Pervasive Compiler Verification – From Verified Programs to Verified Systems

Dirk Leinenbach[1,2,3]

German Research Center for Artificial Intelligence (DFKI)
P.O. Box 15 11 50
66041 Saarbrücken, Germany

Elena Petrova[1,4]

Saarland University, Computer Science Dept.
P.O. Box 15 11 50
66041 Saarbrücken, Germany

Abstract

We report in this paper on the formal verification of a simple compiler for the C-like programming language $C0$. The compiler correctness proof meets the special requirements of pervasive system verification and allows to transfer correctness properties from the $C0$ layer to the assembler and hardware layers. The compiler verification is split into two parts: the correctness of the compiling specification (which can be translated to executable ML code via Isabelle's code generator) and the correctness of a $C0$ implementation of this specification. We also sketch a method to solve the boot strap problem, i.e., how to obtain a trustworthy binary of the $C0$ compiler from its $C0$ implementation. Ultimately, this allows to prove pervasively the correctness of compiled $C0$ programs in the *real* system.

Keywords: Compiler Verification, Theorem Proving, System Verification, HOL, Hoare Logic

1 Introduction

The Verisoft project aims at the pervasive formal verification of computer systems comprising hardware (the verified VAMP processor [7,12] and devices [1,18]), system software [15], and applications [5]. 'Pervasive' means to prove a single, integrated correctness theorem for the whole system instead of verifying separate properties for each layer without justification that they *formally* fit together (cf. [30]).

Except for very small parts of the system level software, software in Verisoft is implemented in the C-like programming language $C0$. This language has been

[1] Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project (http://www.verisoft.de) under grant 01 IS C38.

[2] Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

[3] Email: Dirk.Leinenbach@dfki.de

[4] Email: petrova@wjpserver.cs.uni-sb.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

designed to be expressive enough to allow implementation of low-level software while—at the same time—being 'neat' to allow for efficient formal verification of medium-sized $C0$ applications. However, pervasive verification does not stop at the $C0$ level. To allow execution of verified programs on the *real* hardware they must be compiled to binary code. This translation could itself introduce errors into an otherwise verified $C0$ program. Thus, verification of the translation process is essential for pervasive system verification when using a high-level programming language. Furthermore, the formulation of the compiler correctness statement has to be adequate for pervasive verification [24].

In order to bridge the gap between verified software and verified hardware, we have defined a *compiling specification* for a $C0$ compiler in Isabelle / HOL [35] and additionally *implemented* the compiler in $C0$. Both the compiling specification and its implementation have been formally verified [23,38]. For the latter we have shown using a $C0$ verification environment [40] that it produces the same list of assembler instructions as specified by the compiling specification. For the former we have verified a small-step simulation theorem, which states that the original $C0$ program and the compiled code behave equivalently. This theorem respects resource restrictions (e.g., bounded memory size) of the target machine and permits to discharge them at the $C0$ level. That the theorem is formulated in a small-step manner allows to argue about interleaving and non-terminating computations.

This paper is supposed to give an overview of the compiler verification efforts in Verisoft. For more details and precise formal definitions see [23,38].

1.1 Requirements Analysis and Related Work

Compiler verification is a well established field [13]. There are correctness proofs covering issues from simple expression translation in [27] to compilers with optimizations in [8,25]. Also, different source languages are considered: from toy languages to subsets of C [25] and Java [41] or the Java virtual machine [22].

In the Verifix project [14], impressive work concerning correct compilers has been done. In [44], the authors present an elegant theory for the translation of intermediate languages to machine languages; the work was partially formalized in the PVS theorem prover. The implementation of a compiler for ComLisp (a subset of Common Lisp) was verified on the machine code level by a manual check [16].

Recently, Leroy et al. have formally verified an optimizing two-step translation from Clight (a subset of C) first to the intermediate language C Minor and then to PowerPC assembler [8,25]. The proof in the Coq proof assistant is based on big-step semantics of the source and target languages. An executable compiler was obtained by automatic (unverified) extraction from the Coq specification.

However, a compiler correctness theorem to be used for pervasive system verification has to meet extra requirements. We highlight the most important ones.

Language Model

C0 is a sequential language and even the target machine is a uni-processor architecture. So, sequential reasoning, big-step semantics, and classical Hoare logics seem to be adequate. But interleaving and non-terminating system software as well

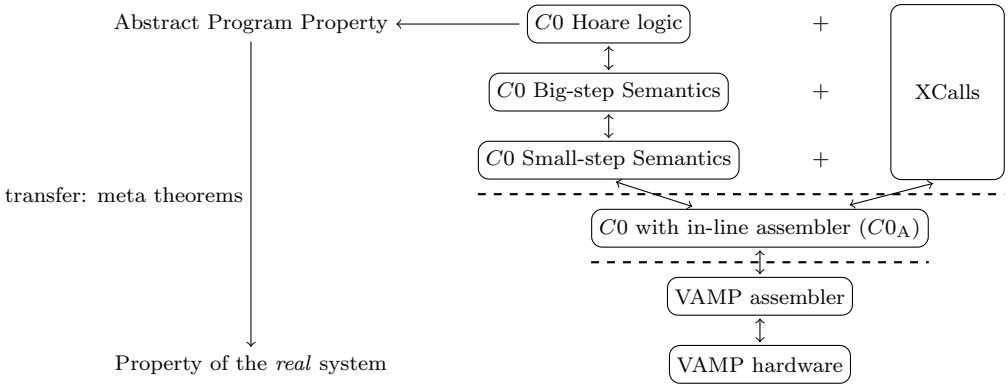

Fig. 1. Semantics Layers in Verisoft

as interrupt driven devices demand a concurrent model [2,18]. Small-step semantics and a small-step compiler correctness theorem are appropriate to handle this.

Compiler correctness proofs w.r.t. small steps semantics exist on paper [26,34]. But the proofs are usually carried out 'big step style' by a straightforward induction over the syntax tree. This works only for terminating programs. In our context it is *much* more comfortable to work with a compiler correctness statement in the form of a small-step simulation theorem as it has been done for a back-end in [44].

