# Translation Validation for Synchronous Languages *

A. Pnueli O. Shtrichman M. Siegel
Weizmann Institute of Science, Rehovot, Israel

January 12, 1998

**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 demonstrate the practicability of translation validation for highly optimizing, industrial code generators from DC+, a widely used intermediate format for synchronous languages, to C. We introduce new abstraction techniques as part of the automation of our approach.

## 1 Introduction

Compiler verification is an extremely complex task and every change to the compiler (even minor revisions) requires redoing the proof. Thus, compiler verification tends to "freeze" the compiler design and discourages any future improvements and revisions which is not acceptable in an industrial setting. This drawback can be avoided by a well designed translation validation approach, first introduced in [10], which compares the input and the output of the compiler for each individual run independently of *how* the output is generated from the input.

In this paper we consider translation validation for synchronous languages. Synchronous languages [8], such as Esterel [3], Argos [9], Signal [2] and Lustre [4], are mainly used in industrial applications for the development of safety critical, reactive systems. In particular, they are 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 (some details are given in Section 2.2) in order to derive highly efficient implementations in languages such as C and ADA. In order to share code generation tools (and silicon compilers, simulators, verification tools etc.) for synchronous languages, the DC+ format has been developed. DC+ [7] is an equational representation for both imperative and declarative synchronous languages.

In this paper we explain the theory underlying (fully automatic) translation validation for two industrial compilers from DC+ to C. These compilers – which apply more than 100 optimization rules during code generation [11] – are developed in the ESPRIT project SACRES by the French company TNI and by Inria (Rennes) and are used by Siemens, SNECMA and British Aerospace.

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

While developed in the context of code generators for synchronous languages, the proposed method has a considerably 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. Any source language with these features can benefit from the method proposed in this paper. For example, the language UNITY [5] which comes from the world of asynchronous distributed systems is a possible customer for the proposed method.

We present a common semantical model for DC+ 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 TVT (Translation Validation Tool). The tool is currently tested on industrial case studies. 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 compiler are based on the same definition of "correct code generation". This is the case for the TNI and the Inria compiler and, indeed, TVT is used to validate code originating from either of these two compilers.

**Related work**: In [10] we addressed code validation for a compiler from Signal to C which did not perform optimizations while generating the C code. The revision of this work to deal with the optimizing compilers from TNI and Inria is the topic of this paper. Neither the TNI nor the Inria compiler are verified. They are constantly revised/improved which makes a direct verification of the compiler itself extremely expensive or postpones it indefinitely. The work in [6] performs translation validation on a purely syntactical 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 purely *semantical* approach which is far more involved than finding a syntactical correspondence between programs. We are not aware of any other translation validation approaches for industrial compilers.

The paper is organized as follows. In Section 2 we give a brief introduction to DC+. Section 3 presents the concepts which underly the generation of the proof obligations. In Section 4 we present the decision procedures to check the validity of these proof obligations. Section 5 contains some conclusions. Because of space limitations proofs and some constructions are omitted.

## 2 The DC+ Format

A DC+ 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 DC+ program are *flows*, which represent a sequence 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 (standardly [2]) represented by the bottom symbol $\perp$. Clocks are boolean 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 DC+ program with its environment, also *internal* flows are used and manipulated by the program.

### 2.1 DC+ and its Semantics

In order to present the formal semantics of DC+ we introduce a variant of *synchronous transition systems* (STS) [10]. 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 variables in $V$. $\Sigma_V$ denotes 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 $\Theta$ characterizing the initial states of system $A$, and a transition relation $\rho$. 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 to as $\Sigma_A$ instead of $\Sigma_{V_A}$. 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 are intended to represent flows of DC+ programs, thus their domains contain the designated element $\perp$ to indicate absence of the respective flow.

A *computation* of an STS $A = (V, \Theta, \rho)$ 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 $A$.

For the purpose of translation validation, DC+ programs are translated into the STS formalism. A brief introduction to DC+ and some details of its translation to STS are given next.

A DC+ program consists of a set of constraints which determine the transition relation of the system (there are no composition operators). At each instant of time all constraints have to be satisfied by the values that the flows have at this instant. The constraints are expressed as *equation* and *memorization* statements. The equation $v = exp$ [1] defines the flow $v$ to be equal to the expression *exp* at any instant, which implies that also their clocks coincide. Formally this equation contributes the following clause to the transition relation of the STS which represents the DC+ source:

$$v' = \text{ if } \ clk(exp') \text{ then } \ exp' \text{ else } \ \perp$$

The second kind of constraints is a memorization statement $r \stackrel{m}{=} exp$ which defines $r$ to hold the last (*not* including the present) non-bottom value of *exp*. Also memorizations imply that the arguments have the same clocks. Whereas equations are used to specify *instantaneous reactions* of the system, memorizations are used to define the *internal state* of the system, i.e. its registers when the DC+ program is considered as an operator network [7]. The formal semantics of memorizations is

$$\begin{array}{lrcl} \wedge & x.r' & = & \text{if } \ clk(exp') \text{ then } \ exp' \text{ else } \ x.r \\ \wedge & r' & = & \text{if } \ clk(exp') \text{ then } \ x.r \text{ else } \ \perp \end{array}$$

This definition introduces an auxiliary variable $x.r$ which stores the last non-bottom value of *exp* when *cond* was true. Variable $x.r$ is initialized in $\Theta$ of the STS representing the DC+ source to define the first non-bottom value of $r$ (an init-construct in DC+ defines such initial values). From now on we refer to flows defined by memorizations as *register flows*. Variables in an STS which represent register flows will typically be denoted by $r$, corresponding memorization variables by $x.r$.

