Translation Validation: from SIGNAL to C * **

A. Pnueli O. Shtrichman M. Siegel

Weizmann Institute of Science, Rehovot, Israel

Abstract. *Translation validation* is an alternative to the verification of translators (compilers, code generators). Rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program. In order to be a practical alternative to compiler verification, a key feature of this validation is its *full automation*.
Since the validation process attempts to "unravel" the transformation effected by the translators, its task becomes increasingly more difficult (and necessary) with the increase of sophistication and variety of the optimizations methods employed by the translator. In this paper we address the practicability of translation validation for highly optimizing, industrial code generators from SIGNAL, a widely used synchronous language, to C. We introduce new abstraction techniques as part of the automation of our approach.

1 Introduction

A significant number of embedded systems contain safety-critical aspects. There is an increasing industrial awareness of the fact that the application of formal specification languages and their corresponding verification/validation techniques may significantly reduce the risk of design errors in the development of such systems. However, if the validation efforts are focused on the specification level, the question arises how can we ensure that the quality and integrity achieved at the specification level is safely transferred to the implementation level. Today's process of the development of such systems consists of hand-coding followed by extensive unit and integration-testing.

The highly desirable alternative, both from a safety and a productivity point of view, to automatically generate code from verified/validated specifications, has failed in the past due to the lack of technology which could convincingly demonstrate to certification authorities the correctness of the generated code. Although there are many examples of compiler verification in the literature (see,

* This research was done as part of the ESPRIT project SACRES and was supported in part by the Minerva Foundation and an infra-structure grant from the Israeli Ministry of Science and Art

** Preliminary versions of some parts of this paper were published before in [17], [19] and [20] for-example, [5, 9, 10, 15, 12, 11, 14, 13]), the formal verification of industrial code generators is generally prohibitive due to their size. Another problem with compiler verification is that the formal verification freezes the design and evolution of the compiler, as each change to the code generators nullifies their previous correctness proof.

Alternately, code-validation suggests to construct a fully automatic tool which establishes the correctness of the generated code individually for each run of the code generator. In general, code-validation can be the key enabling technology to allow the usage of code generators in the development cycle of safety-critical and high-quality systems. The combination of automatic code generation and validation improves the design flow of embedded systems in both safety and productivity by eliminating the need for hand-coding of the target code (and consequently coding-errors are less probable) and by considerably reducing unit/integration test efforts.

Of course, it is not clear that every compiler and every source and target languages can be verified according to the code-validation paradigm. But the fact that the compiler we considered was highly optimized and the source and target languages had completely different structures (synchronous versus sequential code) indicates that this method has the potential of solving realistic, non-trivial cases.

The work carried out in the SACRES project proves the feasibility of code-validation for the industrial code generator used in the project, and demonstrates that industrial-size programs can be verified fully automatically in a reasonable amount of time.

1.1 Technical Introduction

In this paper we consider translation validation for the synchronous languages Signal [3]. This language is mainly used in industrial applications for the development of safety-critical, reactive systems. In particular, it is designed to be translatable into code which is as time/space efficient as handwritten code. This code is generated by sophisticated code generators which perform various analyses/calculations on the source code in order to derive highly efficient implementations in languages such as C, ADA, or JAVA.

The presented translation validation approach addresses two industrial compilers from SIGNAL to C. These compilers – which apply more than 100 optimization rules during code generation [16] – were developed in the ESPRIT project SACRES by the French company TNI and by Inria (Rennes) and are used by Siemens, SNECMA and British Aerospace. Their formal verification is prohibitive due to their size (more than 20,000 lines of code each) and the fact that they are constantly improved/extended.

While developed in the context of code generators for synchronous languages, the proposed method has wider applicability. The main feature which enables us to perform the validation task algorithmically is that the source language has a restricted explicit control structure. This is also represented by the fact that the resulting C-code consists of a single main loop whose body is a loop-free program. Source languages with these features can benefit from the method proposed in this paper. For example, the language UNITY [6] which comes from the world of asynchronous distributed systems is another possible client of the proposed method.

We present a common semantic model for SIGNAL and C, introduce the applied notion of *being a correct implementation*, formulate the correctness of the generated C code as proof obligations in first order logic, and present efficient decision procedures to check the correctness of the generated proof obligations. All translations and constructions which are presented in the course of the paper have been implemented in a tool called CVT (Code Validation Tool) [19]. CVT has been used to validate the code generated from a 6000 lines SIGNAL program with more than 1000 variables. This program is a turbine-control system which was developed as an industrial case study by SNECMA in the SACRES project.

A major advantage of a carefully designed translation validation tool is that it can replace the need for correctness proofs for *various* compilers if these compilers are based on the same definition of "correct code generation". This is the case for the TNI and the Inria compiler and, indeed, CVT is used to validate code originating from either of these two compilers.

1.2 Run Time Result Verification

There is a growing interest in the concept of verifying run-time results, rather than programs or models. A recently held workshop entitled "run-time result verification" presented various applications of run-time verification techniques. These applications, in most cases, had similar characteristics to compiler verification, although they come from a variety of domains: it is very hard, or even impossible to formally verify them in the 'traditional', single-time proof on the one hand, and on the other hand, their design often changes while the system is developed. Another motivation for this approach is that in some cases the program itself is not available for inspection due to commercial and intellectual property factors[1]. A good example of such a domain is the formal verification of decision procedures. Implementations of Decision procedures are often experimental and not very robust. They are typically complex and keep evolving over time. Being generic tools for verifying safety-critical systems, the correctness of the tools is at least as important as that of the applications they help to verify. In [21], the decision procedure generates proofs during run-time, in order to validate the decision process. An axiomatization of the proof system is presented in another formal system, which is referred to as the 'logical framework' in which the proof is carried out. The logical framework tool includes a proof checker of its own, that can check proofs of any system that is axiomatized in the framework. This enables automatic validation of every run of the decision procedure

[1] this is also a common argument in the domain of testing, where often 'black-box' testing is used rather than 'white-box' testing simply because the code is not available for inspection

by producing a proof script for the run and applying to it the framework's proof checker.

The concept of validating translators/compilers by proving semantical equivalence between source and target code has been applied in various other projects. In [22] and [8] the verification of the translator from a high level description language to executable code is carried out in two distinct stages: in the 'on-line' stage, each run of the translator is verified by proving several simple conditions on the syntactical structure of the results, in comparison to the source code. This stage is fully automatic and implemented by a simple 'checker'. The reduction from full translation verification to these simple rules is carried out manually in a one-time effort (the 'off-line' stage) by an expert. This approach was applicable in their case due to the relatively structured translation scheme of the source code. A declaration in the source code is translated into code which performs RNS transformations (it reduces arithmetical operations to a set of independent arithmetic operations on integers of limited size). Thus, the main task of the validation process is to verify that these transformations are correct. After computing a mapping between the input and output variables of the source and target programs, they prove that the correspondence between the 'semantic' states of the two programs is preserved by the execution steps. In this sense there model is similar to the model we will present in this paper. In [7], translation validation is done on a purely syntactic level. Their method is based on finding a bijection between abstract and concrete instruction sets (resp. variables) because they are considering a *structural* translation of one sequential program into another sequential program. Since we are dealing with optimizing compilers we have to employ a far more involved *semantic* approach.

The rest of the paper is organized as follows. In Section 2 we give a brief introduction to SIGNAL and present a running example. After introducing *synchronous transition systems* as our model of computation in Section 3 we address the formal semantics of SIGNAL. Section 4 presents the concepts which underly the generation of the proof obligations. In Section 5 we present the decision procedure to check the validity of these proof obligations. Finally, Section 7 contains some conclusions and future perspectives.

2 An Illustrative Example

In this section we first illustrate details of the compilation process by means of an example and then explain the principles which underly the translation validation process.

A SIGNAL program describes a reactive system whose behavior along time is an infinite sequence of *instants* which represent reactions, triggered by external or internal events. The main objects manipulated by a SIGNAL program are *flows*, which are sequences of values synchronized with a *clock*. A flow is a typed object which holds a value at each instant of its clock. The fact that a flow is currently absent is represented by the bottom symbol $\perp$ (cf. [3]). Clocks are unary flows, assuming the values $\{\mathrm{T}, \perp\}$. A clock has the value T if and only if the flow associated with the clock holds a value at the present instant of time. Actually, any expression *exp* in the language has its corresponding clock *clk*(*exp*) which indicates whether the value of the expression at the current instant is different from $\perp$.

