Validating the Translation of an Industrial Optimizing Compiler*

I. Gordin† R. Leviathan† A. Pnueli†‡

August 18, 2004

{ira.gordin | raya.leviathan | amir.pnueli}@weizmann.ac.il

Abstract

The paper presents an approach to the translation validation of an optimizing compiler which translates synchronous C programs into machine code programs. Being synchronous means that both source and target programs are loop free. This enables representation of each of these programs by a single state transformer, and verification of the translation correctness is based on comparison of the source and target state transformers. The approach has been implemented on a tool called MCVT which is also described.

1 Introduction

In recent years there is a growing awareness of the importance of formally proving the correctness of safety critical portions of software systems. The industry is taking first steps towards use of formal methods for the validation of real systems. At the same time the importance of compiler optimizations is increasing, together with the development of modern CPU architectures. Consequently, the validity of the optimized code is not evident. This is the reason that standards and regulations are required to qualify a compiler, to be used in a safety critical system. In other cases, a manual check of the translated code is required. Dealing with commercial compiler poses new obstacles for the code validation effort. Intermediate representation is not available to the verifier and a major part of the knowledge about optimizations is confidential. The available compiler output is a standard object file format or assembly code, and standard debug information. However, when optimizations are activated, the debug information is only partially reliable.

The research described in this paper is part of the SafeAir-II project of IST (*Information Society Technologies program of the European Union*), which aims towards the development of Advanced Design Tools for Safety Critical Software. The tool described here is a tool for *translation validation* of an optimizing compiler. This tool accepts a source program and the corresponding target code produced by the compiler and validates that the target code is a correct translation of the source program. The compiler that was chosen is WindRiver DiabData [Dia] compiler for the PowerPC [PPC02] family of processors, a compiler which is widely used in the development of safety-critical embedded systems.

The major challenge of this research is to handle an **industrial environment**, where the source language, the processor, the compiler as well as the applications, are all industrial grade. In order

*This research was supported in part by the Minerva Center for Verification of Reactive Systems, IST project SafeAir-II, and NSF grant CCR-0205571

†Dept. of Computer Science, Weizmann Institute of Science, Rehovot, Israel

‡New York University

to be useful to the industrial sector, **a fully automatic process** is obligatory. The programs we handled were C programs, automatically produced from a **synchronous language** — SCADE [Ver99] used to design embedded systems. Synchronous languages, such as SCADE provide means to describe a reactive systems with finite or infinite computations. The C programs that are produced are composed of compilation units that have a special structure. Except for the main program which contains a single loop that repeats forever, all other units (procedures) and the body of the main program's loop are practically loop-free. That is, the only nested loops they may contain have a fixed constant number of iterations (known at compile time). On the other hand, they may contain an arbitrarily complex conditional control structure. As a result, the produced machine code contains a significant number of conditional branches, and thus the number of possible paths through these programs is significantly high, though their length is bounded. These properties enable us to describe the effect of executing the program at each cycle (single iteration of the main program's loop) by *STS - State Transformer System*.

We propose a proof procedure which is capable of proving the translation correctness without previous knowledge about the optimizations performed by the compiler, and the compiler internal intermediate representation. When dealing with programs with more general control structure, e.g. non-trivial nested loops, the general approach to translation validation (e.g. [ZPFG02]) requires the construction of a control abstraction mapping as well as local (non-observable) variables mappings. In our more restricted context of synchronous control programs, these constructions are unnecessary. Instead, we present a special algorithm (called ANNOTATION) which, in a forward traversal of a loop-free program, constructs a state transformer which expresses all the possible data transformations this program may effect in a single execution.

The MCVT tool **machine code validation tool** described here, uses the proof procedure, together with the ANNOTATION algorithm and additional heuristics, to automatically produce verification conditions. The verification conditions are formulas of decidable logic theories. MCVT uses CVC, a Cooperating Validity Checker [SBD02], developed at Stanford University, to establish the validity of the verification conditions, and thus the correctness of the translation.

Section 3 contains the theoretical framework, including the definitions of state transformer system, correctness of translation and a reduced procedure to prove it for synchronous programs. In Section 4 we introduce the source synchronous programs with their inherent restrictions, and present the syntax and semantics of synchronous C programs. Section 5 presents the syntax and semantics of synchronous machine code programs. Subsection 5.2 describes a linear time algorithm to construct the STS corresponding to a synchronous machine code program. We conclude with a presentation of the MCVT tool, the heuristics and algorithms incorporated into it, and results of using the tool.

2 Related Work

Several methods were proposed in order to mechanically solve the translation validation problem. Most methods start by constructing a data abstraction mapping. Then, either proof obligations or checkers, are produced. The proof obligations can be validated by an "off the shelf" validity checker, or by a theorem prover [ZG97, GZ99, GZG00, GZ00]. In other cases, the validation process is based on heuristics and, interaction between the heuristic part and the decision procedures, or the formulas' simplifier [Nec00, RM00]. Yet another approach lets the compiler produce the proofs that are needed to establish its correctness [RM00, GZ00].