**Example 1** *The figure below shows a possible scenario for the memorization statement* $r \stackrel{m}{=} v$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $v$ | $\perp$ | 2 | $\perp$ | 3 | 5 | $\perp$ | 8 |
| $x.r$ | 0 | 2 | 2 | 3 | 5 | 5 | 8 |
| $r$ | $\perp$ | 0 | $\perp$ | 2 | 3 | $\perp$ | 5 |
| $clk(r), clk(v)$ | $\perp$ | T | $\perp$ | T | T | $\perp$ | T |

Time

time instants

*Note, that variable $x.r$ is needed since the last non-bottom value of $v$ is not accessible because $v$ is volatile. Variable $x.r$ itself is persistent.*

[1] we omit an optional activation condition to simplify the presentation

There are two kinds of functions which can be used in DC+expressions: *monochronous* functions, such as $+, -, div, \dots$ , are standard operators on flows whose results share the same clock as their arguments while *polychronous* functions, such as $\mathsf{when}(w, cond)$ and $\mathsf{pcond}(cond, exp_1, exp_2)$, introduce and handle flows with different clocks. The latter operators can be used for under/oversampling of flows. They are translated as follows:

$$\mathsf{when}(exp, cond) \stackrel{\text{def}}{=} \text{if} \quad cond = \mathrm{T} \text{ then } \quad exp \text{ else } \perp$$

$$\mathsf{pcond}(cond, exp_1, exp_2) \stackrel{\text{def}}{=} \left\{ \begin{array}{lll} \text{if} & cond = \mathrm{T} \wedge clk(exp_1) \text{ then} & exp_1 \\ \text{else if} & cond = \mathrm{F} \wedge clk(exp_2) \text{ then} & exp_2 \\ \text{else } \perp & & \end{array} \right\}$$

Based on these definitions we can define the semantics of a DC+program $D$ by an STS $S = (V, \Theta, \rho)$ as follows. Set $V$ is identical to the set of flows in $D$ plus the memorization variables $x.r$ which are introduced by the semantics above. Assertion $\Theta$ defines all variables to be initially absent [7] except memorization variables which are initialized as stated in the DC+source. Finally, $\rho$ is obtained as the conjunction of the predicates which define the semantics of equation and memorization statements.

In the following sections we assume that the type definitions for variables also specify the "DC+ 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.

### 2.2 Compilation of Multi-clocked Synchronous Languages

The compilation scheme for multi-locked synchronous languages (e.g. Signal, DC+) to imperative, sequential languages (e.g. C, ADA) looks as follows. The set of equation and memorization statements of a program $D$ form a *linear equation system* $LES$ on the flows of $D$ and their associated clocks. Solutions of $LES$ for a given set of input/register values determine the next state of the system. The compiler derives from $D$ an imperative program $C$ which consists of one main loop whose task is to repeatedly compute such solutions of the $LES$.

For these solutions to be efficiently computable without expensive fix-point iteration, only programs are compiled which induce an *acyclic dependency relation* on flows. This dependency relation is not static but rather depends on which flows are currently present. Thus, the compiler computes from the source program another linear equation system $CC$ – the, so called, *clock calculus* [2] – which records the dependencies amongst clocks and a *conditional* dependency graph $CDG$ on flows.

If $CC$ has a unique solution for all possible inputs and $CDG$ is acyclic, then system $LES$ is uniquely solvable for all possible input/register values, i.e. the program $D$ is deadlock free and determinate in its input/register variables A formal definition of determinacy is given in Section 3.1). The program is rejected by the compiler if one of the conditions is not satisfied. Otherwise the produced code contains statements originating from the clock calculus and assignments to variables (representing the flows of $D$) whose order must be consistent with the dependency graph $CDG$. 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.

**Example 2** *A part of the (non-optimized) translation of a* DC+ *program into C code is given below:*

```
.....                                WHILE true DO
r_in =m= in                          { ..... ;
out = when(2*r_in,in>10)             read(in);
.....                                c_out = (in>10)
                                     IF c_out { ........... ;
```

```
                    out = 2*r_in;
                    write(out)
                  }
          r_in = in;
          ........
        }
```

For the translation validation process also the C programs are translated into the STS formalism. Since the generated C code uses in the body of the main loop only a small fragment of ANSI C (e.g. no pointers, no loops), the translation is straightforward. Note, however that the C programs use *persistent* variables (i.e. variables which are never absent) to implement DC+ programs which use volatile variables. This has to be taken into account when defining the notion of "correct implementation" in the next section.