Besides *external* flows (input/output flows), which determine the interface of the SIGNAL program with its environment, also *internal* flows are used and manipulated by the program. Consider the following SIGNAL program DEC:

```
process DEC=
     ( ? integer FB
       ! integer N
     )
       (| N:= FB default (ZN-1)
        | ZN:= N $ init 1
        | FB^=when (ZN<=1)
        |)
     where
        integer ZN init 1 ;
end
```

Program DEC (standing for "decrement") has an input FB and an output N, both declared as integer variables. Now and then, the environment provides a new input via variable FB. Receiving a new positive input, the program starts an internal process which outputs the sequence of values FB, FB-1, ..., 2, 1 via output variable N. After outputting 1, the program is ready for the next input. This program illustrates the capability of a multi-clock synchronous language to generate a new clock (the clock of the output N) which over-samples the input clock associated with FB. The program uses the local variable ZN to record the previous value of N.

The body of DEC is composed of three statements which are executed concurrently as follows. An input FB is read and copied to N. If N is greater than 1 it is successively decremented by referring to ZN, which holds the previous value of N (using $ to denote the "previous value" operator) . No new input value for FB is accepted until ZN becomes (or is, in case of a previous non-positive input value for FB) less than or equal to 1. This is achieved by the statement

```
FB^=when (ZN<=1),
```

which is read "the *clock* of FB is on when ZN ≤ 1", and allows FB to be present only when ZN ≤ 1. When the clock of FB is off, the `default` action of assigning N the value of `ZN-1` is activated. A possible computation of this program is:

$$\begin{pmatrix} \mathrm{FB}:\perp \\ \mathrm{N}:\perp \\ \mathrm{ZN}:1 \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{FB}:3 \\ \mathrm{N}:3 \\ \mathrm{ZN}:1 \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{FB}:\perp \\ \mathrm{N}:2 \\ \mathrm{ZN}:3 \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{FB}:\perp \\ \mathrm{N}:1 \\ \mathrm{ZN}:2 \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{FB}:5 \\ \mathrm{N}:5 \\ \mathrm{ZN}:1 \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{FB}:\perp \\ \mathrm{N}:4 \\ \mathrm{ZN}:5 \end{pmatrix} \rightarrow \ldots$$

Where $\perp$ denotes the absence of a signal. Note, that SIGNAL programs are not expected to terminate.

2.1 Compilation of Multi-clocked Synchronous Languages

The compilation scheme for SIGNAL to an imperative, sequential languages (s.a. C, ADA) proceeds as follows. The statements of a SIGNAL program P form a *Set of Logical Equations* (SLE) on the flows of P and their associated clocks. Solutions of SLE for a given set of input/register values determine the next state of the system. The compiler derives from P an imperative program C which consists of one main loop whose task is to repeatedly compute such solutions of the SLE. In order to do so, the compiler computes from SLE a *conditional* dependency graph on flows and another linear equation system – the, so called, *clock calculus* [3] – which records the dependencies amongst clocks. The produced code contains statements originating from the clock calculus and assignments to variables (representing the flows of P) whose order must be consistent with the dependency graph. These assignments are performed if the corresponding flow is currently present in the source program, i.e. the clocks of flows determine the control structure of the generated program.

The C program which is generated by the compiler from the DEC program consists of a main program containing two functions:

- An *initialization* function, which is called once to provide initial values to the program variables.
- An *iteration* function which is called repeatedly in an infinite loop. This function, whose body calculates the effect of one synchronous "step" of the abstract program, is the essential part of the concrete code.

The iteration function obtained by compiling DEC is given by:

`logical DEC_iterate()`

{
l0: $\mathrm{h1}_c = \mathrm{TRUE};$
l1: $\mathrm{h2}_c = \mathrm{ZN}_c <= 1;$
l2: if $(\mathrm{h2}_c)$
l2.1: read(FB_c);
l3: if $(\mathrm{h2}_c)$
l3.1: $\mathrm{N}_c = \mathrm{FB}_c;$
else
l3.2: $\mathrm{N}_c = \mathrm{ZN}_c - 1;$
l4: write(N_c);
l5: $\mathrm{ZN}_c = \mathrm{N}_c;$
return TRUE;
}

The labels in function DEC_iterate() are not generated by the compiler but have been added for reference. We added 'c' as subscript for the program variables, to distinguish them from the SIGNAL variables.

The C-code introduces explicit boolean variables to represent the clocks of SIGNAL variables. Variable $\mathrm{h1}_c$ is the clock of N_c and ZN_c, and $\mathrm{h2}_c$ is the clock of FB_c.

The C program works as follows. If $\mathrm{h2}_c$, the clock of FB_c, has the value T, a new value for FB is read and assigned to the variable N_c. If $\mathrm{h2}_c$ is F, N_c gets the value $\mathrm{ZN}_c - 1$. In both cases the updated value of N is output (at l_4) and also copied into ZN_c, for reference in the next step .

A computation of this program is given below. We skip intermediate states and consider complete iterations of the while loop. The notation $X : *$ is used to denote that variable X has an arbitrary value.

$$\begin{pmatrix} \mathrm{FB} : * \\ \mathrm{N} : * \\ \mathrm{ZN} : 1 \\ \mathrm{h1} : * \\ \mathrm{h2} : * \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathrm{FB} : 3 \\ \mathrm{N} : 3 \\ \mathrm{ZN} : 3 \\ \mathrm{h1} : \mathrm{T} \\ \mathrm{h2} : \mathrm{T} \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathrm{FB} : 3 \\ \mathrm{N} : 2 \\ \mathrm{ZN} : 2 \\ \mathrm{h1} : \mathrm{T} \\ \mathrm{h2} : \mathrm{F} \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathrm{FB} : 3 \\ \mathrm{N} : 1 \\ \mathrm{ZN} : 1 \\ \mathrm{h1} : \mathrm{T} \\ \mathrm{h2} : \mathrm{F} \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathrm{FB} : 5 \\ \mathrm{N} : 5 \\ \mathrm{ZN} : 5 \\ \mathrm{h1} : \mathrm{T} \\ \mathrm{h2} : \mathrm{T} \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathrm{FB} : 5 \\ \mathrm{N} : 4 \\ \mathrm{ZN} : 4 \\ \mathrm{h1} : \mathrm{T} \\ \mathrm{h2} : \mathrm{F} \\ \pi : l_0 \end{pmatrix} \xrightarrow{*} \cdots$$

Taking into account that $\mathrm{h1}_c$ is the clock of N_c and that $\mathrm{h2}_c$ is the clock of FB_c, we have an accurate state correspondence between the computation of the SIGNAL program and the computation of the C-code, when we restrict our observations to subsequent visits at location l_0.

This state correspondence is a general pattern for programs generated by the SACRES compiler. Intuitively, the generated C-code correctly implements the original SIGNAL program if the sequence of states obtained at the designated control location l_0 corresponds to a possible sequence of states in the abstract system.

In the rest of the paper, we show how this approach can be formalized and yield a fully automatic translation validation process.

3 Computational Model and Semantics of SIGNAL

In order to present the formal semantics of SIGNAL we introduce a variant of *synchronous transition systems* (STS) [20]. STS is the computational model of our translation validation approach.

Let V be a set of typed variables. A *state* s over V is a type-consistent interpretation of the variables in V. Let Σ_V denote the set of all states over V. A *synchronous transition system* $A = (V, \Theta, \rho)$ consists of a finite set V of typed variables, a satisfiable assertion Θ characterizing the initial states of system A, and a transition relation ρ. This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma_V$ to its possible successors $s' \in \Sigma_V$ by referring to both unprimed and primed versions of variables in V. Unprimed variables are interpreted according to s, primed variables according to s'. To the state space of an STS A we refer as Σ_A. We will also use the term "system" to abbreviate "synchronous transition system". Some of the variables in V are identified as *volatile* while the others are identified as *persistent*. Volatile variables represent flows of SIGNAL programs, thus their domains contain the designated element $\perp$ to indicate absence of the respective flow.

For a variable $v \in V$ we write $clk(v)$ and $clk(v')$ to denote the inequalities $v \neq \perp$ and $v' \neq \perp$, implying that the signal v or v' is present.