The Verisoft project uses several semantical layers to base reasoning on the right abstraction level [2]. This increases efficiency—when using the more abstract layers—while still allowing formulation and verification of detailed, concrete properties on the lower layers. Figure 1 depicts this stack. Results from the higher layers have to be formally transferred to the lower layers using meta theorems. Finally, this yields a single correctness theorem for the complete system. To support $C0$ programs which invoke in-line assembler code in the Hoare logic, we formalize the effect of the in-line assembler parts axiomatically using so-called *XCalls* [2]. Their implementation has to be plugged in at the level of the $C0_{\mathrm{A}}$ semantics which combines $C0$ with in-line assembler.

Pervasiveness

In [9] the specification of an optimizing compiler back-end from the SSA intermediate language has been formally verified. However, the machine model used there is not the language of a realistic processor and hence the work does not suffice to bridge the gap between software and hardware for pervasive verification. On the other hand, the work from [43] describes a framework for modeling the semantics of expression evaluation including non-determinism in the evaluation order. In the context of pervasive verification, such complicated languages are not desirable as they make correctness proofs of larger programs infeasible.

Pervasive verification has to handle resource restrictions on the target machine. Our compiler correctness theorem incorporates these restrictions and allows to discharge them at the C0 rather than at the assembler level which simplifies reasoning and increases productivity. The small-step character of our simulation theorem allows to easily argue about resource restrictions also for intermediate states.

The famous CLI project [6] resulted in a stack of verified components including a

compiler specification. The produced collection of verified programs has mostly been done in low-level languages. Recently, Zhong Shao [33] presented very nice logics for the assembler level verification of different kinds of low-level software. However, to allow for the *efficient* verification of medium-sized applications we have to use a high-level implementation language.

Early papers consider only verification of a compiling specification rather than verification of its implementation, although in [11,29] the authors already pointed out the necessity of the implementation correctness proof. Later, Goerigk et al. added a new aspect of compiler correctness, namely the *bootstrapping problem*, i.e., generation of the first trustworthy executable of a verified compiler [16].

Integration of Solutions

As pointed out above, there are many additional challenges for compiler verification due to pervasive verification. Some of them have been solved (in isolation) in a similar or even more general way in other work. However, in the context of pervasive verification an essential part of the verification effort has to be invested in the combination of the individual solutions into a single framework. In addition to the impressive work of the CLI stack project [6], early work from Joyce [21] discusses problems imposed by the formal combination of a verified compiler with verified hardware. To the best of our knowledge, the work presented in this paper is the first which integrates all the separate solutions into a single framework that provably [2] meets the needs of pervasive verification of complex systems.

1.2 Outline

The remainder of this paper is structured as follows. In Section 2, we introduce the $C0$ language and sketch its small-step semantics. We present a simulation theorem for the compiling specification in Section 3 and a correctness proof for the compiler implementation in Section 4. The section about the correct compiler implementation contains a sketch of our approach to solve the bootstrap problem. We conclude in Section 5 and discuss some future work.

2 The $C0$ Language

Semantics of the full C language are complex [17,36,37] and the use of all features of C leads to an error-prone programming style [31]. In contrast, formal verification of programs is easier and more efficient for programming languages with concise semantics. Verisoft uses the C-like imperative language $C0$ which has sufficient features to implement all system and application software in Verisoft while still allowing for efficient verification of programs with several thousand lines of code.

$C0$ has several limitations compared to standard C [20]; we list the most important ones. Side effects in expressions are not allowed, which forbids in particular function calls as subexpressions and requires a special function call statement. Pointers are typed and must not point to local variables or to functions; void pointers and pointer arithmetic are not supported. Arrays have to be of fixed size and

$$\begin{aligned} ty \;=\; & \mathit{Bool}_\mathrm{T} \mid \mathit{Int}_\mathrm{T} \mid \mathit{Char}_\mathrm{T} \mid \mathit{Unsigned}_\mathrm{T} \\ & \mid \mathit{Str}_\mathrm{T}(\mathbb{S} \times \mathit{ty\ list}) \mid \mathit{Arr}_\mathrm{T}(\mathbb{N}, \mathit{ty}) \mid \mathit{Ptr}_\mathrm{T}(\mathbb{S}) \mid \mathit{Null}_\mathrm{T} \end{aligned}$$

Fig. 2. Data Type *ty* for *C*0 Types

$$\begin{aligned} \mathit{expr} \;=\; & \mathit{Lit}(\mathit{lit}) \mid \mathit{Var}(\mathbb{S}) \mid \mathit{Arr}(\mathit{expr}, \mathit{expr}) \mid \mathit{Str}(\mathit{expr}, \mathbb{S}) \\ & \mid \mathit{UnOp}(\mathit{unop}, \mathit{expr}) \mid \mathit{BinOp}(\mathit{binop}, \mathit{expr}, \mathit{expr}) \\ & \mid \mathit{LazyBinOp}(\mathit{lazyop}, \mathit{expr}, \mathit{expr}) \\ & \mid \mathit{AddrOf}(\mathit{expr}) \mid \mathit{Deref}(\mathit{expr}) \end{aligned}$$

Fig. 3. Data Type *expr* for *C*0 Expressions

are represented by a separate type in *C*0. Low-level data types (unions or bit fields) and control flow statements (switch, goto, long jumps) are not supported.

*C*0 supports four *basic* types: booleans, 32-bit signed integers, 32-bit natural numbers, and 8-bit signed integers. Pointers, fixed size arrays, and structures are supported as *aggregate* types (cf. Figure 2). Pointer types do not directly include the type to which the point; instead, we use an additional indirection via *type names*. This allows the definition of self-referencing pointer types (e.g., a list component type whose 'next' field is a pointer to the component type). The mapping from type names to types is handled via a so-called type name environment (cf. Section 2.1). Observe, that there exists a special type for null pointer constants. *Elementary* types comprise basic types and pointers.

Variable names and literals are expressions. If e and i are expressions and cn is a component name, then array access $e[i]$, access to structure components $e.cn$, dereferencing $*e$, and the 'address-of' operator $\&e$ are also expressions. Additionally, *C*0 supports the usual unary and binary operators. In Figure 3, we give a formal definition of the data type *expr* which models *C*0 expressions in Isabelle.