The CVT tool [PSS98b, PSS99] is an example of an industrial quality translation validation tool. It demonstrates an automatic procedure for validating the translation from the synchronous language SCADE to a sequential C program. The main concerns addressed by this tool are the semantic difference between a synchronous language and a sequential C code, as well as the data abstraction mapping. The proof obligations that are produced, include equalities and uninterpreted functions, relying on the fact that the compiler does not change the order arithmetic functions are applied. The final validation is done by a decision procedure developed specially for the CVT tool. This paper can be considered as a continuation of the CVT project by validating the next link in the development sequence: from C programs produced by the SCADE tool to optimized machine code.

The VOC project [ZPFG02] concentrates on the translation of C programs by an optimizing compiler, where the investigated optimizations are machine independent. The language of both source and target systems is the intermediate representation of the compiler. Two different proof rules are presented: one which handles loop transformations and another for "structure preserving" transformations. The first proof rule deals with "structure preserving" transformations and is based on Floyd's inductive-assertion method [Flo67]. The other proof rule deals with non-"structure preserving" transformations, which are permutations on the loop iterations' order. The method we employ for our validation bears some resemblance to the structure preserving part. However, due to the special restrictions intrinsic to synchronous programs, we can employ a significantly simpler and more efficient approach which is described in this paper.

Other works are done as part of the Verifix project [GZ99, GZG00]. These rely on the PVS theorem prover [COR$^+$95], in order to mechanize the verification, but full automation is not supported. One part of the Verifix project concentrates on compilers back-end. It formalizes the semantics of the assembly code, as well as of an intermediate representation, using high logics which are supported by PVS, whereas our goal is to use formulas whose validity is decidable. Another aspect of the Verifix project, which relates to our work, is the *program checker* approach. This approach, when applied to compilers verification, is the same as translation validation, since a validation checker is produced for each translation. However, contrary to our work, the checker is produced by the compiler. In the work described in [RM00], it is the compiler that produces the proofs, but the validity of the proof is established by the tool itself. The correctness proof consists of two sub-proofs: a sub-proof which shows that the analysis of the input program produces a correct result and, a sub-proof that establishes an equivalence between the original and the transformed programs.

Another method is described in [Nec00]. It is based on heuristics and a special purpose simplifier. The latter is composed of inference rules and is built in a tool. The tool is able to validate translations automatically. It can handle a wide range of optimizations, but as reported, may fail for loop unrolling and does not handle software pipelining. This work also uses the intermediate representation and validates each optimization pass separately.

While we choose to validate the translation, some works formally validate the translator itself. For example, this approach is used in [SSB01], where *ASM (Abstract State Machines)* are used to define the semantics of the source (Java) and target (JVM) systems. Compiler formal specifications are provided as well as a proof of the correctness of this compiler with respect to the Java and JVM specifications. The compiler correctness proof is done manually.

All the above mentioned works use compiler assistance, to some degree. It is a major challenge in translation validation research to minimize this dependence.

3 State Transformers and Their Refinement

To define the notion of correct translation between synchronous programs, we need to define the semantics of the source and target programs. To express the semantics of synchronous programs, we use *State Transformer Systems* (STS), which are a degenerate case of the *synchronous transition systems* introduced in [PSS98a].

3.1 State Transformer System

A state transformer system $\mathcal{S} : \langle V, O, F \rangle$, consists of the following components:

- V — a finite set of typed *system variables*. A type-consistent interpretation of the variables V is a system *state*. We denote by Σ the set of system states. For a state $s \in \Sigma$ and variable $v \in V$, we denote by $s[v]$ the value assigned to v by s.
- $O \subseteq V$ — a set of *observable variables*. The observable variables are the variables which are visible to the external world. A *state-observation* is a projection of a state $s \in \Sigma_V$ on the observable variables and is denoted by $s \Downarrow O$.
- $F : \Sigma \mapsto \Sigma$ — a *State transformer*, expressing the state transformation effected by the system. If the system variables are $V = \{v_1, \ldots, v_k\}$, then F consists of the list of functions $F_1, \ldots, F_k$, where each F_i maps Σ into the domain of v_i. We denote by F_O, the restriction of the list of functions to the functions corresponding to the observable variables.

A *computation* of a STS is a pair $\sigma = (s_0, s_1)$ where $s_1 = F(s_0)$. A pair of O-states $o_0, o_1 \in \Sigma\Downarrow_O$ is called an *observation* of $\mathcal{S}$ if there exists $s_0, s_1 \in \Sigma$, such that $o_0 = s_0\Downarrow_O$, $o_1 = s_1\Downarrow_O$, and (s_0, s_1) is a computation of $\mathcal{S}$.

3.2 Transformer Composition

In many cases, we have to compose two transformation functions. For example, we may have a concatenation of two statements S_1; S_2 and the transformers F_{S_1}, F_{S_2}, corresponding to statements S_1, S_2, respectively. The transformer for the concatenation S_1; S_2 will be given by the *composition*

$$F_{S_1;S_2} \quad = \quad F_{S_1} \circ F_{S_2}$$

In general, assume the system variables $V = \{v_1, \ldots, v_m\}$, and the two transformers $F^1, F^2 : \Sigma \mapsto \Sigma$. Then the composition $F^1 \circ F^2$ is defined by

$$F^1 \circ F^2 \quad = \quad \lambda V.F^2(F^1(V))$$

In particular, if $F = F^1 \circ F^2$, then $F_j(V) = F_j^2(F^1(V))$, for $j = 1, \ldots, m$.