# 3 Correct Implementation: Refinement

Our approach to establish that "the C-code correctly implements the DC+ source" is based on the notion of *refinement*. The presented concepts have been approved by TNI and Inria.

## 3.1 Refinement and Refinement Mappings

Consider the two STSs $A = (V_{\mathrm{A}}, \Theta_{\mathrm{A}}, \rho_{\mathrm{A}})$ and $C = (V_{\mathrm{C}}, \Theta_{\mathrm{C}}, \rho_{\mathrm{C}})$, with $IO_A = IO_C$, to which we refer as the *abstract* and *concrete* system, respectively. We say that $C$ *refines* $A$, denoted by $C$ **ref** $A$, if for all $\sigma = \langle s_0, s_1, s_2, \ldots\rangle$ in $\|C\|$ there exists a $\tau = \langle t_0, t_1, t_2, \ldots\rangle$ in $\|A\|$ such that

$$\forall x \in IO_A . \forall i \in \mathbb{N}.\ s_i[x] = t_i[x] \text{ or } \ t_i[x] = \perp.$$

In order to establish this notion of refinement for two given systems we have to *construct* for each concrete computation $\sigma \in \|C\|$ the corresponding abstract computation $\tau \in \|A\|$ such that the above property is satisfied. Such constructions are usually done by means of *refinement mappings* [1]. Rather than the standard static correspondence between concrete and abstract variables, we need a more general mechanism which relates *persistent* variables of the STS-representation of the C-code (denoted C-STS from now on) to *volatile* variables of the STS-representation of the DC+program (DC+STS).

**Definition 3** *Given systems $A = (V_A, \Theta_A, \rho_A)$ and $C = (V_C, \Theta_C, \rho_C)$ with $IO_A = IO_C$. A mapping $f : \Sigma_C \to \Sigma_A$ is a* clocked refinement mapping *from $C$ to $A$ if it satisfies the requirements of*

- *Initiation: $s \models \Theta_C$ implies $f(s) \models \Theta_A$, for all $s \in \Sigma_C$.*
- *Propagation: $(s, s') \models \rho_C$ implies $(f(s), f(s')) \models \rho_A$, for all $s, s' \in \Sigma_C$.*
- *Preservation of Observation: $\forall x \in IO_A . \forall s \in \Sigma_C .\ f(s)[x] = s[x]$ or $f(s)[x] = \perp$.*

The idea of this definition is, that in each time instant $i$ and for each observable variable $x \in IO_A = IO_C$ either $x$ is present in the abstract system and $f(s)[x]$ coincides with $s[x]$ or $x$ is absent in $f(s)$. In the following presentation we omit the qualifier "clocked" if it is clear from the context.

**Theorem 4** *If there exists a clocked refinement mapping from $C$ to $A$ then $C$* **ref** *$A$.*

Usually, finding such a mapping $f$ is left to the ingenuity of the verifier. In the context of code validation it is essential that $f$ can be *automatically constructed* from the source and target programs.

In order to facilitate the generation of suitable refinement mappings, we perform a transformation of the transition relation of C-STS. Originally, the transition relation in C-STSupdates variables *successively* till finally the result of the *simultaneous* update of variables in DC+STS has been computed (these steps correspond to one iteration of the main loop of the C-program, cf. Example 2). We construct a new transition relation which reflects the *accumulated* effect of the individual steps in the execution of the loop's body. This compression of the transition relation $\rho$ of C-STS is done by a standard substitution algorithm (note, that the body of the main loop is itself loop-free). The size of the resulting transition relation is small than the size of the original transition relation. From now on we consider the generation of refinement mappings for DC+STS and C-STS where C-STS uses the compressed transition relation.

The main idea in [10] was to generate refinement mappings which reconstruct the values of *all* abstract variables. In order to do so, it was necessary to extract from the structure of the C-code the information whether an abstract variable is currently present/absent, i.e. we reconstructed the clocks of these variables. With this information about clocks we could define the correct values of abstract variables from the values of their concrete counterparts. Such a reconstruction of all abstract variables is not possible in the case of the optimizing code generators, because:

1. Internal variables are possibly eliminated for space efficiency during compilation; so there are no corresponding variables in the C-code from which we could automatically reconstruct their values.

2. The reconstruction of the clocks of abstract variables was based in [10] on the assumption that an abstract variable is present iff the corresponding concrete variable has been updated in the current iteration (cf. Example 2). The optimizing compilers *move assignments* between if-blocks in the C-code such that neither the fact that a concrete variable has been written implies that its abstract counterpart is actually present nor does the presence of an abstract variable implies that its concrete counterpart is written in the current iteration.

Since the code generators cannot eliminate $IOR$ variables without producing incorrect code, we can exploit the property of *determinacy* – which is a central property in synchronous programs, see Section 2 – to *implicitly* reconstruct local abstract variables.