A *computation* of $A = (V, \Theta, \rho)$ is an infinite sequence $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$, with $s_i \in \Sigma_V$ for each $i \in \mathbb{N}$, which satisfies $s_0 \models \Theta$ and $\forall i \in \mathbb{N}.\ (s_i, s_{i+1}) \models \rho$. Denote by $\|A\|$ the set of computations of the STS A.

Before we describe how to construct an STS Φ_P corresponding to a given SIGNAL program P, let us first describe the primitives of a SIGNAL program.

3.1 A Sketch of SIGNAL Primitives

SIGNAL supports the following primitives.

1.	$v := f(u_1, \ldots, u_n)$	function extended to sequences
2.	$w := v$ **\$ init** w_0	shift register
3.	$v := u$ **when** b	data dependent down-sampling
4.	$w := u$ **default** v	merging with priority
5.	$P \vert Q$	program composition

In these primitives, u, v, w, b denote typed *signals*, i.e., sequences of values of the considered type extended with the special symbol $\perp$. In the **when** expression, the b signal is assumed boolean. In the last instruction, both P and Q denote SIGNAL programs.

The assignment $v := f(u_1, \ldots, u_n)$ can only be applied to signals $u_1, \ldots, u_n$ which share the same clock. It defines a new signal v with the same clock such that, for every $i = 1, 2, \ldots,\ v[i] = f(u_1[i], \ldots, u_n[i])$.

The assignment $w := v$ **\$ init** w_0 defined a signal w with the same clock as v and such that $w[1] = w_0$ and, for every $i > 1$, $w[i] = v[i-1]$.

The assignment $v := u$ **when** b defines a signal whose values equal the value of u at all instances in which u and b are both defined and $b = true$.

The assignment $w := u$ **default** v defines a signal w which equals u whenever u is defined and equals v at all instance in which v is defined while u is absent.

Finally, $P|Q$ is the composition of the programs (set of statements) P and Q.

3.2 System Variables

The system variables of Φ are given by $V = U \cup X$, where U are the SIGNAL *signals* explicitly declared and manipulated in P, and X is a set of auxiliary *memorization variables*, whose role is explained below.

3.3 Initial Condition

The initial condition for Φ is given by

$$\Theta\colon \quad \bigwedge_{u \in U} u = \perp$$

By convention, the initial value of all declared signal variables is $\perp$. The initialization of memorization variables is explained in the following subsection.

3.4 The Transition Relation and its Properties

The composition | of SIGNAL programs corresponds to logical conjunction. Thus, the transition relation ρ will be a conjunction of assertions where each SIGNAL statement gives rise to a conjunct in ρ. Below, we list the statements of SIGNAL and present for each of them the conjunct it contributes to the transition relation.

- Consider the SIGNAL statement $v := f(u_1, \ldots, u_n)$, where f is a state-function. This statement contributes to ρ the following conjunct:

  $$\begin{array}{ll} & clk(u_1') \equiv \ldots \equiv clk(u_n') \\ \wedge & v' = \textbf{if } clk(u_1') \textbf{ then } f(u_1', \ldots, u_n') \textbf{ else } \perp \end{array}$$

  This formula requires that the signals $v, u_1, \ldots, u_n$ are present at precisely the same time instants, and that at these instants $v = f(u_1, \ldots, u_n)$.

- The statement

  $$r := v \,\$\, \textbf{init}\, w_0$$

  contributes to ρ the conjunct:

  $$\begin{array}{lll} & m.r' = & \text{if } \; clk(v') \text{ then } \; v' \text{ else } \; m.r \\ \wedge & r' = & \text{if } \; clk(v') \text{ then } \; m.r \text{ else } \; \perp \end{array}$$

  This definition introduces a memorization variable $m.r$ which stores the last (including the present) non-bottom value of v. Variable $m.r$ is initialized in the initial condition Θ to w_0. From now on we refer to flows r that are defined by this type of statement as *register flows*. Variables in an STS which represent register flows will typically be denoted by r, and the corresponding memorization variables by $m.r$. Note that unlike the other system variables in the constructed STS, memorization variables are *persistent*.

- The statement

  $$v := u \textbf{ when } b$$

  contributes to ρ the conjunct:

  $$v' = \text{ if } \; b' = \mathrm{T} \text{ then } \; u' \text{ else } \; \perp.$$

- The statement

  $$w := u \textbf{ default } v$$

  contributes to ρ the conjunct:

  $$w' = \text{ if } \; clk(u') \text{ then } \; u' \text{ else } \; v'.$$

According to the above explanations, the SIGNAL program DEC is represented by the following STS $A = (V_{\mathrm{a}}, \Theta_{\mathrm{a}}, \rho_{\mathrm{a}})$.

$$V = \{\mathrm{FB},\ \mathrm{N},\ \mathrm{ZN},\ \mathrm{m.ZN}\}$$

$$\Theta = (\mathrm{FB} = \bot\ \wedge\ \mathrm{N} = \bot\ \wedge\ \mathrm{ZN} = \bot\ \wedge\ \mathrm{m.ZN} = 1)$$

$$\rho_a = \begin{pmatrix} & \mathrm{N}' & = \text{if} & \mathrm{FB}' \neq \bot & \text{then} & \mathrm{FB}' & \text{else} & \mathrm{ZN}' - 1 \\ \wedge & \mathrm{m.ZN}' & = \text{if} & \mathrm{N}' \neq \bot & \text{then} & \mathrm{N}' & \text{else} & \mathrm{m.ZN} \\ \wedge & \mathrm{ZN}' & = \text{if} & \mathrm{N}' \neq \bot & \text{then} & \mathrm{m.ZN} & \text{else} & \bot \\ \wedge & \mathrm{ZN}' \leq 1 & \Leftrightarrow & \mathrm{FB}' \neq \bot & & & & \end{pmatrix}$$

In the following sections, we assume that the type definitions for variables also specify the "Signal type" of variables, i.e. whether they are input, output, register, memorization or local variables. The respective sets of variables are denoted by I, O, R, M, L. Combinations of these letters stand for the union of the respective sets; e.g. IOR stands for the set of input/output/register variables of some system. Note that the M variables are not originally present in the Signal program, but are introduced by its translation into the STS notation.

For the translation validation process, the generated C programs is also translated into the STS formalism. Below, we present the STS representation of the DEC_iterate() generated code, where the predicate *pres-but*(U) indicates that all variables in the set $V \setminus U$ preserve their values during the respective transition. Thus, the values of all variables except, possibly, those in U are preserved.

$C = (V_c, \Theta_c, \rho_c)$ where

$$V = \{\mathrm{FB}_c,\ \mathrm{N}_c,\ \mathrm{ZN}_c,\ \mathrm{h1}_c,\ \mathrm{h2}_c\}$$

$$\Theta = (\mathrm{ZN}_c = 1\ \wedge\ pc = l_0)$$

$$\rho_c = \begin{pmatrix} & (pc = l_0 & \wedge & \mathrm{h1}'_c = \mathrm{T} & \wedge\ pc' = l_1 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{h1}_c)) \\ \vee & (pc = l_1 & \wedge & \mathrm{h2}'_c = (\mathrm{ZN}_c \leq 1) & \wedge\ pc' = l_2 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{h2}_c)) \\ \vee & (pc = l_2 & \wedge & \mathrm{h2}_c & \wedge\ pc' = l_{2.1} & \wedge\ \textit{pres-but}(\mathrm{pc})) \\ \vee & (pc = l_2 & \wedge & \neg\mathrm{h2}_c & \wedge\ pc' = l_3 & \wedge\ \textit{pres-but}(\mathrm{pc})) \\ \vee & (pc = l_{2.1} & \wedge & & pc' = l_3 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{FB}_c)) \\ \vee & (pc = l_3 & \wedge & \mathrm{h2}_c & \wedge\ pc' = l_{3.1} & \wedge\ \textit{pres-but}(\mathrm{pc})) \\ \vee & (pc = l_3 & \wedge & \neg\mathrm{h2}_c & \wedge\ pc' = l_{3.2} & \wedge\ \textit{pres-but}(\mathrm{pc})) \\ \vee & (pc = l_{3.1} & \wedge & \mathrm{N}'_c = \mathrm{FB}_c & \wedge\ pc' = l_4 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{N}_c)) \\ \vee & (pc = l_{3.2} & \wedge & \mathrm{N}'_c = \mathrm{ZN}_c - 1 & \wedge\ pc' = l_4 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{N}_c)) \\ \vee & (pc = l_4 & \wedge & & pc' = l_5 & \wedge\ \textit{pres-but}(\mathrm{pc})) \\ \vee & (pc = l_5 & \wedge & \mathrm{ZN}'_c = \mathrm{N}_c & \wedge\ pc' = l_0 & \wedge\ \textit{pres-but}(\mathrm{pc}, \mathrm{ZN}_c)) \end{pmatrix}$$

Note, that the C programs use *persistent* variables (i.e. variables which are never absent) to implement SIGNAL programs which use volatile variables. This has to be taken into account when defining the notion of "correct implementation" in the next section.