For example, assume that $V = \{a, b\}$, statement S_1 is the assignment $b := b + 1$ and statement S_2 is the multiple assignment $(a, b) := (2 * b, a + b + 1)$. Then the respective transformers are given by

$$\begin{array}{rcl} F_{S_1} & : & (a, b+1) \\ F_{S_2} & : & (2 * b, a + b + 1) \\ F_{S_1;S_2} = F_{S_1} \circ F_{S_2} & : & (2(b+1), a + b + 2) \end{array}$$

3.3 Refinement Between STS's

A correct translation needs only to ensure the preservation of the observable behavior [ZG97]. A target program is a correct translation of a source system program, if every observation that exists in the target system, also exists in the source system. This characteristic is formalized by the notion of *refinement* [EdR+99]. Let $\mathcal{S}^S = \langle V^S, O^S, F^S \rangle$ and $\mathcal{S}^T = \langle V^T, O^T, F^T \rangle$, be source and target STS's. Let $R \subseteq \Sigma^S \Downarrow_O \times \Sigma^T \Downarrow_O$ be a relation between source state observations and target state observations. We say that $\mathcal{S}^T$ is an *R-refinement* of $\mathcal{S}^S$ if, for every source and target states $s \in \Sigma^S$, $t \in \Sigma^T$,

$$R(s \Downarrow_O, t \Downarrow_O) \quad \text{implies} \quad R(F^S_O(s), F^T_O(t))$$

Equivalently, for every $s_1, s_2 \in \Sigma^S$ and $t_1, t_2 \in \Sigma^T$, such that $s_2 = F^S(s_1)$ and $t_2 = F^T(t_1)$, if the observable parts of s_1, t_1 are R-related, then so are the observable parts of s_2, t_2.

System $\mathcal{S}^T$ is a *refinement* of $\mathcal{S}^S$ if there exists a *refinement relation* R, such that $\mathcal{S}^T$ is an R*-refinement* of $\mathcal{S}^S$.

3.4 Verifying the Correctness of Translation - The COMPARE Procedure

Let $\mathcal{S}^S : \langle V^S, O^S, F^S \rangle$ and $\mathcal{S}^T : \langle V^T, O^T, F^T \rangle$ be a source and the target STS'a, respectively. We assume that the observable variables of the two systems are $O^S : \{V_1, \ldots, V_k\}$, $O^T : \{v_1, \ldots, v_k\}$, and that there exists a $1-1$ correspondence between these variables. Furthermore, we will restrict our attention to the case that the refinement mapping R is given by the conjunction of equalities $V_1 = v_1 \ \wedge \ \cdots \ \wedge \ V_k = v_k$. Under these assumptions, we can use the following procedure COMPARE to verify that $\mathcal{S}^T$ is a refinement of $\mathcal{S}^S$:

1. Construct a data mapping $\alpha : V_1 = v_1 \wedge \ldots V_i = v_i, \ldots, V_k = v_k$.
2. For each observable variable of the source system $V_i \ \ i = 1, \ldots, k$, construct a verification condition VC_i :

   $$VC_i : \qquad \alpha \quad \Longrightarrow \quad F^S_i(V^S) = F^T_i(V^T)$$

3. Establish the validity of all the verification conditions.

4 The Source Synchronous Programs

The source programs we handle are produced from a high-level synchronous language SCADE [Ver99]. In this language, a *node* is a network of operators. A node has a formal interface (input and output variables) and local variables. The formal interface defines the observable part of the node. The node represents a computation of the values of the output variables as functions of the input variables. Each node has its own distinct output. A SCADE program is a collection of nodes, which are activated by an external loop. The input variables values that are used in each iteration, are the values that were computed in the previous iteration. Thus the order of running the nodes has no effect on the output of the program. Each node is translated into one compilation unit of the C language. The C code of the different nodes is compiled into a machine code. The code of each node usually contains no loops. In rare cases, it may contain loops with constant number of iterations which can easily be unrolled. Our notion of *synchronous program* relates to a program that represents one node, thus containing no loops. We define two types of synchronous programs: synchronous machine code programs, and synchronous C programs.

4.1 Synchronous C Programs

Synchronous C programs are C programs without loops. The following statements are not allowed: *while, for, do-while* and *goto*. The syntax of synchronous C programs is defined as follows:

Let $V = \{v_1, \ldots, v_m\}$ be a set of integer variables, then

- For variable $v_i \in V$ and expression E over V, "$v_i := E$" is an *assignment* statement.
- For boolean expression C and statements S_1, S_2, "**if** C **then** S_1 **else** S_2" is a *conditional statement*.
- If $S_1, \ldots, S_k$ are statements then "$S_1; \ldots; S_k$" is a *concatenation statement*.