**Definition 5** *An* STS $S = (V, \Theta, \rho)$ *is* determinate in $V \subseteq V_S$ *if the following holds:*

$$\forall s_1, s_2, s_3 \in \Sigma_S.\ ((s_1, s_2) \models \rho \wedge (s_1, s_3) \models \rho \wedge s_2[V] = s_3[V]) \Rightarrow s_2[V_S \setminus V] = s_3[V_S \setminus V]$$

Determinacy of $S$ in $V$ says that, after a transition, the value of variables in set $V_S \setminus V$ are uniquely determined once the values for the variables in $V$ have been fixed. DC+ programs generated from Esterel, Lustre etc. are by construction determinate in their $IRM$ variables. Thus, from now on we always assume that the considered STS representations of DC+programs are determinate in their set of $IRM$ variables.

In order to determine corresponding abstract states it thus suffices to reconstruct these $IRM$ variables by the refinement mapping. Besides this we have to reconstruct the values of abstract output variables to check whether the generated abstract and concrete outputs indeed coincide. Consequently we now concentrate on a refinement approach which allows to reconstruct the abstract IORM variables. Fortunately, the clock generation scheme as presented in [10] – and briefly recalled in Section 3.2 – can still be applied for IOR variables [11]. The reason is that a "re-timing" of IOR variables (i.e. moving assignments to other if-blocks as explained in point 2) tends to an incorrect implementation since either inputs/register variables would be provided at wrong time instants or/and outputs would be generated at instants where they are not supposed to occur. For memorization variables no clock has to be constructed since they always carry non-bottom values by definition.

Technically we eliminate all local variables in DC+STS $= (V, \Theta, \rho)$ by removing them from $V$, removing their initializations from $\Theta$ and hiding them from $\rho$ by existential quantification. The result of applying this transformation to some STS $A$ is denoted by $A^{\exists}$. Determinacy of $A$ in $IRM_A$ implies that it suffices to construct an inductive refinement mapping from $C$ to $A^{\exists}$ to actually prove that $C$ correctly implements $A$, i.e. $C$ $\mathsf{ref}$ $A$.

However, there is one remaining problem with the reconstruction of the values of register variables. Registers are updated during one iteration of the main loop *after* they have been used in assignments of other variables, cf. Example 2. Thus, at the end of an iteration, register variables are already updated for the *next* iteration. So, the values of abstract register variables have to be reconstructed from the values of the corresponding variables at the *beginning* of the iteration while input/output/memorization variables can be reconstructed from the values of corresponding variables at the *end* of the iteration. This situation is handled by automatically inserting a *history variable* $h.r$ [1] into the C-code for each register variable $r$. This is done by inserting the assignment $h.r = r$ directly before each assignment to $r$ in the C code. These history variables recall the previous value of register variables.

The constructions explained above (introduction of history variables for register variables, compression of $\rho_c$, hiding of local variables in $\rho_a$) allow us to use refinement mappings as introduced in Definition 3 to establish the correctness of the generated C-code w.r.t. the DC+ source program.

### 3.2 Syntactic Representation and Proof Rule for Refinement

In the quest for automating the code validation process, we present in this section a *syntactical representation* of clocked refinement mappings and the associated proof rule. In this, we follow the ideas in [10]. Then, we describe how the components used in the proof rule can be computed, so that the translation validation process can be carried out fully automatically.

Consider two STSs $A$ and $C$ with $IO_A = IO_C$. Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be a *substitution* that replaces each abstract variable $v \in V_A$ by an expression $\mathcal{E}_v$ over the concrete variables $V_C$. Such a substitution $\alpha$ induces a mapping between states, denoted by $f_\alpha$. Let $s_C$ be some concrete state in $\Sigma_C$. The abstract state $s_A \stackrel{\text{def}}{=} f_\alpha(s_C)$ corresponding to $s_C$ under substitution $\alpha$ assigns to each variable $v \in V_A$ the value of expression $\mathcal{E}_v$ evaluated in $s_C$. In this way, a refinement mapping can be syntactically defined by means of an appropriate substitution $\alpha$. Such a substitution $\alpha$ is defined to be *observation preserving* if $\forall v \in IO_A.\ \models (v)[\alpha] = v \vee (v)[\alpha] = \perp$, cf. Definition 3.

Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be an observation preserving substitution

| | | |
|---|---|---|
| **R1.** | $\Theta_C \ \Rightarrow\ \Theta_A[\alpha]$ | Initiation |
| **R2.** | $\rho_C \ \Rightarrow\ \rho_A[\alpha]$ | Propagation |

$$\frac{\textbf{R1. } \Theta_C \Rightarrow \Theta_A[\alpha] \qquad \textbf{R2. } \rho_C \Rightarrow \rho_A[\alpha]}{C \ \mathsf{ref}\ A}$$

Rule $\mathsf{ref}$: Proving Refinement

The proof rule can be easily adapted to incorporate an invariant of the concrete system. However, after compressing the concrete transition relation no invariant is needed since the compressed concrete transition relation and the abstract transition relation also coincide (if the C-code is indeed correct) on non-reachable states. The reason for this is, that code generators can not exploit any reachability information for optimizations. This is one of the key observations which facilitate automatic refinement proofs.