4 The "Correct Implementation" Relation

The notion of *correct implementation* used in this work is based on the general concept of refinement between synchronous transition systems.

Let $A = (V_A, \Theta_A, \rho_A, \mathcal{O}^{\mathrm{A}})$ and $C = (V_C, \Theta_C, \rho_C, \mathcal{O}^{\mathrm{C}})$ be an abstract and concrete STS's, where $\mathcal{O}^{\mathrm{A}}$ and $\mathcal{O}^{\mathrm{C}}$ are *observation functions*, respectively mapping the abstract and concrete states into a common data domain $\mathcal{D}$.

An *observation* of STS is any infinite sequence of $\mathcal{D}$-elements which can be obtained by applying the observation function $\mathcal{O}^{\mathrm{A}}$ to each of the states in a computation of A. That is, a sequence which has the form $\mathcal{O}^{\mathrm{A}}(s_0), \mathcal{O}^{\mathrm{A}}(s_1), \ldots$, for some $\sigma : s_0, s_1, \ldots$, a computation of A. We denote by $Obs(A)$ the set of observations of system A. In a similar way, we define $Obs(C)$ the set of observations of STS C.

We say that system C *refines* system A, denoted

$$C \sqsubseteq A,$$

if $Obs(C) \subseteq Obs(A)$. That is, if every observation of system C is also an observation of A.

4.1 Adaptation to SIGNAL Compilation

To adapt this general definition to the case at hand, there are several factors that need to be considered.

A first observation is that whatever the SIGNAL program can accomplish in a single step takes the corresponding C program several steps of execution. In fact, it takes a single full execution of the loop's body in the C program to perform a single abstract step. Consequently, we take for the STS representation of the compared C program a system obtained by *composing* the transition relations of the individual statements inside the loop's body. Since the body does not contain any nested loops, the computation of the overall transition relation is straightforward and can be achieved by successive substitutions. We refer to the resulting STS as the *composite* STS corresponding to the C program.

A second consideration is the choice of the abstract and concrete observation functions. For the abstract SIGNAL program, the natural observation is the snapshot of the values of the input and output variables. Thus, the abstract observation will be the tuple of values for the IO variables. For example, for the DEC SIGNAL program, the observation function is given by $\mathcal{O}^{\mathrm{A}} = (\mathrm{FB}, \mathrm{N})$.

A major feature of all the SIGNAL compilers we are treating is that all variables in the IOR set are preserved in the translation and are represented by identically named C variables. Therefore, the natural candidate for the concrete observation is the tuple of values of the IO C variables.

Unfortunately, there is a difference in the types of an IO SIGNAL variable and its corresponding C variable. While any IO signal $v \in IO$ may also assume the value $\perp$, signifying that the signal v is absent in the current step, all C variables are persistent and can never assume the value $\perp$. This implies that we have to identify for every concrete step and every IO-variable v whether the abstract version of v is present or absent in the current step.

Our decision was that an input variable v should be considered present in the current step iff a new value for v has been read during the current execution of the loop's body. Similarly, an output variable v is considered present iff the variable v was written during the current step.

To detect these events, we have instrumented the given C program by adding a boolean variable $rd.v$ for each input variable v, and a boolean variable $wr.v$ for each output variable[2]. All these auxiliary variables are set to 0 (*false*) at the beginning of the loop's body. After every `read`(v) operation, we add the assignment $rd.v := 1$, and after every `write`(v) operation, we add the assignment $wr.v := 1$.

In Fig. 1, we present the instrumented version of function DEC_iterate().

```
logical DEC_iterate()
{
       rd.FB_c = F;  wr.N_c = F;
l0:    h1_c = TRUE;
l1:    h2_c = ZN_c <= 1;
l2:    if (h2_c)
l2.1:    {read(FB_c); rd.FB_c = T;};
l3:    if (h2_c)
l3.1:    N_c = FB_c;
       else
l3.2:    N_c = ZN_c − 1;
l4:    {write(N_c); wr.N_c = T;};
l5:    ZN_c = N_c;
         return TRUE;
}
```

Fig. 1. Instrumented version of function DEC_iterate().

In Fig. 2, we present the composite STS corresponding to the instrumented C program. This presentation of the composite STS identifies the concrete observation function as $\mathcal{O}^{\mathrm{C}} = (\mathcal{O}^{\mathrm{C}}_{FB}, \mathcal{O}^{\mathrm{C}}_{N})$, where $\mathcal{O}^{\mathrm{C}}_{FB}$ and $\mathcal{O}^{\mathrm{C}}_{N}$ are defined in Fig. 2.

As can be seen in Fig. 2, the instrumented variables $rd.\mathrm{FB}_c$ and $wr.\mathrm{N}_c$ are defined within ρ_C and then used in the definition of the observation function $\mathcal{O}^{\mathrm{C}}$. Therefore, it is possible to simplify the composite STS by substituting the

[2] Our final implementation does not really add these auxiliary variables but performs an equivalent derivation. Introducing these variables simplifies the explanation.

$$
\begin{aligned}
V_C \; &: \{\mathrm{FB}_c,\ \mathrm{N}_c,\ \mathrm{ZN}_c,\ \mathrm{h1}_c,\ \mathrm{h2}_c,\ rd.\mathrm{FB}_c,\ wr.\mathrm{N}_c\} \\
\Theta_C \; &: \mathrm{ZN}_c = 1 \ \wedge \ pc = l_0 \\
\rho_C \; &: \left(\begin{array}{l}
\quad\ (\mathrm{h1}'_c = \mathrm{T}) \\
\wedge \ (\mathrm{h2}'_c = (\mathrm{ZN}_c \leq 1)) \\
\wedge \ (\mathrm{h2}'_c \Rightarrow (\mathrm{N}'_c = \mathrm{FB}'_c)) \\
\wedge \ (\neg \mathrm{h2}'_c \Rightarrow (\mathrm{FB}'_c = \mathrm{FB}_c \wedge \mathrm{N}'_c = \mathrm{ZN}_c - 1)) \\
\wedge \ (\mathrm{ZN}'_c = \mathrm{N}'_c) \\
\wedge \ (rd.\mathrm{FB}'_c = \mathrm{h2}'_c) \\
\wedge \ (wr.\mathrm{N}'_c = \mathrm{T})
\end{array}\right) \\
\mathcal{O}^{\mathrm{C}}_{FB} \; &: \mathbf{if}\ \boldsymbol{rd}.\mathrm{FB}_c\ \mathbf{then}\ \mathrm{FB}_c\ \mathbf{else}\ \perp \\
\mathcal{O}^{\mathrm{C}}_{N} \; &: \mathbf{if}\ \boldsymbol{wr}.\mathrm{N}_c\ \mathbf{then}\ \mathrm{N}_c\ \mathbf{else}\ \perp
\end{aligned}
$$

Fig. 2. Composite STS corresponding to the instrumented C program.