Given a synchronous C program, we can define its semantics as an STS. We take V as the set of system variables. As observables, we take an arbitrary subset $O \subseteq V$. The transformer function F is defined inductively for each statement, as follows:

- For an assignment $S = v_i := E$, the transformer is given by
  $F_S(V) = (v_1, \ldots, v_{i-1}, E, v_{i+1}, \ldots, v_m)$.
- For a conditional $S =$ **if** C **then** S^1 **else** S^2, the transformer is given by
  $F_S = (ite(C, F_1^1, F_1^2), \ldots, ite(C, F_m^1, F_m^2))$,
  where $ite(C, a, b)$ is an *if-then-else* expression, and $F^1 = (F_1^1, \ldots, F_m^1)$, $F^2 = (F_1^2, \ldots, F_m^2)$, are the transformers for S^1, S^2, respectively.
- For a concatenation $S = S_1; \ldots; S_k$, the transformer is given by
  $F_S = F_{S_1} \circ \cdots \circ F_{S_m}$.

5 The Target Synchronous Languages

As the target language we take machine code programs.

5.1 Synchronous Machine Code Programs

A *Machine Code Program* is composed of machine instructions. Each instruction has a *label*, which is a non-negative integer. We allow three types of instructions:

- For a variable v and an expression *Exp* of compatible same type, "$v :=$ *Exp*" is an assignment instruction. *Exp* can include calls to functions.
- For a non-negative integer n, "**branch** n" is an *unconditional branch* instruction.
- For a non-negative integer n and a boolean expression C, "**branch** (C) n" is a *conditional branch* instruction with condition C and destination n.

A *machine code program* P is a sequence of *instructions* $P = Inst_{0,\ldots,}Inst_n$. A branch to a label whose value is $\ell > n$, is interpreted as program termination, and is equivalent to a branch with destination $n+1$. To guarantee that the program contains no loops, we require that the destination d of a branch instruction (whether conditional or unconditional) labeled ℓ satisfies $d > \ell$, i.e. we allow only forward branching.

5.2 The STS Semantics of Machine Code Programs

We apply forward analysis to a synchronous program $P = Inst_0, \ldots, Inst_n$, with a set of variables, $V = \{v_0, \ldots v_k\}$, in order to produce its corresponding state transformer. The transformer is computed incrementally. At each step, one instruction is considered, and a transformer is computed, based on the transformer computed for the preceding instruction. We refer to this incremental computation as the ANNOTATION algorithm.

Let $P = Inst_0, Inst_1, \ldots, Inst_n$ be a program, where each $Inst_i, i = 0, \ldots, n$ is a machine instruction. Assume that $V = v_1, ..., v_m$ are the program variables. The ANNOTATION algorithm computes the annotations $Ann[0], ..., Ann[n]$ and the transformers $F^0, F^1, ..., F^n, F^{n+1}$, inductively. For each instruction $i = 0, ..., n$, the annotation $Ann[i]$ expresses the condition under which instruction $Inst_i$ is executed, in terms of the input variables (the values of the variables before the program is started). The transformer F^{i+1} expresses the state transformation effected by the program $P^{i+1} = Inst_0, \ldots, Inst_i$, where F^{n+1} is the transformer for the entire program. The initial transformer F^0 has the trivial form

$$F^0(V) \quad = \quad (v_1, \ldots, v_m)$$

The ANNOTATION **Algorithm** - This algorithm scans the program from $Inst_0$ to $Inst_n$, in order to compute F, the transformer of program P.

1. Initially:

   $Ann[0] = true,\ Ann[1] = \ldots = Ann[n] = false,\ F^0 : \lambda V.V.$

2. For $i = 0, \ldots, n$, examine $inst_i$,

   (a) If $inst_i$ is an assignment instruction $\quad i : v_j := E$,
   then do:

   $$\begin{array}{lcl} Ann[i+1] & := & Ann[i+1] \vee Ann[i] \\ F^{i+1} & := & F^i \circ \lambda V.(v_1, \ldots, v_{j-1},\ ite(Ann[i], E, v_j)\ , v_{j+1}, \ldots, v_m) \end{array}$$

   (b) If $inst_i$ is a conditional branch instruction $\quad$ **branch** $(C)\ j$,
   then do:

   $$\begin{array}{lclclcl} Ann[i+1] & := & Ann[i+1] & \vee & Ann[i] & \wedge & \neg C(F^i(V)) \\ Ann[j] & := & Ann[j] & \vee & Ann[i] & \wedge & C(F^i(V)) \\ F^{i+1} & := & F^i & & & & \end{array}$$

   (c) If $inst_i$ is an unconditional branch instruction $\quad$ **branch** j,
   then do:

   $$\begin{array}{lcl} Ann[j] & := & Ann[j]\ \vee\ Ann[i] \\ F^{i+1} & := & F^i \end{array}$$

Finally, F is taken to be F^{n+1}.

6 The MCVT Tool

MCVT, machine-code validation tool, was developed to validate the translation of synchronous programs. The source programs are C programs, produced automatically from SCADE synchronous programs. The target program is binary optimized code, produced by the commercial compiler DiabData of WindRiver, and intend to run on a PowerPC computer. MCVT accepts the source and target programs and checks that the target is a correct translation of the source. If so, it prints "Valid", otherwise it prints "Invalid". The whole process is fully automatic, though it might be long.