*C*0 statements are modeled in Isabelle via the data type *stmt* (cf. Figure 4). Observe, that statements of a *C*0 program are annotated with unique identifiers of type *sid* (which is isomorphic to the natural numbers). These identifiers allow us to map statements occurring in the dynamic program rest to the original statements in the function table of a *C*0 program and to determine the function they belong to and their relation to other statements of the program.

In the following, let s and e (with arbitrary subscripts) denote statements and expressions. Besides sequential composition $\mathit{comp}(s_1, s_2)$, while loops $\mathit{while}(e, s)$, conditional statements $\mathit{if}(e, s_1, s_2)$, and the empty statement *skip*, *C*0 supports the following statements.

Assignments come in two flavors. *Normal* assignments $\mathit{ass}(e_l, e_r)$ copy the value of one expression to another. Unlike standard C, *C*0 supports assignments of arbitrary aggregate types.[5] *Complex* assignments $\mathit{ass}_\mathrm{C}(e_l, l_c)$, which assign a *complex*

[5] In addition to the rather restricted assignments of structures in C90, the C99 standard supports assignments like `x = (struct s){.n1 = e1, n2 = e2};`. However, C restricts this kind of assignments to initializers.

$$
\begin{aligned}
\mathit{stmt} \;=\; & \mathit{skip} \mid \mathit{comp}(\mathit{stmt}, \mathit{stmt}) \\
& \mid \mathit{ass}(\mathit{expr}, \mathit{expr}, \mathit{sid}) \mid \mathit{ass}_{\mathrm{C}}(\mathit{expr}, \mathit{lit}_{\mathrm{c}}, \mathit{sid}) \mid \mathit{new}(\mathit{expr}, \mathbb{S}, \mathit{sid}) \\
& \mid \mathit{return}(\mathit{expr}, \mathit{sid}) \mid \mathit{if}(\mathit{expr}, \mathit{stmt}, \mathit{stmt}, \mathit{sid}) \\
& \mid \mathit{while}(\mathit{expr}, \mathit{stmt}, \mathit{sid}) \mid \mathit{asm}(\mathit{asm\ list}, \mathit{sid}) \\
& \mid \mathit{scall}(\mathbb{S}, \mathbb{S}, \mathit{expr\ list}, \mathit{sid}) \mid \mathit{xcall}(\mathbb{S}, \mathit{expr\ list}, \mathit{expr\ list}, \mathit{sid})
\end{aligned}
$$

Fig. 4. Data Type *stmt* for *C*0 Statements

literal l_c to an expression, are needed to initialize variables of aggregate types in a *single* step. This is required for the equivalence proof to the Hoare logic [40]. The left side of complex assignments is a normal expression of some aggregate type and the right side is a literal of the same type. Observe, that complex literals are only supported in this special case and must not be used inside normal expressions.

Dynamic allocation of zero-initialized heap memory for a type t is supported via $\mathit{new}(e, t)$ which assigns a pointer to the newly allocated memory region to the left side expression e. Observe, that *C*0 does not support explicit deallocation. Instead, a garbage collector will be used to deallocate unreachable parts of the heap in user applications. [6] The implementation correctness of a copying garbage collector for *C*0 has already been formally verified but is not yet integrated into the compiler correctness proof.

Function calls to a function f with parameters e_1 to e_n are represented by $\mathit{scall}(x, f, e_1, \ldots, e_n)$. Because *C*0 expressions must not have side effects, function calls are not supported as subexpressions. Instead, the return value of the function will be copied implicitly to variable x. Return from functions is handled by $\mathit{return}(e)$.

In the remainder of this paper we will often use the shorthand notation $r; s; t$ instead of $\mathit{comp}(r, \mathit{comp}(s, t))$ for consecutive statements r, s, and t.

2.1 C0 Small-step Semantics

*C*0 programs are represented in Isabelle by a symbol table *gst* for the global variables, a type name environment *te*, and a function table *ft*. The symbol table is a list of variable names together with their types. The type name environment maps type names to types. The function table maps function names to functions which are represented by a tuple consisting of a symbol table for the function's parameters, a symbol table for the local variables, the function's return type, and a statement representing the body of the function.

Configurations

Configurations c of the *C*0 small-step semantics consist of two components: the program rest $c.pr :: \mathit{stmt}$ and the memory configuration $c.mem$. The program rest stores those statements which still have to be executed. It is initialized with the

[6] The operating system kernel of the Verisoft project [19,15] does only allocate a fixed amount of memory at startup. Thus, garbage collection is not necessary and the collector is deactivated for the kernel.

body of the 'main' function and grows / shrinks during program execution. A program has terminated when $c.pr = skip$.

The memory configuration is a triple consisting of a global memory frame $c.mem.gm :: frame$, a stack of local memory frames $c.mem.lm :: (gvar \times frame)\ list$, and a memory frame for heap variables $c.mem.hm :: frame$. Each memory frame m consists of a symbol table $m.st$ which lists the variables of the frame and of a content function $m.ct :: \mathbb{N} \to mcell$ which maps addresses (natural numbers) to memory cells. A single memory cell can store values of elementary types. Values of aggregate types are stored flattened as a consecutive sequence of memory cells. Each local memory frame stores additionally a so-called *g-variable* which encodes the memory location where the function's result has to be stored.

Generalized Variables

Generalized variables (short g-variables) are a structural way of referring to memory objects. Pointers in the $C0$ small-step semantics are represented using g-variables. There are three base cases for g-variables: global variables of name x are represented by $gvar_{\mathrm{gm}}(x)$, local variables x in the i-th local memory frame by $gvar_{\mathrm{lm}}(i, x)$, and nameless heap variables with index i by $gvar_{\mathrm{hm}}(i)$. The inductive case defines g-variables for structure and array access. If g is a g-variable of structure type then a component $g' = gvar_{\mathrm{str}}(g, n)$ of name n is also a g-variable. If g is a g-variable of array type then its i-th element $g' = gvar_{\mathrm{arr}}(g, i)$ is also a g-variable. In these two cases, g' is called a sub g-variable of g.

We inductively define the set of *reachable* g-variables: a g-variable g is reachable iff (i) g is a global or local g-variable, (ii) another reachable pointer g-variable points to g, or (iii) g is a sub g-variable of a reachable g-variable.