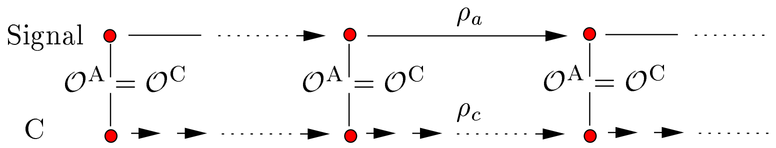
definition of the instrumented variables into $\mathcal{O}^{\mathrm{C}}$ and removing their definition from ρ_C. The actual implementation of the presented techniques within the code validation tool CVT never explicitly generate the instrumented version of the program but computes directly the simplified version as presented in Fig. 3.

$$
\begin{aligned}
V_C \; &: \{\mathrm{FB}_c,\ \mathrm{N}_c,\ \mathrm{ZN}_c,\ \mathrm{h1}_c,\ \mathrm{h2}_c\} \\
\Theta_C \; &: \mathrm{ZN}_c = 1 \ \wedge \ pc = l_0 \\
\rho_C \; &: \left(\begin{array}{l}
\quad\ (\mathrm{h1}'_c = \mathrm{T}) \\
\wedge \ (\mathrm{h2}'_c = (\mathrm{ZN}_c \leq 1)) \\
\wedge \ (\mathrm{h2}'_c \Rightarrow (\mathrm{N}'_c = \mathrm{FB}'_c)) \\
\wedge \ (\neg \mathrm{h2}'_c \Rightarrow (\mathrm{FB}'_c = \mathrm{FB}_c \wedge \mathrm{N}'_c = \mathrm{ZN}_c - 1)) \\
\wedge \ (\mathrm{ZN}'_c = \mathrm{N}'_c)
\end{array}\right) \\
\mathcal{O}^{\mathrm{C}}_{FB} \; &: \mathbf{if}\ \mathrm{h2}_c\ \mathbf{then}\ \mathrm{FB}_c\ \mathbf{else}\ \perp \\
\mathcal{O}^{\mathrm{C}}_{N} \; &: \mathrm{N}_c
\end{aligned}
$$

Fig. 3. The actual (simplified) $\mathrm{STS}_{\mathrm{DEC}}$ as generated by the CVT tool.

Before computing the composite STS corresponding to a given C program, the CVT tool performs some syntactic checks. For example, it checks that all references to an input variable $i \in I$ syntactically succeed the statement which reads the external input into i. Symmetrically, the tool checks that all assignments to an output variable $o \in O$ syntactically precede the statement which externally writes o. Failure in any of these checks will cause the tool to declare the translation as invalid.

In summary, we expect the abstract and concrete systems to be related as depicted below:



4.2 Proving Refinement by Simulation plus Abstraction Mapping

Let $A = (V_A, \Theta_A, \rho_A, \mathcal{O}^{\mathrm{A}})$ and $C = (V_C, \Theta_C, \rho_C, \mathcal{O}^{\mathrm{C}})$ be a given abstract and concrete systems. The standard way of proving that C refines A (cf. [1]) is based on the identification of an *abstraction mapping* $V_A = \alpha(V_C)$ mapping concrete states to abstract states, and establishing the premises of rule REF presented in Fig. 4. In many places, the mapping α is referred to as *refinement mapping*. Premise R1 of the rule ensures that the mapping α maps every initial concrete

For an abstraction mapping $V_A = \alpha(V_C)$,

R1.	$\Theta_C \ \wedge\ V_A = \alpha(V_C)$	$\rightarrow$	Θ_A	Initiation
R2.	$V_A = \alpha(V_C) \ \wedge\ \rho_C \ \wedge\ V'_A = \alpha(V'_C)$	$\rightarrow$	ρ_A	Propagation
R3.	$V_A = \alpha(V_C) \quad \rightarrow \quad \mathcal{O}^{\mathrm{A}} = \mathcal{O}^{\mathrm{C}}$			Compatibility with observations

$$\overline{\quad C \sqsubseteq A \quad}$$

Fig. 4. Rule REF.

state into an initial abstract state. Premise R2 requires that if the abstract state s_A and the concrete state s_C are α-related, and s'_C is a ρ_C-successor of s_C, then the abstract state $s'_A = \alpha(s'_C)$ is a ρ_A-successor of s_A. Together, R1 and R2 establish by induction that, for every concrete computation $\sigma_C : s^0_C, s^1_C, \ldots,$ there exists a corresponding abstract computation $\sigma_A : s^0_A, s^1_A, \ldots,$ such that $s^j_A = \alpha(s^j_C)$ for every $j = 0, 1, \ldots$. Applying premise R3, we obtain that the observations obtained from σ_A and σ_C are equal. This shows that rule REF is sound.

Rule REF , in the form presented in Fig. 4, is often not complete. In many cases we need to add an auxiliary invariant to the premises. However, in the case at hand, the relation $V_A = \alpha(V_C)$ is the only invariant we need.

4.3 Construction of the Mapping α

To complete the description of our verification methodology, it only remains to describe how we can construct automatically the abstraction mapping α.

The range of the mapping α is a tuple of values over the domains of the abstract variables V_A. In fact, for every abstract variable $v \in V_A$, the mapping α

contains a component $\alpha_v(V_{C})$ which defines the value of v in the abstract state α-related to the concrete state represented by V_{C}.

For the abstract observable variables $v \in IO$, premise R3 already constrains us to choose $\alpha_v(V_{C}) = \mathcal{O}_v^{\mathrm{C}}(V_{C})$. It therefore remains to describe the mappings α_v for $v \in V_{C} - IO = MRL$.

Recall that every variable $v \in R$ gives rise to an abstract register variable v, an abstract memorization variable $m.v$ introduced into the STS corresponding to A, and a corresponding concrete variable v_c. For example, the register flow ZN in the SIGNAL program DEC, gave rise to a similarly named variable and to the memorization variable m.ZN in the STS DEC and to the concrete variable ZN_c in function DEC_iterate().

Therefore, we define for each register flow r the following two instances of the α mapping:

$$\alpha_{m.r} \quad = \quad r_c \qquad\qquad \alpha'_{m.r} \quad = \quad r'_c.$$

For example, the mapping into the abstract variable m.ZN will be given by the equation $\mathrm{m.ZN} = \mathrm{ZN}_c$.

It only remains to define α_v for $v \in RL$. Since variables in L do not necessarily have counterparts in the C program, it may not be so easy to find an expression over V_{C} which will capture their values.

Here we are helped by the fact that every compilable SIGNAL program A is *determinate* in the new values of I (the inputs) and R (the register flows). Equivalently, the corresponding STS_{A} is determinate in the values of I' (new inputs) and M (old memorization values). Determinateness means that a set of values for these variables uniquely define the new values of all other variables (all variables in $IORML$). Determinateness is the necessary condition for being able to compile the program into a deterministic running program. Input SIGNAL programs which are not determinate are rejected by all the SIGNAL compilers we worked with.

We use the fact that every abstract variable $v \in RL$ has a unique equation of the form $v' = eq_v$ within STS_{A}. In principle, determinateness implies that we could chase these equations applying successive substitutions until we find for v a defining expression all in terms of the concrete variables V_{C}. However, this is not actually necessary. Instead, we transform premises R1–R3 into the following two verification conditions:

$$\textbf{W1.}\quad \Theta_{C} \;\wedge\; \bigwedge_{r\in R}(m.r = r_c) \;\wedge\; \bigwedge_{v\in IORL}(v = \perp) \qquad \rightarrow \quad \Theta_{A}$$

$$\textbf{W2.}\quad \bigwedge_{r\in R}\left(\begin{array}{c} m.r = r_c \\ \wedge\; m.r' = r'_c \end{array}\right) \;\wedge\; \rho_C \;\wedge\; \bigwedge_{v\in IO}(v' = (\mathcal{O}_v^{\mathrm{C}})') \;\wedge\; \bigwedge_{v\in RL}(v' = eq_v) \qquad \rightarrow \quad \rho_{A}$$

Obviously, we can dispense with premise R3 since $\mathcal{O}^{\mathrm{A}} = \mathcal{O}^{\mathrm{C}}$ is automatically guaranteed by taking $\alpha_v = \mathcal{O}_v^{\mathrm{C}}$ for each $v \in IO$. In verification condition W1, we simplified the mapping α having the prior information that only the $m.r \in M$

variables have non-bottom values and they are mapped by α into their concrete counterparts r_c.

In verification condition W2 we used the fact that the only unprimed variables to which $\rho_{_A}$ refers are the memorization variables $m.r$. Therefore, it is sufficient to map them in the unprimed version of $V_{_A} = \alpha(V_{_C})$. For the primed version of α, we used $\mathcal{O}_v^{\mathrm{C}}$ for all $v \in IO$, $m.r' = r'_{_c}$ for all $r \in R$ $(m.r \in M)$. and the original $\mathrm{STS}_{_A}$ equations for each $v \in RL$.