In this section we describe the general architecture of MCVT. For clarity, we use the symbol F_{v_1} to represent the state transformer component that correspond to variable v in the STS under discussion.

6.1 Compact Expression Representation

Using a compressed state transformer function has the disadvantage of producing huge expressions. For example, the transformer of the following machine code program:

$$\begin{array}{l} \texttt{P}: \\ \quad \texttt{0}:\texttt{ro} := \texttt{v}+\texttt{5}; \\ \quad \texttt{1}:\texttt{r1} := \texttt{r0}*\texttt{10}; \\ \quad \texttt{2}:\texttt{v} := \texttt{r0}+\texttt{r1}; \end{array}$$

is $F^P \quad = \quad \lambda(r_0, r_1, v)\ .\ (v+5\ ,\ (v+5)\cdot 10\ ,\ v+5+(v+5)\cdot 10)$. However, internally, each expression has only one copy and, a pointer to this copy is produced whenever the expression is used. Thus, the state transformer of program P is represented by the compact tree that is shown in Fig. 1.

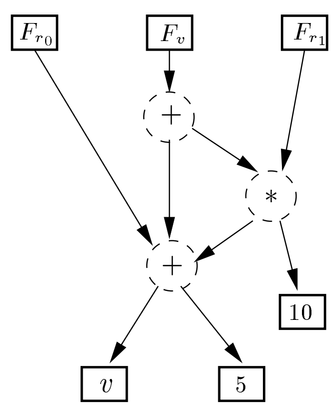

Figure 1: Compact expressions tree

6.2 The Formulas' Simplifier

The annotations that are produced for machine-code programs tend to be big, since each branch instruction adds a term. However, in many cases significant simplifications are possible. A common pattern in the machine-code is the following one:

```
l0:  branch < cond > l2;
l1:  assignment instructions...
     branch l3;
l2:  assignment instructions...
l3:
```

Such a pattern is the result of an $if - else$ statement in the source C code. Let $Ann[l_0] = true$. Then, $Ann[l_1] = \neg cond$, $Ann[l_2] = cond$ and $Ann[l_3] = cond \vee \neg cond = true$. In order to simplify the annotations, the condition expressions are handled symbolically. MCVT replaces every condition by a proposition symbol and simplifies the resulting formulas by resolution. The resolution is not used to validate the formula, but rather to extract an equivalent smaller one. A BDD simplifier could be used as well.

DNF formulas - Let $p_1, p_2, \ldots$ be a set of propositions. The set $\mathcal{L}$ of *literals* is

$$\mathcal{L} ::= p \mid \neg p \mid true \mid false.$$

The $\mathcal{C}$ set of *clauses is*

$$\mathcal{C} ::= l_1 \wedge l_2 \wedge \ldots \mid true \mid false, l_i \in \mathcal{L}.$$

The set $\mathcal{F}$ of DNF - *Disjunctive Normal Form* formulas is

$$\mathcal{F} ::= c_1 \vee c_2, \ldots \mid true \mid false, c_i \in \mathcal{C}.$$

DNF resolution [Rob65, BA01] - For clauses C, C' and a literal l, let $C = l \wedge C^{'}$ be a clause, then $C - l = C^{'}$. An empty clause is equivalent to $true$. If a formula contains a $true$ clause, the whole formula is equivalent to $true$. Let $F = C \vee F^{'}$ be a DNF formula, then $F - C = F'$ and $F = F' + C$. An empty formula is equivalent to $false$. The resolution is based on the observation that if

$$C_1, C_2 \in F, l \in C_1, \neg l \in C_2, C_1 - l = C_2 - \neg l,$$

then

$$F \approx F - C_1 - C_2 + (C_1 - l).$$

6.3 The Code Pattern Recognizer

6.3.1 Special Machine Instructions

Most compilers use only a subset of the processor instruction set. Our implemented tool, as well as the formal semantics definition, consider only those instructions that are used by the compiler. We devote special attention to a few instructions which are used in a special way by the compiler, and which require a special treatment.

The Compare Instruction - The PowerPC processor has eight condition registers (see [PPC02]), each consisting of a triple of bits. The instruction `compare i,< val1 > < val2 >` compares the values of `< val1 > < val2 >` and assigns values to condition register number i. The first bit of the register is set if `< val1 >` is greater than `<val2 >` , the second bit is set if `< val1 >` is smaller than `< val2 >` , and the third bit is set if both values are equal. A bit in the condition register that was not set, is reset. In the state transformer system of a machine

	`add`	`r0,v2,3`	add 3 to v2 and put the result in integer register r0
	`compare`	`cr0,r0,1`	compare r0 to 1 and, set the three bits of condition register cr0
	`branch`	`cr0.2 l2`	branch to l2 if bit no. 2 of cr0 is set
`l1:`	`move`	`v1,0`	move 0 to v1
	`branch`	`l3`	
l2:	`move`	`v1,1`	move 1 to v1
`l3:`			