In order for rule $\mathsf{ref}$ to be useful in a fully automatic translation validation process, an appropriate substitution $\alpha$ has to be generated automatically.

Based on the previous explanations we can define the following generic substitutions $\alpha$.

**Definition 6** *Given $A \in$ STS, representing the* DC+*program where local variables have been eliminated, and $C \in$ STS, representing the compressed C-code. We define $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ by:*

$$
\begin{array}{llll}
\alpha(v) = & \textit{if} \ \ \mathrm{clk}_c(v) \ \ \textit{then} \ \ v \ \ \textit{else} \ \ \perp & & \textit{for all } v \in IO_A (= IO_C) \\
\alpha(r) = & \textit{if} \ \ \mathrm{clk}_c(r) \ \ \textit{then} \ \ h.r \ \ \textit{else} \ \ \perp & & \textit{for all } r \in R_A (= R_C) \\
\alpha(x.r) = r & & & \textit{for all } x.r \in M_A
\end{array}
$$

This specific definition of $\alpha$ automatically yields observation preserving substitutions.

The algorithm for computing the clock expressions above works as follows. The construction is based on viewing the body of the main loop of the C-code as a (cyclic) directed graph where every edge $e$ is labeled by either a *guard* $\gamma(e)$ (originating from if-conditions) or an action which can be a *read* of an input variable, a *write* of an output variable, or an assignment to a variable (cf. Example 2.

The clock expression $clk_c(v)$ is computed by considering the guards along paths leading to assignments to $v$ (resp. read/write statements in case of input/output variables). Let $path(v)$ be the conjunction of all guards along a fixed path leading to an assignment of $v$ and let $written(v)$ be the disjunction of the predicates $path(v)$ for all possible paths leading to an assignment of $v$. Then $clk_c(v)$ is obtained from $written(v)$ by replacing all register variables $r$ by $h.r$ (recall that the register variables are already updated for the next iteration). The combination of the techniques and constructions mentioned above now allow us to automatically extract two first order logic formulas (corresponding to **R1.** and **R.** in rule ref) which state the correctness of the generated code if these formulas can be shown to be valid.

We would like to emphasize that the presented approach is immune against the optimizations performed by the industrial code generators that we consider. The proof technique exploits, in contrast to our previous work [10], only minimal knowledge about the code generation process. We only assume that $IORM$ variables are reconstructible which is the minimal requirement for the C-code to be a correct implementation of the DC+ source [11].

# 4 Checking the Proof Obligations

The generated proof obligations are infinite state assertions, thus just invoking a BDD-based decision procedure obviously does not work. Directly supplying them to a theorem prover such as PVS and starting proof strategies turned out to be far too slow.

In this section we explain the theoretical basis for an efficient BDD-based validation of the proof obligations on the basis of *uninterpreted functions*. The introduced concepts are expected to have a wider applicability as a general decision procedure for fragments of first order logic.

## 4.1 Preliminaries for the abstractions

All the generated proof obligations are of the form $\varphi_C \Rightarrow \exists y_1, \ldots, y_n.\, (\varphi_A \wedge \bigwedge_{i=1}^{i=n} y_i = exp_i)$ where $\varphi_C$ is the left hand side of the implications in Rule ref. The right hand side consists of the abstract local variables which are hidden by existential quantification and a conjunct $\varphi_A$ which deals with the other variables. (We assume that the substitution $\alpha$ in Rule ref has already been performed.) In case of a determinate DC+ program, the set of equalities $y_1 = exp_1, \ldots, y_n = exp_n$ uniquely determine the values of $y_1, \ldots, y_n$ in terms of the other abstract variables. Thus we can use the following transformation in order to remove the existential quantifications from the proof obligations.

$$
\begin{array}{ll}
\varphi_C \Rightarrow \exists y.\, (y = exp \wedge \varphi_A) & \\
\varphi_C \Rightarrow \forall y.\, (y = exp \Rightarrow \varphi_A) & \text{existance and uniqueness of } y \\
\forall y.\, (\varphi_C \Rightarrow (y = exp \Rightarrow \varphi_A) & \text{since } y \text{ does not occur free in } \varphi_C \\
\forall y.\, (\varphi_C \wedge y = exp) \Rightarrow \varphi_A & \text{propositional logic}
\end{array}
$$

This last formula is validity equivalent to the quantifier-free implication $(\varphi_C \wedge y = exp) \Rightarrow \varphi_A$. So, from now on we can concentrate on quantifier-free formulas with free variables. In order to simplify the presentation we consider formulas $\varphi$ with variables of type boolean and integer and functions over these domains. Predicates are treated as boolean valued functions.

In the rest of this section we use a validity relation which is parameterized by a *declaration* $D$ and an *interpretation* $I$, denoted by $\models_I^D \varphi$. Here, the declaration $D$ determines the type of the variables in $\varphi$ and $I$ interprets a subset of the function symbols occuring in $\varphi$. We say that $\varphi$ is *valid w.r.t.* $(I, D)$, denoted by $\models_I^D \varphi$, if $\varphi$ is valid in every model $M$ where the function symbols are interpreted according to $I$ and the variables according to $D$. Note, that $M$ may interpret in an arbitrary way all the function symbols whose interpretation is not fixed by $I$.