Theorem 1. *With the notation introduced above, if verification conditions W1 and W2 are valid, then $C \sqsubseteq A$.*

4.4 Illustrate on the Example

Applying these methods to the case of the SIGNAL program DEC and its translated C-program DEC_iterate(), we obtain the following two verification conditions:

$$\textbf{U1.}\ \mathrm{ZN}_c = 1 \ \wedge\ \mathrm{m.ZN} = \mathrm{ZN}_c \ \wedge \begin{pmatrix} & \mathrm{FB} = \bot \\ \wedge & \mathrm{N} = \bot \\ \wedge & \mathrm{ZN} = \bot \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} & \mathrm{FB} = \bot \\ \wedge & \mathrm{N} = \bot \\ \wedge & \mathrm{ZN} = \bot \\ \wedge & \mathrm{m.ZN} = 1 \end{pmatrix}$$

$$\textbf{U2.}\ \rho_{_C} \ \wedge \begin{pmatrix} & \mathrm{m.ZN} = \mathrm{ZN}_c \\ \wedge & \mathrm{FB}' = \textbf{if } \mathrm{h2}'_c \textbf{ then } \mathrm{FB}'_c \textbf{ else } \bot \\ \wedge & \mathrm{N}' = \mathrm{N}'_c \\ \wedge & \mathrm{m.ZN}' = \mathrm{ZN}'_c \\ \wedge & \mathrm{ZN}' = \textbf{if } \mathrm{N}' \neq \bot \textbf{ then } \mathrm{m.ZN} \textbf{ else } \bot \end{pmatrix} \quad \rightarrow \quad \rho_{_A}$$

Note that the α-mapping for the IO variables FB' and N', and for the M variable m.ZN is given directly in terms of the concrete variables, while the α-mapping of ZN' is given in terms of the abstract variables N' and m.ZN. In principle, it is possible to substitute the definitions of N' and m.ZN in the right-hand side of the definitions for ZN' and obtain a mapping which expresses all the relevant abstract variables in terms of the concrete variables. Performing the substitution and some simplifications concerning comparison to $\bot$, we obtain the following version of U2:

$$\textbf{U2a.} \qquad \rho_{_C} \ \wedge \begin{pmatrix} & \mathrm{m.ZN} & = & \mathrm{ZN}_c \\ \wedge & \mathrm{FB}' & = & \textbf{if } \mathrm{h2}'_c \textbf{ then } \mathrm{FB}'_c \textbf{ else } \bot \\ \wedge & \mathrm{N}' & = & \mathrm{N}'_c \\ \wedge & \mathrm{m.ZN}' & = & \mathrm{ZN}'_c \\ \wedge & \mathrm{ZN}' & = & \mathrm{ZN}_c \end{pmatrix} \quad \rightarrow \quad \rho_{_A}$$

The presented approach is immune against the optimizations performed by the industrial code generators that we considered. The proof technique exploits, in contrast to our previous work [20], only minimal knowledge about the code generation process. We only assume that IOM variables are reconstructible which is the minimal requirement for the C-code to be a correct implementation of the SIGNAL source [16].

5 Checking the Proof Obligations

As shown in the previous section, the generated proof obligations are quantifier-free implications referring to potentially infinite data domains such as the integers. Direct submission of these implications to a theorem prover such as PVS and invoking various proof procedures turned out to be far too slow.

In this section we explain the theoretical basis for an efficient BDD-based evaluation of the proof obligations on the basis of *uninterpreted functions.*

Typically, the verification conditions involve various arithmetical functions and predicates, tests for equality, boolean operations, and conditional (if-then-else) expressions. It has been our experience that the compiler performs very few arithmetical optimizations and leaves most of the arithmetical expressions intact. This suggests that most of the implications will hold independently of the special features of the operations and will be valid even if we replace the operations by *uninterpreted functions*.

5.1 The Uninterpreted Functions Encoding Scheme

Under the uninterpreted functions abstraction, we follow the encoding procedure of [2]. For every operation f occurring in a formula φ, which is not an equality test or a boolean operator, we perform the following:

- Replace each occurrence of a term $f(t_1,\ldots,t_k)$ in φ by a new variable v_f^i of a type equal to that of the value returned by f. Occurrences $f(t_1,\ldots,t_k)$ and $f(u_1,\ldots,u_k)$ are replaced by the same v_f^i iff t_j is identical to u_j for every $j = 1,\ldots,k$.
- Let $\hat{t}$ denote the result of replacing all outer-most occurrences of the form $f(t_1,\ldots,t_k)$ by the corresponding new variable v_f^i in a sub-term t of φ. For every pair of newly added variables v_f^i and v_f^j, $i \neq j$, corresponding to the non-identical occurrences $f(t_1,\ldots,t_k)$ and $f(u_1,\ldots,u_k)$, add the implication $(\hat{t_1} = \hat{u_1} \wedge \cdots \wedge \hat{t_k} = \hat{u_k}) \Rightarrow v_f^i = v_f^j$ as antecedent to the transformed formula.

Example 1. Following are ρ_c and ρ_a of program DEC, after performing the uninterpreted functions abstraction. Note how the '-' function and the '$\leq$' predicate were replaced by the new symbols v_-^i and $v_\leq^i$ respectively, where i is a running index:

$$\rho_c = \left(\begin{array}{ll} & (\mathrm{h1}'_c = \mathrm{T}) \\ \wedge & (\mathrm{h2}'_c = (v_\leq^1)) \\ \wedge & (\mathrm{h2}'_c \Rightarrow \mathrm{N}'_c = \mathrm{FB}'_c) \\ \wedge & (\neg \mathrm{h2}'_c \Rightarrow (\mathrm{FB}'_c = \mathrm{FB}_c \wedge \mathrm{N}'_c = v_-^1)) \\ \wedge & (\mathrm{ZN}'_c = \mathrm{N}'_c) \end{array}\right)$$

$$\rho_a = \left(\begin{array}{l} \quad \mathrm{N}' = \text{ if } \mathrm{FB}' \neq \perp \text{ then } \mathrm{FB}' \text{ else } v^2_- \\ \\ \wedge\ \mathrm{m.ZN}' = \text{ if } \mathrm{N}' \neq \perp \text{ then } \mathrm{N}' \text{ else } \mathrm{m.ZN} \\ \\ \wedge\ \mathrm{ZN}' = \text{ if } \mathrm{N}' \neq \perp \text{ then } \mathrm{m.ZN} \text{ else } \perp \\ \\ \wedge\ v^2_\leq \Leftrightarrow \mathrm{FB}' \neq \perp \end{array} \right)$$

After adding the functionality constraints, we obtain:

$$\varphi:\ (\widetilde{\alpha}\ \wedge\ (\mathrm{ZN}_c = \mathrm{ZN}'\ \wedge\ 1 = 1 \rightarrow (v^1_\leq = v^2_\leq\ \wedge\ v^1_- = v^2_-)) \rightarrow (\rho_c \rightarrow \rho_a))$$

where $\widetilde{\alpha}$ stands for the conjunction

$$\left(\begin{array}{lrcl} & \mathrm{m.ZN} & = & \mathrm{ZN}_c \\ \wedge & \mathrm{FB}' & = & \textbf{if } \mathrm{h2}'_c \textbf{ then } \mathrm{FB}'_c \textbf{ else } \perp \\ \wedge & \mathrm{N}' & = & \mathrm{N}'_c \\ \wedge & \mathrm{m.ZN}' & = & \mathrm{ZN}'_c \\ \wedge & \mathrm{ZN}' & = & \mathrm{ZN}_c \end{array} \right)$$

as presented in the verification condition U2a. We can use the substitution $\widetilde{\alpha}$ to replace all occurrences of abstract variables in φ by their corresponding concrete expressions. After some simplifications, this yields the following implication referring only to the concrete variables and the newly added v's:

$$\widetilde{\varphi}:\quad \left(\begin{array}{l} \quad v^1_\leq = v^2_\leq \\ \wedge\ v^1_- = v^2_- \end{array} \right)\ \wedge\ \left(\begin{array}{l} \quad (\mathrm{h1}'_c = \mathrm{T}) \\ \wedge\ (\mathrm{h2}'_c = (v^1_\leq)) \\ \wedge\ (\mathrm{h2}'_c \Rightarrow \mathrm{N}'_c = \mathrm{FB}'_c) \\ \wedge\ (\neg \mathrm{h2}'_c \Rightarrow (\mathrm{FB}'_c = \mathrm{FB}_c \wedge \mathrm{N}'_c = v^1_-)) \\ \wedge\ (\mathrm{ZN}'_c = \mathrm{N}'_c) \end{array} \right) \quad \rightarrow$$

$$\mathrm{N}'_c = \textbf{if } \mathrm{h2}'_c \textbf{ then } \mathrm{FB}'_c \textbf{ else } v^2_- \quad \wedge \quad \mathrm{ZN}'_c = \mathrm{N}'_c \quad \wedge \quad v^2_\leq \Leftrightarrow \mathrm{h2}'_c$$

Note that the third conjunct of ρ_A, the one related to ZN', has been simplified away. The reason is that this conjunct was used to define the α mapping for ZN', so it would be trivially satified after the substituition. ▟

The resulting equality formula belongs to a fragment of first order logic which has a *small model* property [4]. This means that the validity of these formulas can be established by solely inspecting models up to a certain finite cardinality. In order to make these finite domains as small as possible we apply another technique called *range allocation*.

