath Kannan. Designing Programs that Check Their Work. *Journal of the ACM*, 42(1):269–291, 1995. Preliminary version: *Proceedings of the 21st ACM Symposium on Theory of Computing (1989)*, pp. 86–97.
- BLR93. Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993. Preliminary version: *Proceedings 22nd ACM Symposium on Theory of Computing (1990)*, pp. 73–83.
- Boe98. Boris Boesler. Codeerzeugung aus Abhängigkeitsgraphen. Diplomarbeit, Universität Karlsruhe, June 1998.
- CF95. R. Cytron and J. Ferrante. Efficiently Computing Φ -Nodes On-The-Fly. *ACM Transactions on Programming Languages and Systems*, 17(3):487–506, 1995.
- CFR⁺91. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- CLN⁺00. Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A Certifying Compiler for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107, Vancouver, British Columbia, Canada, May 2000.
- DP85. Rina Dechter and Judea Pearl. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, 32(3):505–536, July 1985.
- Gau02. Thilo Gaul. AJACS: Applying Java to Automotive Control Systems. *Automotive Engineering Partners*, 4, August 2002.
- GDG⁺96. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F.W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In P. Fritzson, editor, *Poster Session of CC'96*. IDA Technical Report LiTH-IDA-R-96-12, Linköping, Sweden, 1996.
- GHZG99. Thilo Gaul, Andreas Heberle, Wolf Zimmermann, and Wolfgang Goerigk. Construction of Verified Software Systems with Program-Checking: An Application to Compiler Back-Ends. In *Proceedings of the Workshop on Runtime Result Verification (RTRV'99)*, 1999.
- GKC01. Thilo Gaul, Antonio Kung, and Jerome Charousset. AJACS: Applying Java to Automotive Control Systems. In Caspar Grote and Renate Ester, editors, *Conference Proceedings of Embedded Intelligence 2001, Nürnberg*, pages 425–434. Design & Elektronik, February 2001.
- GZG00. Thilo Gaul, Wolf Zimmermann, and Wolfgang Goerigk. Practical Construction of Correct Compiler Implementations by Runtime Result Verification. In *Proc. SCI'2000, International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, USA, 2000.

- HGG⁺99. Andreas Heberle, Thilo Gaul, Wolfgang Goerigk, Gerhard Goos, and Wolf Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In D. Björner, M. Broy, and A.V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99*, pages 493–502, Akademgorodok, Novosibirsk, Russia, July 1999. Springer Verlag, Lecture Notes in Computer Science, Vol. 1755.
- Mey97. Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.
- Nec00. George C. Necula. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 83–94, Vancouver, British Columbia, Canada, May 2000.
- NK97. A. Nymeyer and J.-P. Katoen. Code generation based on formal BURS theory and heuristic search. *Acta Informatica* 34, pages 597–635, 1997.
- NL97. George C. Necula and Peter Lee. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997.
- NL98. George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Quebec, Canada, May 1998.
- NR01. George C. Necula and S. P. Rahul. Oracle-Based Checking of Untrusted Software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, London, UK, January 2001.
- Pap94. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
- PSS98a. A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt.). *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.
- PSS98b. A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In B. Steffen, editor, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, April 1998. Springer Verlag, Lecture Notes in Computer Science, Vol. 1384.
- ZPL01. L. Zuck, A. Pnueli, and R. Leviathan. Validation of Optimizing Compilers. Technical Report MCS01-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, August 2001.