Expression Evaluation and Transition Function

The value of expressions e–remember that $C0$ expressions are side effect free–and g-variables g in configuration c is computed via $va(c, e)$ and $va(c, g)$, respectively. It is represented as a sequence of memory cells. The transition function δ_{C0} computes for a given $C0$ configuration c the next configuration c'. If a runtime error (e.g., division by zero) occurs, the functions returns the special error state $\perp$ which it will never leave. We define $C0$ computations by repeated application of the transition function: we start in an initial configuration c^0 and define inductively $c^{i+1} = \delta_{\mathrm{C0}}(c^i)$.

For later reference, we highlight some parts of the definition of the new program rest $c'.pr$. Let the old program rest start with statement s, i.e., $c.pr = s; r$. In most cases s is simply executed and the new program rest is set to $c'.pr = r$. In three cases the length of the program rest can grow. (i) If $s = while(e, s')$ and $va(c, e) = true$ then the new program rest is $c'.pr = s'; s; r$. (ii) If $s = if(e, s_1, s_2)$ then the new program rest is $c'.pr = s_1; r$ or $c'.pr = s_2; r$. (iii) If s is a function call to some function f with body b then the new program rest is $c'.pr = b; r$.

An Invariant on Program Rests

We prove an invariant about program rests of the $C0$ small-step semantics which will be used in the correctness proof for the compiling specification in Section 3:

each statement s in the program rest of a computation for some program p, except for return statements, is always followed by some statically determined *successor* statement $succ(p, s)$.

To formalize this invariant we need additional definitions. Observe, that we model partial functions in Isabelle with an option type; here, we hide this formalism and represent undefined values by the special symbol $\perp$.

We denote by $s2l :: stmt \to stmt\ list$ a function which flattens a statement tree spanned by skip and compound statements into a list of statements as follows:

$$s2l(s) = \begin{cases} [\,] & \text{if } s = skip \\ s2l(s_1) \circ s2l(s_2) & \text{if } s = comp(s_1, s_2)\ . \\ [s] & \text{otherwise} \end{cases}$$

Let p be a $C0$ program and fb the function body which contains statement s in the function table of p. We define the *parent statement* of s in program p in the following way.

$$pa(p, s) = \begin{cases} \perp & \text{if } s \in s2l(fb), \text{ i.e., if } s \text{ is a top-level statement} \\ s' & \text{if } \exists s' \in p.\ s' = while(e, lb) \wedge s \in s2l(lb) \\ s' & \text{if } \exists s' \in p.\ s' = if(e, s_1, s_2) \wedge (s \in s2l(s_1) \vee s \in s2l(s_2)) \end{cases}$$

By induction we define the i-th parent statement by $pa^0(p, s) = s$ and $pa^{i+1}(p, s) = pa^i(p, pa(p, s))$. We define the *environment* of statements s, i.e., the list of statements in the basic block which s belongs to.

$$env(p, s) = \begin{cases} s2l(s_1) & \text{if } pa(p, s) = if(e, s_1, s_2) \wedge s \in s2l(s_1) \\ s2l(s_2) & \text{if } pa(p, s) = if(e, s_1, s_2) \wedge s \in s2l(s_2) \\ s2l(lb) & \text{if } pa(p, s) = while(e, lb) \\ s2l(fb) & \text{otherwise, i.e., if } s \text{ is a top-level statement} \end{cases}$$

For a statement s we define its *direct successor* $succ_{\mathrm{d}}(p, s)$ to be the next statement in the environment of s. The direct successor is undefined if s is the last statement in $env(p, s)$. Finally, if s is not the last statement of a function body (in this case it would be a return statement), we recursively define its *successor* $succ(p, s)$.

$$succ(p, s) = \begin{cases} succ_{\mathrm{d}}(p, s) & \text{if } succ_{\mathrm{d}}(p, s) \neq \perp \\ pa(p, s) & \text{if } succ_{\mathrm{d}}(p, s) = \perp \text{ and } pa(p, s) \text{ is a while loop} \\ succ(p, pa(p, s)) & \text{otherwise} \end{cases}$$

In the following, we will always argue in the context of a fixed $C0$ program; thus, we will mostly omit the first parameter p of the above definitions.

Theorem 2.1 (Invariant on Program Rests) *If $s \in s2l(c^i.pr)$ for some step number i of a $C0$ computation and s is not a return statement then the next statement in $s2l(c^i.pr)$ is the successor statement of s, i.e., s is always followed by its successor statement.*

Proof. This theorem depends follows from the fact that the program rest of $C0$ programs only changes in a certain way. We prove it by induction on the step number i.

For all statement trees which are literally copied from the function table, the invariant holds by definition of $succ(s)$. This proves the induction base, where the program rest consists only of the body of the main function. For the induction step let the program rest be $c^i.pr = s; r$. We do a case distinction on the statement s, which will be executed in the next step.

If s is an assignment, a while with *false* condition, a return, or a new statement, s is simply consumed ($c^{i+1}.pr = r$) and the invariant obviously holds.

If $s = while(e, lb)$ and $va(c^i, e) = true$ we have $c^{i+1}.pr = lb; while(e, lb); r$. Because lb is part of the function table, the invariant holds for this part of the new program rest. The other part of the program rest remains unchanged. The crucial point is to prove that the while statement is the correct successor for the last statement s' of the loop body, formally: $succ(s') = while(e, lb)$. This follows from the second case of the definition of $succ(s)$.

If $s = if(e, s_1, s_2)$ the new program rest is $c^{i+1}.pr = s_1; r$ or $c^{i+1}.pr = s_2; r$, depending on the value of e. Let for both cases s' denote the last statement in the corresponding branch s_1 or s_2. We have to show that s' is followed by $succ(s')$ in the new program rest. By the third case of the definition of $succ$, we know that $succ(s') = succ(s)$. Therefore, we can conclude with help of the induction hypothesis that s' is followed by the same statement which followed the conditional s in the original program rest (in all cases this is the first statement in statement list r).