The domain that can always be taken when using these kind of abstractions is simply a finite set of integers whose size is the number of (originally) integer/float variables (e.g. if there are n integer/float variables, then each of these variables ranges over $[1..n]$). It is not difficult to see that this range is sufficient for proving the invalidity of a formula if it was originally not valid. The invalidity of the formula implies that there is at least one assignment that makes the formula false. Any assignment that preserves the partitioning of the variables in this falsifying assignment will also falsify the formula (the absolute values are of no importance). This is why the $[1..n]$ range, which allows all possible partitions, is sufficient regardless of the formula's structure.

5.2 Range Allocation

The size of the state-space imposed by the $[1..n]$ range as suggested in the previous section is n^n. For most industrial-size programs this state-space is far too big to handle. But apparently there is a lot of redundancy in this range that can be avoided. The $[1..n]$ range is given without any analysis of the formula's structure. Note that our informal justification of the soundness of this method is independent of the structure of the formula we try to validate, and thus the range is sufficient for all formulas with n variables. This is probably the best we can do when the only information we have about the formula is that it has n variables. However, a more detailed analysis of the structure of the formula we wish to validate makes it possible to significantly decrease the ranges, and consequently the state space can be drastically reduced. This analysis is performed by the 'Range Allocation' module, using the *range allocation algorithm*, which significantly reduces the range of each of these (now enumerated type) variables, and enables the handling of larger programs. By applying the range-allocation technique, CVT decreases the state space of the verified formulas typically by orders of magnitude. We have many examples of formulas containing 150 integer variables or more (which result in a state-space of 150^{150} if the $[1..n]$ range is taken) which. after performing the range allocation algorithm, can be proved with a state-space of less than 100, in less than a second.

The range allocation algorithm is somewhat complex and its full description is beyond the scope of this paper. We refer the reader to [17] for more details, and describe here only the general idea.

The algorithm attempts to solve a satisfiability (validity) problem efficiently, by determining a *range allocation* $R : Vars(\varphi) \mapsto 2^{\mathbb{N}}$, mapping each integer variable $x_i \in \varphi$ into a small finite set of integers, such that φ is satisfiable (valid) iff it is satisfiable (respectively, valid) over some R-interpretation. After each variable x_i is encoded as an enumerated type over its finite domain $R(x_i)$, we use a standard BDD package, such as the one in TLV (see Section 5.5), to construct a BDD B_φ. Formula φ is satisfiable iff B_φ is not identical to 0.

Obviously, the success of our method depends on our ability to find range allocations with a small state-space.

In theory, there always exists a *singleton* range allocation R^*, satisfying the above requirements, such that R^* allocates each variable a domain consisting of a single integer, i.e., $|R^*| = 1$. This is supported by the following trivial argument:

If φ is satisfiable, then there exists an assignment $(x_1, \ldots, x_n) = (z_1, \ldots, z_n)$ satisfying φ. It is sufficient to take $R^* : x_1 \mapsto \{z_1\}, \ldots, x_n \mapsto \{z_n\}$ as the singleton allocation. If φ is unsatisfiable, it is sufficient to take $R^* : x_1, \ldots, x_n \mapsto \{0\}$.

However, finding the singleton allocation R^* amounts to a head-on attack on the primary NP-complete problem. Instead, we generalize the problem and attempt to find a small range allocation which is adequate for a *set* of formulas Φ which are "structurally similar" to the formula φ, and includes φ itself.

Consequently, we say that the range allocation R is *adequate* for the formula set Φ if, for every equality formula in the set $\varphi \in \Phi$, φ is satisfiable iff φ is satisfiable over R.

5.3 An Approach Based on the Set of Atomic Formulas

We assume that φ has no constants or boolean variables, and is given in a positive form, i.e. negations are only allowed within atomic formulas of the form $x_i \neq x_j$. Any equality formula can be brought into such positive form, by expressing all boolean operations such as $\rightarrow$, $\leq$ and the *if-then-else* construct in terms of the basic boolean operations $\neg$, $\vee$, and $\wedge$, and pushing all negations inside.

Let $At(\varphi)$ be the set of all atomic formulas of the form $x_i = x_j$ or $x_i \neq x_j$ appearing in φ, and let $\Phi(\varphi)$ be the family of all equality formulas which have the same set of atomic formulas as φ. Obviously $\varphi \in \Phi(\varphi)$. Note that the family defined by the atomic formula set $\{x_1 = x_2, x_1 \neq x_2\}$ includes both the satisfiable formula $x_1{=}x_2 \vee x_1{\neq}x_2$ and the unsatisfiable formula $x_1{=}x_2 \wedge x_1{\neq}x_2$.

For a set of atomic formulas A, we say that the subset $B = \{\psi_1, \ldots, \psi_k\} \subseteq A$ is *consistent* if the conjunction $\psi_1 \wedge \cdots \wedge \psi_k$ is satisfiable. Note that a set B is consistent iff it does not contain a chain of the form $x_1 = x_2,\ x_2 = x_3,\ \ldots,\ x_{r-1} = x_r$ together with the formula $x_1 \neq x_r$.

Given a set of atomic formulas A, a range allocation R is defined to be *satisfactory* for A if every consistent subset $B \subseteq A$ is R-satisfiable.

For example, the range allocation $R\colon x_1, x_2, x_3 \mapsto \{0\}$ is satisfactory for the atomic formula set $\{x_1 = x_2,\ x_2 = x_3\}$, while the allocation $R\colon x_1 \mapsto \{1\},\ x_2 \mapsto \{2\},\ x_3 \mapsto \{3\}$ is satisfactory for the formula set $\{x_1 \neq x_2,\ x_2 \neq x_3\}$. On the other hand, no singleton allocation is satisfactory for the set $\{x_1 = x_2,\ x_1 \neq x_2\}$. A minimal satisfactory allocation for this set can be given by $R\colon x_1 \mapsto \{1\},\ x_2 \mapsto \{1, 2\}$.

Claim. The range allocation R is satisfactory for the atomic formula set A iff R is adequate for $\Phi(A)$, the set of formulas φ such that $At(\varphi) = A$.

Thus, we concentrate our efforts on finding a small range allocation which is satisfactory for $A = At(\varphi)$ for a given equality formula φ. In view of the claim, we will continue to use the terms satisfactory and adequate synonymously.

We partition the set A into the two sets $A = A_= \cup A_{\neq}$, where $A_=$ contains all the equality formulas in A, while $A_{\neq}$ contains the inequalities.

Note that the sets $A_=(\varphi)$ and $A_{\neq}(\varphi)$ for a given formula φ can be computed without actually carrying out the transformation to positive form. All that is required is to check whether a given atomic formula has a positive or negative *polarity* within φ, where the polarity of a sub-formula p is determined according to whether the number of negations enclosing p is even (positive polarity) or odd (negative polarity). Additional considerations apply to sub-formulas involving the *if-then-else* construct.

Example 2. Let us illustrate these concepts on program DEC whose validity we wish to check.