Figure 2: Machine code example - `v1 := (v2 + 3 == 1)`

program, every bit of a condition register is represented by a boolean variable. The set of condition registers is represented by a two dimensional array of boolean type elements $cr[i,j]$ (also denoted as $cr_{i,j}$), where $i = 0 \ldots 7$ and $j = 0, 1, 2$ which stand for eight triples. The value of each array element is a boolean expression. The `compare` instruction assigns value to one condition register, which consists of three elements of the array. For example, the transformer semantics for the instruction: `compare 1,0,r0` is:

$$\lambda(cr_{1,0}, cr_{1,1}, cr_{1,2}) \, . \, ((r_0 < 0) \, , \, (r_0 > 0) \, , \, (r_0 = 0))$$

The machine code program in Fig. 2 demonstrates a use of the `compare` instruction. The C source code that corresponds to this example is : `v1 := (v2+3=1)`. As the only observable variables are v_1, v_2, the corresponding observable part of the machine code program's transformer is:

$$\lambda(v_1, v_2) \, . \, (ite((v_2 + 3 = 1), 1, 0), v_2)$$

Bit Instructions - Other interesting instructions are those that manipulate bits. For example, the instruction `rlwinm` (Rotate Left Word i bits, then And with Mask) is used by the compiler to extract some bits from an integer, or a condition register. Its semantics is defined, using the functions:

$$bit\_array : array \ of \ booleans \longmapsto integer$$
$$extract\_bits : integer \times integer \longmapsto integer$$

However, when these functions are called in a special sequence with predefined values, their common semantics becomes a specific known arithmetic function. For example, the code in Fig. 3 computes the same value for v_1 as the code in Fig. 2. The corresponding component F_{v_1} is:

$$F_{v_1} \quad = \quad extract(bit\_array(cr \ with(\ [2] : (v_3 + 3 = 1))), 2)$$

which is recognized by MCVT as equivalent to:

$$F_{v_1} \quad = \quad ite((v_3 + 3 = 1), 1, 0)$$

The optimizing compiler tries to choose the best machine-code sequence to perform a high-level language operation. In some cases, it is a sequence of bit manipulation instructions. These sequences are known patterns. We do not want to use the bit manipulation semantics because it complicates the produced state transformer. Therefore, MCVT recognizes these patterns, when they are used in the target machine-code and, replaces them with equivalent high-level operations.

	`add`	`r0,v2,3`	add 3 to v2 and put the result in integer register r0
	`compare`	`cr0,r0,1`	compare r0 to 1 and, set the three bits of condition register cr0
`l1:`	`move`	`r1,cr`	move the content of the condition registers to integer register r1
	`rlwinm`	`v1,r1,2,1`	rotate the content of r1 into v1, so that bit no. 2 moves to the least significant bit and extracts it.

Figure 3: Machine code example - Optimized Machine Code

Let the source C program contain the line: `A := (I=0);`. In order to avoid branch instructions which are expensive in terms of performance, the compiler produces the following instruction sequence:

$\texttt{r}_0$`:=cnlzw I;`	Count the Number of Leading Zeros of I and assign to r_0.
`A:=rlwnm` $\texttt{r}_0, 5, 1$	Rotate r_0 by 5, then extract the first bit.

The execution of these two instructions assigns `A` the value one if `I` equals zero and, zero otherwise. This instruction sequence is identified and replaced by its higher level STS semantics: $F_A = ite(I = 0, 1, 0)$.

6.4 The Data Abstraction

The variables of the source C program are local and global ones. The target program variables are the machine registers, as well as the memory variables, which are referred to as memory addresses. The target system observable variables, are decided by the compiler memory allocation scheme and the calling conventions. At this stage of the research we assume that only the memory is observable and thus, only memory variables are observable. Also, no consistency check is done, to validate that the calling conventions are correctly implemented in the target code. Another assumption relates to the variables' type. We assume that all variables are of integer type and, that all integers are of the same size.

Registers of the target system, as well as local variables of the source system, are not part of the system state transformer due to the use of the annotation algorithm, and state transformer composition. Thus, in order to construct the data abstraction, it is sufficient to map the memory variables of the source system, to their memory addresses. MCVT reads the debug information from the binary code and, uses it to produce the required mapping. Generally, the debug information of an optimized code is not necessarily correct. Yet, it is correct for memory variables at the entry and the exit points of the program. In any case, the soundness of the proof does not depend on the correctness of this information. That is, erroneous information could seldom lead to false positive conclusion.

6.5 Decomposing the Verification Conditions

Due to the use of STS semantics, and the observables mapping extracted from the symbolic information, it is possible to decompose the verification conditions, so as to have one verification condition for each observable variable. We demonstrate this by an example.

Let the source transformer be

$$F^S : \lambda(A, B) \,.\, (B + 1) \;,\; ite(A > 5, 1, 2))$$

the target transformer

$$F^T : \lambda(mem_0, mem_4) \,.\, (mem_4 + 1 \;,\; ite(\neg(mem_0 \leq 5), 2, 1))$$

and the data abstraction mapping be the following: $mem_0 = A \wedge mem_4 = B$, then the decomposition module produces the following two verification conditions:

$$VC_1 : mem_0 = A \wedge mem_4 = B \rightarrow mem_4 + 1 = B + 1$$

$$VC_2 : mem_0 = A \wedge mem_4 = B \rightarrow ite(mem_0 \leq 5,\ 2,\ 1) = ite\ (A > 5, 1, 2)$$

6.6 Adaptation to CVC

In certain cases CVC does not satisfy MCVT needs. One example is the type of variables. *Simple types* of CVC are either *real* or *boolean*. For real variables i, j, the formula $i > j \leftrightarrow i \geq j + 1$, is not valid. However, it is valid for integer variables i, j and, the compiler may use this equivalence to optimize the code. To handle this problem, MCVT adds the needed assertions (e.g. $(i > j) = (i \geq j + 1)$), whenever an equality of this kind is encountered.