If s is a function call of some function f with body fb the new program rest is $c^{i+1}.pr = fb; r$. For r, the invariant still holds by induction hypothesis; for fb, it holds by definition of $succ$ because of fb being a sub tree of the function table. The interesting case is again the crossing from fb to r. However, in this special case there is nothing to show because the last statement in the function body is a return and the invariant does not state anything for return statements. □

3 Correctness of the Compiling Specification

The code generation algorithm of the $C0$ compiler is quite simple. It starts by iterating over all functions in the function table and generates code for their bodies. The code generation for statements and expressions—in the context of a certain function—is done by a simple recursive algorithm which follows the structure of the corresponding data types. We denote code generation of the compiling specification for statements s and expressions e in this paper by $code_{\mathrm{s}}(s)$ and $code_{\mathrm{s}}(e)$; analogously, we denote the code generated by the implementation by $code_{\mathrm{i}}(s)$. With $cad(s)$ we denote the start address of the code which has been generated for statement s and with $ead(s)$ the address of the first instruction behind this code. As an example we present the code generation template for loops in Figure 5.

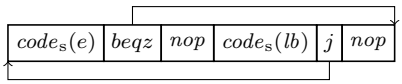

Fig. 5. Code Generation Template for Loops: $code_s(while(e, lb))$

3.1 Simulation Relation

We define a simulation relation between configurations c of the $C0$ machine and configurations d of the VAMP assembler machine. The latter are composed of two program counters $d.pc$ and $d.dpc$ implementing the delayed branch mechanism (see [7,32]), a word addressed memory $d.m$, and a general purpose register file $d.gpr$.

The set of valid g-variables of a $C0$ machine changes with *new* statements, function calls, and returns, and garbage collectors may change the allocated base address of heap g-variables. Thus, the simulation relation is parametrized with the current allocation function *alloc* which maps g-variables to their allocated base address in the VAMP assembler machine.

The simulation relation $consis(c, alloc, d)$ states that the VAMP configuration d encodes the $C0$ configuration c via the allocation function *alloc*. It comprises *control consistency* $consis_c(c, d)$ and *data consistency* $consis_d(c, alloc, d)$. Control consistency states that the VAMP's program counters point to the code of the first statement in the current program rest: $d.dpc = cad(hd(c.pr))$ and $d.pc = d.dpc + 4$. Data consistency is a conjunction of the following predicates.

Code consistency $consis_{code}(c, d)$ requires that the compiled code of the $C0$ program is stored at address 0 of the VAMP machine; this forbids self-modification.

Value consistency $consis_v(c, alloc, d)$ requires for all *reachable* g-variables g of basic type that $C0$ and VAMP machine store the same value: $d.m(alloc(g)) = va(c, g)$. For reachable pointer g-variables p which point to some g-variable g we require that the value stored at the allocated address of p in the VAMP machine is the allocated base address of g, i.e., $d.m(alloc(p)) = alloc(g)$. This defines a subgraph isomorphism between the reachable portions of the heaps of the $C0$ machine and the VAMP machine.

Stack consistency $consis_s(c, d)$ is a predicate on the implementation of the run time stack and the content of some special registers. Informally, it states that the first three words of each frame in the VAMP machine store the return address, i.e., where the code for final return statement jumps to, the destination address for the function's result, and a pointer to the previous frame. Additionally, we require that the return addresses in the VAMP agree with the control flow in the program rest of the $C0$ machine. Formally: that for all $i + j = |c.mem.lm|$ the return address stored in the j-th stack frame matches the address of the statement which follows the i-th return statement in the program rest.

3.2 Simulation Theorem

Essentially, the main theorem about the compiling specification states that for all steps i of the $C0$ machine, there exists a corresponding step number $s(i)$ such that after $s(i)$ steps the assembler machine is consistent with the $C0$ machine after i steps (cf. Figure 6). In reality, the theorem requires several additional preconditions.

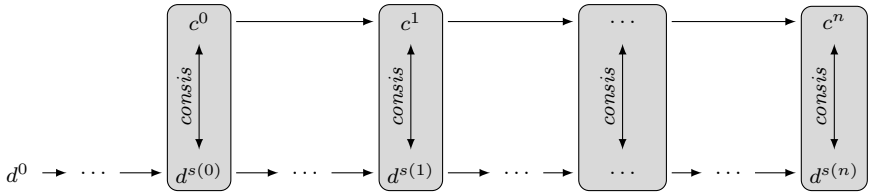

Fig. 6. Small-step Compiler Simulation Theorem

Theorem 3.1 (Simulation Theorem) *Let p be a C0 program, c^0 the corresponding initial configuration of the C0 machine, and d some well-formed initial assembler configuration which contains the compiled code $code_s(p)$ at address 0. Then, it holds for all steps i of the C0 machine executing program p that there exists an assembler step number $s(i)$ and an allocation function $alloc^i$ such that the C0 machine after i steps is consistent with the assembler machine after $s(i)$ steps.*

However, this is only true of the following requirements are fulfilled.[7]

- *The program p has to be translatable for our compiler: $p \in xltbl_{prog}$. Basically, this requires that the compiled code is not too big for the target machine, that jump distances fit into the immediate operands of the corresponding VAMP instructions, and that expression evaluation does not require too much registers to store intermediate results.*
- *We must not reach an error state up to step i of the C0 computation: $c^i \neq \bot$.*
- *There must not be a stack overflow up to step i of the C0 computation: $\forall j \leq i : \neg ovfl_{stack}(c^j)$.*

Formally, the theorem is stated as

$$\begin{aligned} &\forall i : p \in xltbl_{prog} \wedge c^i \neq \bot \wedge \forall j \leq i : \neg ovfl_{stack}(c^j) \\ &\quad \Longrightarrow \exists s(i), alloc^i : consis(c^i, alloc^i, d^{s(i)}). \end{aligned}$$

Proof. We prove this theorem by induction on i. For the induction start $i = 0$ we mainly have to show that the initialization part of $code_s(p)$ works correctly. The induction step from i to $i+1$ is proved by a case distinction over the first statement s in the program rest $c^i.pr = s; r$. We cannot present all cases here but concentrate on one interesting detail of the proof which comes from the fact that we prove a *small-step* simulation theorem.

Assume, that the program rest of the next configuration c^{i+1} starts with some statement s'. For control consistency, we have to show that the program counters eventually point to $cad(s')$. For the three cases in which the program rest grows (cf. Section 2.1), this proof is relatively easy because the correctness arguments are *local* regarding the statement s to be executed in step i. For return statements the proof follows immediately from stack consistency, which guarantees that the return addresses on the stack are correct.