Since our main algorithm checks for satisfiability, we proceed by calculating the polarity of each comparison in $\neg\widetilde{\varphi}$:

$$\begin{aligned} A_= &= \{(\mathrm{FB}'_c = \mathrm{FB}_c),\ (\mathrm{N}'_c = \mathrm{FB}'_c),\ (\mathrm{N}'_c = v^1_-),\ (\mathrm{ZN}'_c = \mathrm{N}'_c),\\ &\quad (v^1_- = v^2_-)\}\\ A_{\neq} &= \{(\mathrm{N}'_c \neq \mathrm{FB}'_c),\ (\mathrm{N}'_c \neq v^2_-),\ (\mathrm{ZN}'_c \neq \mathrm{N}'_c\} \end{aligned}$$

For example, the comparison $(\mathrm{N}'_c = v^1_-)$ in φ is contained within one negation (implied by appearing on the left hand side of the implication). Since we are considering $\neg\varphi$, this amounts to 2 negations, and since 2 is even, we add $(\mathrm{N}'_c = v^1_-)$ to $A_=$. ∎

This example would require a state-space in the order of 10^5 if we used the full $[1..n]$ range. The range allocation algorithm of [17] will find ranges adequate for this formula, with a state space of 32.

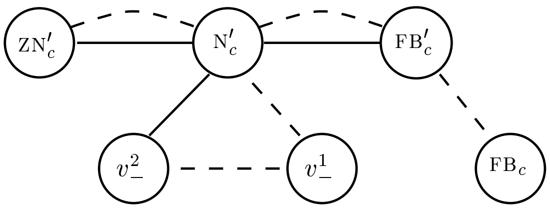


Fig. 5. The Graph $G : G_{\neq} \cup G_=$ representing $\neg\varphi$

5.4 A Graph-Theoretic Representation

The sets $A_{\neq}$ and $A_=$ can be represented by two graphs, $G_=$ and $G_{\neq}$ defined as follows:

(x_i, x_j) is an edge on $G_=$, the *equalities graph*, iff $(x_i = x_j) \in A_=$.

(x_i, x_j) is an edge on $G_{\neq}$, the *inequalities graph*, iff $(x_i \neq x_j) \in A_{\neq}$.

We refer to the joint graph as G.

An inconsistent subset $B \subseteq A$ will appear, graphically, as a cycle consisting of a single $G_{\neq}$-edge and any positive number of $G_=$-edges. We refer to these cycle as *contradictory cycles*.

In Fig. 5, we present the graph corresponding to the formula $\neg\varphi$, where $G_{=}$-edges are represented by dashed lines and $G_{\neq}$-edges are represented by solid lines.

The range allocation algorithm has several stages of traversing the graph, analyzing reachability, removing vertices etc. Without going into the details of the algorithm, we will present the ranges adequate for this graph, as computed by the algorithm: $R : \mathrm{ZN}'_c \mapsto \{1,2\}$, $\mathrm{N}'_c \mapsto \{1\}$, $\mathrm{FB}'_c \mapsto \{1,3\}$, $\mathrm{FB}_c \mapsto \{1,3\}$, $v^1_- \mapsto \{1,4\}$, and $v^2_- \mapsto \{1,4\}$.

Indeed, every consistent subset of the edges in the graph can be satisfied with these ranges. For example, for the subset made of the dashed edges $\{(\mathrm{FB}_c, \mathrm{FB}'_c), (\mathrm{FB}'_c, \mathrm{N}'_c), (v^2_-, v^1_-)$ and the solid edge $\{(v^2_-, \mathrm{N}'_c)\}$ we can assign $\mathrm{FB}_c = \mathrm{FB}'_c = \mathrm{N}'_c = 1$, and $v^1_- = v^2_- = 4$. Clearly this assignment satisfies the constraints that are represented by these edges.

In this case we reduced the state space from 6^6 to 32. In many cases the graphs are not as connected as this one, and therefore the reduction in the state space is much more significant. Often, 60 - 70% of the variables become constants, in the same way that N'_c became a constant in the example given above. We once more refer the reader to [17] for further details.

5.5 The Verifier module (TLV)

The validity of the verification conditions is checked by TLV [18], an SMV-based tool which provides the capability of BDD-programming and has been developed mainly for finite-state deductive proofs (and is thus convenient in our case for expressing the refinement rule). In the case that the equivalence proof fails, a counter example is displayed. Since it is possible to isolate the conjunct(s) that failed the proof, this information can be used by the compiler developer to check what went wrong. A proof log is generated as part of this process, indicating what was proved, at what level of abstraction and when.

6 A case study

We used CVT to validate an industrial size program, a code generated for the case study of a turbine developed by SNECMA, which is one of the industrial case studies in the SACRES project. The program was partitioned manually (by SNECMA) into 5 units which were separately compiled. Altogether the Signal specification is a few thousand lines long and contains more than 1000 variables. After the abstraction we had about 2000 variables (as explained in Section 5, the abstraction module replaces function symbols with new variables). Following is a summary of the results achieved by CVT:

Module	Conjuncts	Time (min.)
M1	530	1:54
M2	533	1:30
M3	124	0:27
M4	308	2:22
M5	860	5:55
Total :	2355	12:08

Although it is hard to assess at this stage how strongly this particular example indicates the feasibility of the Translation Validation approach, the case study undoubtedly shows that a compilation process (of the type we considered) of an industrial size program can be automatically verified in a reasonable amount of time.

7 Conclusions

We have presented the theory which underlies our translation validation approach for optimizing industrial compilers from SIGNAL to C. We described the translation of SIGNAL and C programs to STS, the generation of the substitution α and the final assembling of the proof obligations according to Rule REF as they are implemented in CVT (the Code Validation Tool). In addition, the decision procedure, including the abstraction, the Range Allocation algorithm and various other optimizations that were not presented in this paper, are also implemented in CVT. We believe that the case study that was presented in the paper is a strong indication that translation validation is a viable alternative to full compiler verification.

Dedication This paper is dedicated with friendship and appreciation to Hans Langmaac, a pioneer in the area of compiler verification, whose work proved the great value and feasibility of a seamless and fully verified development path from specification to implementation.

Acknowledgment We wish to express our gratitude to Markus Müller-Olm who reviewed an earlier version of this article in record-time and yet managed to provide us with several significant and insightful comments which led to a meaningful improvement in the quality and validity of this paper.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] W. Ackerman. *Solvable cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[3] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL languages and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1996.

[5] B. Buth, K.-H. Buth, M. Fränzle, B. von Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction'92, 4th International Conference Paderborn, Germany*, volume 641 of *Lect. Notes in Comp. Sci.*, pages 141–155. Springer-Verlag, 1992.

[6] K.M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.

[7] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In O. Grumberg, editor, *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lect. Notes in Comp. Sci.*, pages 202–213. Springer-Verlag, 1997.

[8] A. Cimatti, F. Giunchiglia, P. Traverso, and A. Villafiorita. Run-time result formal verification of safety critical software: an industrial case study. In *Run-Time Result Verification*. The 1999 Federated Logic Conference, 1999.

[9] D.L. Clutterbuck and B.A. Carre. The verification of low-level code. *Software Engineering Journal*, pages 97–111, 1998.

[10] P. Curzon. A verified compiler for a structured assembly language. In *international workshop on the HOL theorem Proving System and its applications*. IEEE Computer Society Press, 1991.

[11] J.D. Guttman, J.D. Ramsdell, and V. Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8:33–100, 1995.

[12] J.D. Guttman, J.D. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[13] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1997.

[14] D.P. Oliva, J.D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8:111–182, 1995.

[15] I.M. O'Neill, D.L. Clutterbuck, and P.F. Farrow. The formal verification of safety-critical assembly code. In *IFAC Symposium on safety of computer control systems*, 1988.

[16] Private communications with TNI (BREST), Siemens (Munich) and Inria (Rennes).

[17] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In N. Halbwachs and D. Peled, editors, *Proc. 11st Intl. Conference on Computer Aided Verification (CAV'99)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1999. to appear.

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

[19] 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.

[20] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *4th Intl. Conf. TACAS'98*, volume 1384 of *Lect. Notes in Comp. Sci.*, pages 151–166. Springer-Verlag, 1998.
[21] A. Stump and D. Dill. Generating proofs from a decision procedure. In *Run-Time Result Verification.* The 1999 Federated Logic Conference, 1999.
[22] P. Traverso and P. Bertoli. Mechanized result verification: an industrial application. In *Run-Time Result Verification.* The 1999 Federated Logic Conference, 1999.