For interpretations $I_1, I_2$ we define $I_1 \preceq I_2$ if $I_1$ and $I_2$ coincide on those function symbols interpreted by $I_1$, but $I_2$ possibly interprets more function symbols. Obviously, we have for $I_1 \preceq I_2$ that $\models_{I_1}^D \varphi$ implies $\models_{I_2}^D \varphi$.

The idea of the forthcoming abstractions is as follows. We have to check the validity of formula $\varphi$ (the proof obligation) w.r.t. a declaration $D$ which assigns integer/boolean types to variables and an interpretation $J$ which gives (a standard) interpretation to all function symbols in $\varphi$.

As a sufficient condition for $\models_J^D \varphi$ we check $\models_I^D \varphi$ where $I \preceq J$ only interprets a subset of the function symbols in $\varphi$. Moving from interpretation $J$ to $I$ means relaxing the constraints on the interpretation of some function symbols and treating them logically as *uninterpreted*.

In a second step we apply the new technique of *function encoding*, described below, in order to substitute uninterpreted functions by fresh variables. The encoded formula, where all uninterpreted functions have been removed, is denoted by $\mathsf{F\text{-}enc}(\varphi)$ and is logically equivalent to $\varphi$ under interpretation $I$, i.e. $\models_I^D \varphi \iff \models_I^D \mathsf{F\text{-}enc}(\varphi)$.

The encoded formulas belong to a fragment of first order logic which has a *small model property*. 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 encoding which replaces constants in $\mathsf{F\text{-}enc}(\varphi)$ by smaller constants such that the encoded formula $\mathsf{CF\text{-}enc}(\varphi)$ is logically equivalent to $\mathsf{F\text{-}enc}(\varphi)$, i.e. $\models_I^D \mathsf{F\text{-}enc}(\varphi) \iff \models_I^D \mathsf{CF\text{-}enc}(\varphi)$.

The final step of the abstraction is to determine the finite domains over which the variables of $\mathsf{CF\text{-}enc}(\varphi)$ need to be interpreted in order to faithfully check the validity of $\models_I^D \mathsf{CF\text{-}enc}(\varphi)$, i.e. to determine a new declaration $D^*$ where all variables range over finite domains such that $\models_I^D \mathsf{CF\text{-}enc}(\varphi) \iff \models_I^{D^*} \mathsf{CF\text{-}enc}(\varphi)$ holds.

We thus get the following overall picture of our abstractions:

$$\models_J^D \varphi \quad \text{if} \quad \models_I^D \varphi \quad \text{iff} \quad \models_I^D \mathsf{F\text{-}enc}(\varphi) \quad \text{iff} \quad \models_I^D \mathsf{CF\text{-}enc}(\varphi) \quad \text{iff} \quad \models_I^{D^*} \mathsf{CF\text{-}enc}(\varphi).$$

Next we explain the function encoding which is common to all our abstractions, then we illustrate the constant encoding mechanisms and the declaration changes for the individual abstractions.

## 4.2 The function encoding scheme

Assume we are given a formula $\varphi$, an interpretation $I$, and a declaration $D$. Furthermore, let $f$ be a function symbol occuring in $\varphi$ which is not interpreted by $I$. Then the function encoding scheme for $f$ looks as follows.

- Replace each occurrence of the form $f(t_1, \ldots, t_k)$ in $\varphi$ 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 $\varphi$. 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 $\hat{t_1} = \hat{u}_1 \wedge \cdots \wedge \hat{t_k} = \hat{u}_k \quad \Rightarrow \quad v_f^i = v_f^j$ as an antecedent to the transformed formula.

**Example 7** *Applying the function encoding to the formula* $\varphi \stackrel{def}{=} (f(x, f(y, y)) = z \wedge x = y \wedge f(x, x) = x) \Rightarrow x = z$ *results in:*

$$\left( \begin{array}{l} \wedge((x = y \wedge v_2 = y) \Rightarrow v_1 = v_2) \\ \wedge((x = x \wedge v_2 = x) \Rightarrow v_1 = v_3) \\ \wedge((y = x \wedge x = y) \Rightarrow v_2 = v_3) \end{array} \right) \Rightarrow [(v_1 = z \wedge x = y \wedge v_3 = x) \Rightarrow x = z]$$

*It is not difficult to check that both formulas are logically equivalent, indeed both are valid formulas under any declaration $D$ and any interpretation $I$ which does not interpret function symbol $f$.*

Let us denote by $\mathsf{f\text{-}enc}(\varphi)$ the result of applying the function encoding to formula $\varphi$.