For the remaining cases, s is completely consumed in the next step and we have $s' = r$. Thus, we have to show that we eventually reach the start of $code_s(r)$ where r—the new head of the program rest—is by Theorem 2.1 the successor statement

[7] Due to space restrictions we do not formally define the requirements in this paper. For details see [23].

of s. However, it is not guaranteed that $code_s(r)$ directly follows $code_s(s)$ in the compiled program; instead some *control code* might be placed between $code_s(s)$ and $code_s(r)$ (cf. Figure 7).

For example, consider s being the last statement in the if-branch s_1 of the conditional statement. There, $code_s(s)$ is followed by a jump instruction which skips the else-branch s_2 and this jump does not belong to $code_s(s_1)$ although it is indispensable to ensure control consistency after execution of s_1. In this case, the proof of control consistency is not *local* w.r.t. s but depends on the code for the conditional statement; even more, the jump instruction behind the code for the loop body also needs to be considered. However, in configuration c^i the conditional statement is no longer present in the program rest and we cannot easily argue about the correctness of the jump instruction. Instead we outsource the correctness proof of the control code into the following lemma. □

The first requirement of the previous theorem, i.e., that the program is translatable, is formulated in the executable subset of Isabelle's specification language and can be easily checked once and for all for a given $C0$ program using Isabelle's ML code generator. The other two requirements argue about runtime properties of the program; in the Verisoft scenario, they follow from the functional correctness proof of the program to be compiled.

Lemma 3.2 (Control Code Correctness) *Let the program counter of assembler configuration d^i point directly behind the last instruction of $code_s(s)$, where s is not a return statement. Then, it holds that after a certain number t of assembler steps we reach a configuration where the program counter points to the first instruction of the successor statement of s and the memory has not been changed.*

$$d^i.dpc = ead(s) \wedge d^i.pc = d^i.dpc + 4$$
$$\Longrightarrow \exists t : d^{i+t}.dpc = cad(succ(s)) \wedge d^{i+t}.pc = d^{i+t}.dpc + 4 \wedge d^{i+t}.m = d^i.m$$

Proof. We prove this theorem by induction—following structurally the definition of $succ(s)$. If s is not the last statement in a loop body or in the branch of some conditional, then there is no control code behind $code_s(s)$ and we are done.

If s is the last statement in the body of some loop $while(e, lb)$, we know from the definition of $succ$ that $while(e, lb)$ itself is the successor of s. By a proof, which is *local* w.r.t. the while loop, we can show that the distance of the jump instruction behind the loop body is correct and we finally have $d^{i+t}.dpc = cad(while(e, lb)) = cad(succ(s))$.

The most complicated case is when s is the last statement in a branch of some conditional statement. First, we show—using a simple auxiliary lemma—that the control code behind the conditional branch correctly jumps behind the code of the conditional statement, i.e., that we reach $cad(pa(s))$. Then, we apply the induction hypothesis to show that we finally reach $cad(succ(pa(s))) = cad(succ(s))$. □

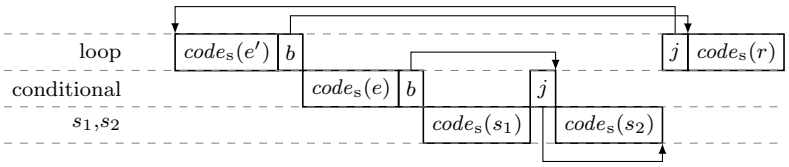

Fig. 7. Fragmented code generation for $code_s(while(e', if(e, s_1, s_2)); r)$

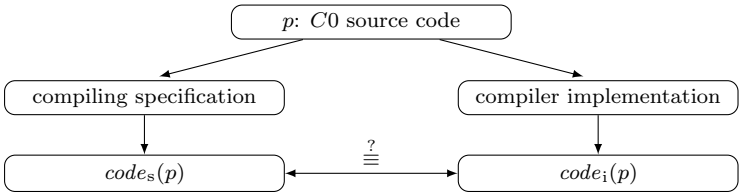

Fig. 8. Correctness of the Compiler Implementation

4 Implementation Correctness

For pervasive verification, it is not sufficient to have a verified compiling specification. Additionally, we need a verified compiler implementation in $C0$ which allows us (after boot strapping, cf. Section 4.4) to *execute* a verified compiler binary on the target platform. In addition to Theorem 3.1, it suffices to show that the compiler implementation produces the same code as the compiling specification (cf. Figure 8).

We have restricted the verification of the compiler implementation in Verisoft to the implementation of the code generation algorithm which consists of roughly 1.500 lines of $C0$ code in about 60 procedures. Due to limited project resources, parsing and I/O operations have not been verified. The verified compiler core is embedded into an unverified front-end written in C/C++, which parses a $C0$ input program, checks its syntactical correctness, and produces a syntax tree, which is then being fed into the compiler core. The verified core translates the $C0$ syntax tree into a list of VAMP assembler instructions, which is output by an unverified I/O routine.

The compiling specification works in one pass: offsets for relative jumps are determined on the fly via functions which compute solely the size of the generated code. In contrast, the compiler implementation in Section 4 uses two-pass compilation. Jump distances for relative jumps are left out in the first pass and filled in with correct values in a second pass when the position of all jump destinations is known.

4.1 Verification Environment

The compiler implementation has been verified in the $C0$ verification environment [40] which is based on a Hoare logic with an automatic verification condition generator (VCG) and allows to prove both partial and total correctness. The VCG automatically applies Hoare rules to a Hoare triple $\{P\}c\{Q\}$ by computing the weakest precondition WP for c, Q, and user-provided invariants for loops. After the program c is completely eliminated, the goal $P \longrightarrow WP$ is to be shown interactively in Isabelle.

The heap model features split heaps for every type (following ideas from [10]),

which gives separation of heap structures of different types for free. Additionally, the verification environment embeds $C0$ expressions *shallowly* into HOL to increase productivity. Due to the shallow embedding, the range of elementary types is—in contrast to the $C0$ small-step semantics—not bounded. Thus, expressions have to be annotated with so-called *guards* to allow the transfer of properties from the Hoare logic layer to the lower layers. Validity of such guards, which are generated automatically, implies the absence of run-time errors caused by over- or underflow.