Another problem is that CVC does not support multiplication commutativity, when both multiplication operands are variables. MCVT adds the needed assertions.

6.7 The Verifier Module (CVC)

Within the scope of this research, we had to choose a validity checker that is capable of automatically validating the verification conditions. The verification conditions that MCVT produces are formulas, which include: first order logic operators (without quantifiers), Presburger Arithmetic, uninterpreted functions and array operations. The following tools were explored:

1. (Stanford) - The Stanford Temporal Prover [BBC+95].

2. Omega (University of Maryland) - A system for verifying Presburger formulas as well as a class of logical formulas and, for manipulating linear constraints over integer variables [KMP+].

3. CVT/C+tlv+tlvp (Weizmann) - A set of tools which provides a range minimizing module, an environment that supports model checking and, a validator for Presburger arithmetic [PSS98c], [PS96].

4. ICS (SRI International) - Integrated Canonizer and Solver. (We checked the alpha version) [FORS01].

5. CVC (Stanford) - Cooperative Validity Checker - Supports a combination of decision procedures for the validity of quantifier-free first-order formulas, with theories of extensional arrays, linear real arithmetic and uninterpreted functions [SBD02].

number of modules	**Number of Lines**	**Validation Time** [seconds]
1	266	1221.82
33	144-155	0.05-0.21
1	6414	0.68
1	6287	0.35
1	6286	0.82
1	6286	0.86
1	6285	0.77
1	6286	0.85
1	6286	0.87
1	6336	32.98
1	6336	32.34

Figure 4: Run time of a jet engine translation validation

After a thorough examination of the above tools, CVC was found to best fit our needs, due to its reliability and the wide range of decision procedures it supports. It combines independent cooperating decision procedures for a wide set of theories, into a decision procedure for the combination of the theories, based on a version of the Nelson-Oppen [NO79] framework. Appendix A demonstrates the application of MCVT on a small synchronous program and its translation. The C source program, the target machine-code, and one verification file (which is one verification condition ready for CVC) are listed there.

7 Conclusions

In this paper we have presented an approach to the validation of translations from synchronous C to synchronous machine code. We described its implementation in the tool MCVT. Our goal is to use MCVT for the translation validation of real industrial application. Thus it has been used to validate the translation of a program developed for training, as part of the SafeAirI project. This program contains the code for a railroad crossing controller. It handles the passage of a train in a one-way railroad crossing. The system consists of a pair of reliable sensors that indicate train entering and exiting the crossing region, a signal for entering trains and a gate for blocking passage of cars from a side road. The verification took 200 seconds on a Linux workstation with i686 CPU.

Currently, we are working on validating the translation of jet engine controller application from Hispano-Suiza. MCVT validates the translation done by the commercial compiler DiabData of WindRiver, which is the compiler used by Hispano-Suiza for the development of this system. Since the jet engine is a safety critical system, it is not allowed to use an optimizer, when compiling the software. As opposed to our expectation, the verification time is increased compared to the verification of the optimized translation. The reason for these surprising results is that the unoptimized assembly code is significantly longer, and contains many redundant machine instructions, which are otherwise eliminated by the optimizer.

Fig. 4 summaries the results of this effort, (running MCVT on a Linux workstation with i686 CPU). For most of the nodes, translation was validated in fractions of minutes. When the size of the module increases, the verification time increases significantly. MCVT fails to validate the translation of some of the modules due to the limited C syntax it currently supports (e.g. `case` statements and floating point constants are not supported). Another limitation is that only a subset of the PowerPC machine code is supported.

We plan to use a new updated version of CVC, CVC Lite [cvc] from Stanford University, which is faster and allows further simplification of the verification conditions. We also plan to extend the set of supported synchronous programs.

References

[BA01] M. Ben-Ari. *Mathematical Logic for Computer Science.* springer Verlag London, 2001.

[BBC+95] N. Bjørner, I.A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[cvc] Cvc lite home page. http://chicory.stanford.edu/CVCL.

[Dia] WindRiver. *DiabC/C++ Compiler for PowerPC, user's guide.*

[EdR+99] K. Engelhardt, W.P. de Roever, et al. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1999.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[FORS01] J.C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. To be presented at CAV'2001, 2001.

[GZ99] G. Goos and W. Zimmermann. Verification of compilers. In Bernhard Steffen and Ernst Rdiger Olderog, editors, *Correct System Design*, volume 1710, pages 201–230. Springer-Verlag, Nov 1999.

[GZ00] G. Goos and W. Zimmermann. Verifying compilers and asms. In Yuri Gurevich, Philipp W. Kutter andMartin Odersky, and Lothar Thiele, editors, *Abstract State Machines, Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 177–202. Springer, Mar 2000.