**Theorem 8** *Given a formula $\varphi$, an interpretation $I$, and a declaration $D$. Let $f$ be a function symbol which is not interpreted by $I$. Then $\varphi$ is valid w.r.t. $(I, D)$ iff $\mathsf{f\text{-}enc}(\varphi)$ is valid w.r.t. $(I, D)$.*

### 4.3 Level-zero abstraction

In Level-zero abstraction we consider the validity of the generated proof obligation w.r.t. an interpretation $I$ which only gives interpretations to (polymorphic) equality, boolean functions (i.e. functions with boolean domain and range) and if-then-else. All remaining function symbols are left uninterpreted and are successively removed by the above scheme. Let $\mathsf{F\text{-}enc}(\varphi)$ denote the resulting formula after elimination.

$\mathsf{F\text{-}enc}(\varphi)$ belongs to a fragment of first order logic formulas which are constructed from equality, boolean combinators and variables/constants of type boolean and integer. This fragment has a small model property which we exploit to check the validity of $\models_I^D \mathsf{F\text{-}enc}(\varphi)$.

In order to limit the domains over which we have to interpret the integer variables in $\mathsf{F\text{-}enc}(\varphi)$ we "re-size" the integer constants appearing in $\mathsf{F\text{-}enc}(\varphi)$ by replacing them by smaller integer constants. This replacement has to be so that the resulting formula, which we denote by $\mathsf{CF\text{-}enc}(\varphi)$, is logically equivalent to $\mathsf{F\text{-}enc}(\varphi)$. Since no ordering information can be expressed in the considered fragment of first order logic (note that we treat at Level-zero also comparison functions as being uninterpreted), we can use the following constant encoding scheme for Level-zero abstractions.

Let $\mathcal{C}$ denote the set of integer constants appearing in $\mathsf{F\text{-}enc}(\varphi)$, and let $|\mathcal{C}|$ denote the size of $\mathcal{C}$. Let $\pi$ be any bijection from $\mathcal{C}$ to $\{0, \ldots, |\mathcal{C}|-1\}$.

The constant encoding consists of replacing each constant $c \in \mathcal{C}$ by its small-integer encoding $\pi(c)$. Let $\mathsf{CF\text{-}enc}(\varphi)$ denote the result of applying the constant encoding transformation to $\mathsf{F\text{-}enc}(\varphi)$. The following claim, where interpretation $I$, declaration $D$, $\mathsf{F\text{-}enc}(\varphi)$ and $\mathsf{CF\text{-}enc}(\varphi)$ are defined as above, justifies this encoding.

**Theorem 9** *Formula $\mathsf{F\text{-}enc}(\varphi)$ is valid w.r.t. $(I, D)$ iff $\mathsf{CF\text{-}enc}(\varphi)$ is valid w.r.t. $(I, D)$.*

Finally, in order to check the validity of $\mathsf{CF\text{-}enc}(\varphi)$ w.r.t. $(I, D)$ we alter the standard declaration $D$ to a declaration $D^*$ which associates *finite* types with all variables previously declared to be integers.

Let $N$ denote the number of distinct variables appearing in $\mathsf{CF\text{-}enc}(\varphi)$ and, as before, let $|\mathcal{C}|$ denote the number of distinct integer constants appearing in $\mathsf{CF\text{-}enc}(\varphi)$. Since $\mathsf{CF\text{-}enc}(\varphi)$ has been obtained by applying the constant-encoding transformation, we know that all of these constants lie in the range $\{0, \ldots, |\mathcal{C}| - 1\}$. Let $D^*$ denote the modified declaration in which all integer variables are redeclared to belong to the integer sub-type $\{0, \ldots, |\mathcal{C}|+N-1\}$.

In the following claim let $I$, $D$, $D^*$ and $\mathsf{CF\text{-}enc}(\varphi)$ be as defined as above.

**Theorem 10** *Formula* $\mathsf{CF\text{-}enc}(\varphi)$ *is valid w.r.t.* $(I, D)$ *iff it is valid w.r.t.* $(I, D^*)$.

Now, validity of $\mathsf{CF\text{-}enc}(\varphi)$ w.r.t. $(I, D^*)$ is a sufficient condition for the validity of $\varphi$ w.r.t. the original interpretation $J$ and the original declaration $D$. However, as in all abstraction approaches, if $\models_I^{D^*} \mathsf{CF\text{-}enc}(\varphi)$ does not hold we can not conclude anything for the validity of the original formula. Thus we suggest to use a more refined abstraction if the Level-zero abstraction failed.

### 4.4 Level-one abstraction

In Level-one abstractions we consider the validity of $\varphi$ w.r.t. an interpretation $I$ which gives interpretations to equality, boolean functions, if-then-else and additionally to comparison operators on integers. All remaining function symbols are left uninterpreted and are successively removed by the function encoding scheme. Again, let $\mathsf{F\text{-}enc}(\varphi)$ denote the resulting formula after elimination.

This formula also belongs to a first order logic fragment which has a small model property. In order to limit the domains over which we have to interpret the integer variables in $\mathsf{F\text{-}enc}(\varphi)$ we have to use a different constant encoding scheme than in the Level-zero abstraction since now ordering information amongst variables and constants can be expressed and thus has to be preserved. Instead of mapping constants to constants we now map constants into fresh variables.