4.2 Correctness Theorem

We formulate pre- and postconditions of the Hoare triples using so-called *abstraction relations*, which state the correspondence between the current state variables and abstract HOL types [28]. Abstraction relations have to be defined for all relevant data structures of the compiler implementation. Absence of pointers in the specification language results in very different representation of objects and, hence, makes abstraction relations and verification more complex.

As we do not prove the correctness of the front-end, we assume that the initial state σ of the compiler core contains a syntax tree of the input program and, analogously, the final state τ encodes the compiled instruction sequence. Formally, this is stated by the two top-level abstraction relations $C0_{\text{prog}}(\sigma, p)$ which states that σ encodes the $C0$ program p and $ASM_{\text{code}}(\tau, l)$ which states that τ encodes a list l of VAMP instructions. Using these abstraction relations we can formulate the top-level correctness theorem for the compiler implementation.

Theorem 4.1 *Let p be a $C0$ program, cimpl the $C0$ function which implements the code generation, and σ the initial state. Then, after executing cimpl, the final state τ encodes exactly that list of VAMP instructions which is specified by the compiling specification via $code_s(p)$. Formally, this is stated by the following Hoare triple.*

$$\{C0_{prog}(\sigma, p)\}\ \mathit{Call}\ \mathit{cimpl}();\ \{ASM_{code}(\tau, code_s(p))\}$$

4.3 Verification Issues

We highlight some of the key verification issues for the compiler implementation (besides code size). One of these follows from the implementation and specification being written in an imperative and a functional programming language, respectively. Thus, the correct implementation of recursive functions by while loops is an issue. Additionally, the recursion directions often differ; for example, in the implementation lists are traversed from head to tail and the specification exploits natural recursion with the last list element as induction base. Another example is the code generation for complex literals, where mutually recursive functions in the specification are implemented by a combination of recursive functions and loops.

In some cases, a *single* function in the specification is implemented by a combination of *several* $C0$ functions. One interesting example is the equivalence of the two-pass translation in the compiler implementation with the single-pass recursive function in the specification. Such cases require the introduction of additional intermediate states and predicates which allow to connect the different implementation

functions until we can finally prove their equivalence with the single specification function.

4.4 Boot Strapping

To solve the bootstrap problem [16], i.e., to obtain a trustworthy binary of the *C*0 compiler, it is not sufficient to verify the code generation algorithm or the *C*0 implementation of the compiler. In Verisoft, we follow two different ways how to get a trustworthy executable from the compiler implementation. First, B. Finkbeiner's group is currently applying translation validation techniques [39] to show that a binary compiler which has been generated by an untrusted bootstrap compiler is a correct translation of the compiler implementation from Section 4. Second, we have used Isabelle's built-in ML code generation feature [3,4] to compile the (verified parts of the) implementation from Section 4 using the functional compiling specification from Section 3 (see also [42]).

Of course, both approaches extend the trusted code base: the first by the translation validation tool, the second by Isabelle's code generation module. However, we can simply apply both methods and compare the resulting binaries. The probability that both produce the same error is negligible.

5 Conclusion and Future Work

We have sketched in this paper the correctness proof of a simple, non-optimizing *C*0 compiler. The correctness proof has been formalized in the theorem prover Isabelle / HOL and is split into a simulation theorem for the compiling specification and a proof for the total correctness (including termination and validity of guards) of the compiler implementation consisting of 1.500 lines of *C*0 code.[8] The formal proofs and definitions consist of roughly 85.000 lines of Isabelle code. This number covers the *C*0 small-step semantics (15.000 lines, including type correctness proofs and Theorem 2.1), the correctness proof for the compiler implementation (40.000 lines), and the simulation proof for the compiling specification (30.000 lines).

The compiler has been verified in the context of pervasive system verification in Verisoft. We had to deal with resource limitations on the target machine (e.g., restricted memory size) and other additional requirements; especially arguments about small-steps semantics have become mandatory. Thus, the top-level correctness theorem had to be extended with additional requirements on the *C*0 computations (e.g., limits on recursion depth), which guarantee that properties proved at the source language layer also hold for the compiled code.

We have also presented a solution to the boot strap problem. The compiling specification is in the executable subset of Isabelle's specification language. Thus, a trustworthy compiler binary can be generated by executing the specification.

The *C*0 implementation of a copying garbage collector has already been verified in Verisoft. However, the integration of this result into the compiler simulation theorem remains as future work.

[8] The formal proofs for the work presented in this paper can be downloaded from the Verisoft repository at http://www.verisoft.de/VerisoftRepository.html.

References

[1] Alkassar, E., M. Hillebrand, S. Knapp, R. Rusev and S. Tverdyshev, *Formal device and programming model for a serial interface*, in: B. Beckert, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, 2007, pp. 4–20.

[2] Alkassar, E., A. Starostin and N. Schirmer, *Formal pervasive verification of a paging mechanism*, in: *Fourteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), to appear*, 2008.

[3] Berghofer, S., "Proofs, Programs and Executable Specifications in Higher Order Logic," Ph.D. thesis, Technical University of Munich (2003).

[4] Berghofer, S. and T. Nipkow, *Executing higher order logic*, in: *TYPES '00: Selected papers from the International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science (LNCS) **2277** (2002), pp. 24–40.

[5] Beuster, G., N. Henrich and M. Wagner, *Real world verification – experiences from the verisoft email client*, in: G. Sutcliffe, R. Schmidt and S. Schulz, editors, *Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning*, 2006.

[6] Bevier, W. R., W. A. Hunt, Jr., J S. Moore and W. D. Young, *An approach to systems verification*, Journal of Automated Reasoning (JAR) **5** (1989), pp. 411–428.

[7] Beyer, S., C. Jacobi, D. Kroening, D. Leinenbach and W. Paul, *Putting it all together: Formal verification of the VAMP*, International Journal on Software Tools for Technology Transfer **8** (2006), pp. 411–430.

[8] Blazy, S., Z. Dargaye and X. Leroy, *Formal verification of a C compiler front-end*, in: J. Misra, T. Nipkow and E. Sekerinski, editors, *FM 2006: 14th International Symposium on Formal Methods*, LNCS **4085** (2006), pp. 460–475.