[GZG00] T. Gaul, W. Zimmermann, and W. Goerigk. Practical Construction of Correct Compiler Implementations by Runtime Result Verification. In *Proceedings of SCI'2000, International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, 2000.

[KMP+] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct 1979.

[PPC02] *Book E- Enhanced PowerPC Architecture*, 2002.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *R. Alur and T. Henzinger, editors,* Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 184–195, 1996.

[PSS98a] A. Pnueli, N. Shankar, and E. Singerman. Fair synchronous transition systems and their liveness proofs. In A.P. Ravn and H. Rischel, editors, *FTRTFT 98: 5th International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems*, volume 1486 of *Lect. Notes in Comp. Sci.*, pages 198–209. Springer-Verlag, 1998.

[PSS98b] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[PSS98c] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 235–246. Springer-Verlag, 1998.

[PSS99] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)- automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2, 1999.

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, Jan 1965.

[SBD02] A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.

[SSB01] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation.* Springer-Verlag, 2001.

[Ver99] Verilog. *SCADE Language Reference Manual - Version 3.0*, January 1999.

[ZG97] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *j-jucs*, 3(5):504–567, may 1997.

[ZPFG02] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: a translation validator for optimizing compilers. In *Proc. of the workshop: Compiler Optimization Meets Compiler Verification (COCV'02), Satellite of ETAPS'02*, pages 6–22, Grenoble, April 2002.

C Program	Machine-Code Program		
typedef struct {	ACS:	lwz	r12,0(r3)
int I3;	ACS+4:	lwz	r6,12(r3)
int I4;	ACS+8:	addi	r12,r12,-51
int out1;	ACS+12:	cmpli	0,r12,149
int gc; }	ACS+16:	addi	r5,r6,123
vars_rec;	ACS+20:	addi	r4,r6,567
int ACS (vars_rec *_C_) {	ACS+24:	lwz	r6,4(r3)
int c1; int c2;	ACS+28:	bc	4,0,ACS+48
c1 = 123+_C_->gc;	ACS+32:	cmpi	0,r6,0
c2 = 567+_C_->gc;	ACS+36:	bc	12,2,ACS+48
if (((_C_->I3) < 200)&&	ACS+40:	stw	r5,8(r3)
(50 < (_C_->I3)) &&	ACS+44:	b	ACS+52
_C_->I4)	ACS+48:	stw	r4,8(r3)
{ _C_->out1 = c1; }	ACS+52:	cntlzw	r12,r6
else {_C_->out1 = c2; }	ACS+56:	rlwinm	r12,r12,27,5,31
_C_- > I4=(_C_->I4==0);	ACS+60:	stw	r12,4(r3)
return (1);	ACS+64:	addi	r3,r0,1
}	ACS+68:	bclr	20,0

Figure 5: C program and its translation to machine code

A Translation Validation of a C Program.

Fig. 5 shows a C program and its optimized translation. The machine code produced, by the compiler, for $\_C\_- > I3 < 200\&\&\_C\_- > I3 > 50$, contains a single unsigned comparison. The machine code produced for the assignment $\_C\_- > I4 = \_C\_- > I4 == 0$ is the sequence : `cntlzw` (count leading zeros), `rlwinm` (rotate with mask). MCVT recognizes the sequence and, treats it properly, even when the two instructions are not adjacent. In Fig. 6 one of the input files to CVC is presented (without the variable declaration part).

```
%Machine code conditions
ASSERT p1 = (((MEMORY0+-51)>=0) AND ((MEMORY0+-51)<149));
ASSERT p2 = (IF (p1) THEN
(MEMORY4=0) ELSE
((-1>=(MEMORY0+-51)) OR ((MEMORY0+-51)=149)) ENDIF) ;

%C code conditions
ASSERT pc1 = (((I3<200) AND (50<I3)) AND ( NOT I4=0));

Target state transformer for one variable
ASSERT PMEMORY8 = (IF (( NOT p1 OR (p1 AND p2))) THEN
(MEMORY12+567) ELSE
(IF ((p1 AND NOT p2)) THEN
(MEMORY12+123) ELSE
MEMORY8 ENDIF) ENDIF) ;

%Data mapping
ASSERT Pout1 = PMEMORY8;
ASSERT gc = MEMORY12;
ASSERT I3 = MEMORY0;
ASSERT Pgc = PMEMORY12;
ASSERT I4 = MEMORY4;
ASSERT PI3 = PMEMORY0;
ASSERT PI4 = PMEMORY4;
ASSERT out1 = MEMORY8;

%Integer assertions
ASSERT (((MEMORY0+-51)>=0)=((MEMORY0+-51)>(0-1)));
ASSERT (((MEMORY0+-51)<149)=(149>=((MEMORY0+-51)+1)));
ASSERT ((-1>=(MEMORY0+-51))=(-1>((MEMORY0+-51)-1)));

%Source state transformer
QUERY Pout1 = (IF (pc1) THEN
(123+gc) ELSE
(567+gc) ENDIF) ;
```

Figure 6: One CVC input file - verification condition for one variable