Let $\mathcal{C} = \{c_1, \ldots, c_m\}$ be the set of constants appearing in $\mathsf{F\text{-}enc}(\varphi)$ where $c_1 < \cdots < c_m$. We introduce new variables $v_{c_1}, \ldots, v_{c_m}$ and define for a pair $(c_1, c_2)$, $c_1 < c_2$, of constants the following clause ($N$ denotes again the number of distinct variables appearing in $\mathsf{F\text{-}enc}(\varphi)$):

$$\mathsf{const}(c_1, c_2) = \begin{cases} v_{c_1} < v_{c_2} \wedge v_{c_2} - v_{c_1} \leq (c_2 - c_1) & \text{if } c_2 - c_1 \leq N \\ v_{c_2} > v_{c_1} & \text{if } c_2 - c_1 > N \end{cases}$$

Then, we add the predicate $\mathsf{const}(c_1, c_2) \wedge \ldots \wedge \mathsf{const}(c_{m-1}, c_m)$ as antecedent to the transformed formula. Again, let interpretation $I$, declaration $D$, $\mathsf{F\text{-}enc}(\varphi)$ and $\mathsf{CF\text{-}enc}(\varphi)$ be as defined above.

**Theorem 11** *Formula* $\mathsf{F\text{-}enc}(\varphi)$ *is valid w.r.t.* $(I, D)$ *iff* $\mathsf{CF\text{-}enc}(\varphi)$ *is valid w.r.t.* $(I, D)$.

Finally, the standard declaration $D$ is altered. Let $N$ denote the number of distinct variables appearing in $\mathsf{CF\text{-}enc}(\varphi)$. Let $D^*$ denote the modified declaration in which all integer variables are redeclared to belong to the integer sub-type $\{0, \ldots, N-1\}$. The following theorem justifies this transformation where $I$, $D$, $D^*$, and $\mathsf{CF\text{-}enc}(\varphi)$ are defined as above.

**Theorem 12** *Formula* $\mathsf{CF\text{-}enc}(\varphi)$ *is valid w.r.t.* $(I, D)$ *iff it is valid w.r.t.* $(I, D^*)$.

Obviously, Level-one yields more faithful abstractions than Level-zero. However, what do we do in case that also Level-one fails? Currently we are elaborating a hierarchy of abstractions by removing less interpretations of function symbols from the original formula. However, for the purpose of code validation our experience suggests that Level-one and Level-one will be sufficient to establish validity of the proof obligations if the generated code is indeed a correct implementation of its DC+ source.

## 5 Conclusion

We have presented the theory which underlies our fully automatic translation validation approach for optimizing industrial compilers from DC+to C. The insertion of history variables into the C code, the translation of DC+ and C programs to STS, the compression of the concrete transition relation, the generation of the substitution $\alpha$ and the final assembling of the proof obligations according to Rule $\mathsf{ref}$ have all been implemented in TVT (Translation Validation Tool). TVT uses the decision procedures explained in Section 4 in order to check the validity of the generated proof obligations.

Towards scalability of TVT: We haven't yet tried to validate the 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 complete DC+ model has more than 1000 variables. When compiled as one monolithic unit – which gives the most efficient code – the generated C program is more than 250KB. It is not very likely that TVT scales up to this size.

However, if the DC+ model is decomposed into separate compilation units (SCU's) then there definitely is the possibility (depending on the size of the SCU's) to apply TVT for separately validating the generated modules. The decomposition mechanisms for DC+ are such that we can exploit a very simple compositionality property which indeed allows to inspect the generated modules in isolation (SCU's are only allowed to be sequentially composed, i.e. there is no communication back and forth between SCU's). This means that there is a trade-off between the efficiency of the generated code (requires large SCU's)and its verifiability (which requires small SCU's).

**Future work:** Currently the generated C implementations of DC+ programs are centralized. Nevertheless, the code generators that we considered are currently extended to also generate *distributed* implementations. We started to investigate the adaptation of the presented framework to also cope with distributed implementations. The main idea here is to reconstruct a centralized implementation for the distributed implementation and then to apply the presented techniques.

Furthermore we currently look for more faithful abstractions, corresponding new constant encoding schemes and better lower bounds for the finite domains over which the transformed proof obligations are interpreted.

## References


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

[2] A. Benviniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language. *Science of Computer Programming*, 16, 1991.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[4] P. Caspi, N. Halbwachs, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

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

[6] A. Cimatti, F. Giuchiglia, and P. Pecchiari et al. A provably correct embedded verifier for the certification of safgety critical software. In *CAV*, number 1254 in LNCS. Springer, 1997.

[7] The declarative code DC+. ESPRIT Project: SACRES, Project Report, 1997. Version 1.3.

[8] *Another look at real-time programming*, volume 9 of *Proc. of the IEEE*, September 1991.

[9] F. Maraninchi. Operstional and compositional semantics of synchronous automata compositions. In *Proceedings CONCUR*, volume 630 of *LNCS*. Springer, 1992.

[10] A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer-Verlag, 1998.

[11] privat communications with TNI (BREST), Siemens (Munich) and Inria (Rennes).