[9] Blech, J. O. and S. Glesner, *A formal correctness proof for code generation from SSA form in Isabelle / HOL.*, in: P. Dadam and M. Reichert, editors, *GI Jahrestagung (2)*, Lecture Notes in Informatics **51** (2004), pp. 449–458.

[10] Burstall, R., *Some techniques for proving correctness of programs which alter data structures*, in: B. Meltzer and D. Michie, editors, *Machine Intelligence 7* (1972), pp. 23–50.

[11] Chirica, L. M. and D. F. Martin, *Toward compiler implementation correctness proofs*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 185–214.

[12] Dalinger, I., "Formal Verification of a Processor with Memory Management Units," Ph.D. thesis, Saarland University, Computer Science Department (2006).

[13] Dave, M. A., *Compiler verification: A bibliography*, Association for Computing Machinery (ACM) SIGSOFT Software Engineering Notes **28** (2003).

[14] Dold, A. and V. Vialard, *A mechanically verified compiling specification for a Lisp compiler*, in: R. Hariharan, M. Mukund and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Sience*, LNCS **2245** (2001), pp. 144–155.

[15] Gargano, M., M. Hillebrand, D. Leinenbach and W. Paul, *On the correctness of operating system kernels*, in: J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS **3603** (2005), pp. 1–16.

[16] Goerigk, W. and U. Hoffmann, *Rigorous compiler implementation correctness: How to prove the real thing correct*, in: D. Hutter, W. Stephan, P. Traverso and M. Ullmann, editors, *Applied Formal Methods – FM-Trends 98*, LNCS **1641**, 1998, pp. 122–136.

[17] Gurevich, Y. and J. K. Huggins, *The semantics of the C programming language*, in: E. Börger, G. Jäger, H. K. Büning, S. Martini and M. M. Richter, editors, *CSL '92: Selected Papers from the Workshop on Computer Science Logic*, LNCS **702** (1993), pp. 274–308.

[18] Hillebrand, M., T. In der Rieden and W. Paul, *Dealing with I/O devices in the context of pervasive system verification*, in: *ICCD '05* (2005), pp. 309–316.

[19] In der Rieden, T. and A. Tsyban, *CVM - a verified framework for microkernel programmers*, in: *3rd intl Workshop on Systems Software Verification (SSV08)* (2008).

[20] "ISO 9899:1999: Programming languages – C," International Standardization Organization, 1999.

[21] Joyce, J. J., *Totally verified systems: Linking verified software to verified hardware.*, in: M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis*, LNCS **408** (1989), pp. 177–201.

[22] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine, and compiler*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 619–695.

[23] Leinenbach, D., "Compiler Verification in the Context of Pervasive System Verification," Ph.D. thesis, Saarland University, Computer Science Department, under appraisal (2008).

[24] Leinenbach, D., W. Paul and E. Petrova, *Towards the formal verification of a C0 compiler: Code generation and implementation correctness*, in: B. Aichernig and B. Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005, pp. 2–11.

[25] Leroy, X., *Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant*, in: J. G. Morrisett and S. L. P. Jones, editors, *33rd symposium Principles of Programming Languages* (2006), pp. 42–54.

[26] Loeckx, J., K. Mehlhorn and R. Wilhelm, "Foundations of Programming Languages," John Wiley & Sons, Inc., 1989.

[27] McCarthy, J. and J. Painter, *Correctness of a compiler for arithmetic expressions*, in: J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, Proceedings of Symposia in Applied Mathematics **19** (1967), pp. 33–41.

[28] Mehta, F. and T. Nipkow, *Proving pointer programs in higher-order logic*, in: F. Baader, editor, *Conference on Automated Deduction (CADE)'03*, LNCS **2741** (2003), pp. 121–135.

[29] Moore, J S., *Piton: A verified assembly level language*, Technical Report 22, Comp. Logic Inc. Austin, Texas (1988).

[30] Moore, J S., *A grand challenge proposal for formal methods: A verified stack*, in: B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, LNCS **2757** (2003), pp. 161–172.

[31] The Motor Industry Research Association (MIRA), Ltd., UK, "MISRA-C:2004 — Guidelines for the use of the C language in critical systems," (2004).
URL http://www.misra-c2.com/

[32] Müller, S. M. and W. J. Paul, "Computer Architecture: Complexity and Correctness," Springer, 2000.

[33] Ni, Z., D. Yu and Z. Shao, *Using XCAP to certify realistic systems code: Machine context management*, in: K. Schneider and J. Brandt, editors, *TPHOLs*, LNCS **4732** (2007), pp. 189–206.

[34] Nielson, H. R. and F. Nielson, "Semantics with Applications: A Formal Introduction," John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

[35] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer, 2002.

[36] Norrish, M., "C Formalised in HOL," Ph.D. thesis, University of Cambridge, Computer Laboratory (1998).

[37] Papaspyrou, N., "A Formal Semantics for the C Programming Language," Ph.D. thesis, National Technical University of Athens (1998).

[38] Petrova, E., "Verification of the C0 Compiler Implementation on the Source Code Level," Ph.D. thesis, Saarland University, Computer Science Department (2007).

[39] Pnueli, A., M. Siegel and E. Singerman, *Translation validation*, in: B. Steffen, editor, *TACAS '98*, LNCS **1384** (1998), pp. 151–166.

[40] Schirmer, N., "Verification of Sequential Imperative Programs in Isabelle/HOL," Ph.D. thesis, Technical University of Munich (2006).

[41] Strecker, M., *Formal verification of a Java compiler in Isabelle*, in: A. Voronkov, editor, *CADE'02*, LNCS **2392**, pp. 63–77.

[42] Tzigarov, H., "Extended Oracle Proof Methods for Isabelle / HOL: Reflection and SMV Support," Diploma's thesis, Saarland University, Computer Science Department (2007).

[43] Zimmermann, W. and A. Dold, *A framework for modelling the semantics of expression evaluation with abstract state machines*, in: E. R. Egon Boerger, Angelo Gargantini, editor, *Abstract State Machines - Advances in Theory and Applications 10th International Workshop, ASM 2003*, LNCS **2589** (2003), pp. 391–406.

[44] Zimmermann, W. and T. Gaul, *On the construction of correct compiler back-ends: An ASM-approach*, Journal of Universal Computer Science **3** (1997), pp. 504–567.