Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

The Code Validation Tool (CVT) *

Automatic verification of a compilation process

A. Pnueli, O. Shtrichman, and M. Siegel **

Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Rehovot, Israel.

The date of receipt and acceptance will be inserted by the editor

Abstract. We describe CVT - a fully automatic tool for Code-Validation, i.e. verifying that the target code produced by a code-generator (equivalently, a compiler or a translator) is a correct implementation of the source specification. This approach is a viable alternative to a full formal verification of the code-generator program, and has the advantage of not 'freezing' the code generator design after verification.

CVT was developed in the context of the ESPRIT project SACRES, and validates the translation from StateMate/Sildex mixed specification into C. The use of novel techniques based on uninterpreted functions and their analysis over a BDD-represented small model enables us to validate source specifications of several thousands lines, which represents a typical industrial size safety-critical application.

Keywords: compiler verification, translation validation, code validation.

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, for-example, [2] [3] [5] [10],[7],[6],[9], and [8]), 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 their designs, 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.

* This research was done as part of the ESPRIT project SACRES, and was supported in part by a grant from the Deutsche Forschungs Gemeinschaft, the Minerva Foundation, and an infrastructure grant from the Israeli Ministry of Science and Art.

** e-mail: {amir|mis|ofers}@wisdom.weizmann.ac.il

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. In the next section we describe the SACRES project and the role of code validation in this context. In Section 3 we briefly describe the logical basis of the correctness proof. In Section 4 we describe the architecture of CVT and the role of each of its modules, and we summarize in Section 5 by presenting results from an industrial case study that was one of the pilot applications considered within the SACRES project.

2 Code validation in the context of the SACRES project

The Code Validation Tool (CVT) is developed as part of the ESPRIT-supported project SACRES [4](which stands for *Safety Critical Real-time Embedded Systems*). The objective of this project is to provide designers of safety-critical systems with an enhanced design methodology supported by a toolset, significantly reducing the risk of design errors and shortening the overall design time. The emphasis within the project is on formal development of systems, providing formal specification, formal verification supported by model checking technology, and validated code-generation.

The architecture of the SACRES toolset is shown in Fig. 1.
Following is a typical scenario of usage of the toolset: After completing the design in her favorite design tool (currently the 'StateMate' and 'Sildex' tools are supported), the user invokes the automatic translation of designs into DC+, a common format for synchronous languages. The design can be mixed: different components can be designed using different tools, as long as these tools are supported within the toolset. In the next step the user invokes the *Proof-Manager*, and performs *component and system verification*. In this stage the user verifies that the design satisfies various properties, which she expresses in the requirement specification language of *Timing Diagrams*, using the *Timing-Diagrams Editor* (TDE). These properties typically correspond to the requirements listed in a requirement document, or to general safety and liveness properties of the system, such as the absence of deadlocks.

The Proof-Manager combines BDD-based automatic verification tool and a theorem-prover, which is invoked when the automatic verification fails (typically due to the size of the model). The various components thus can be verified by different means, while the proof-manager guarantees that the necessary compositionallity requirements are maintained. If the system finds a design error, it presents a counter example by means of simulation (either in StateMate or in TDE).

After the design is verified, the user invokes the code generator (produced by the SACRES partner TNI) to automatically generate executable code (C or ADA). This is where the code validation tool is invoked: The validation of the generated code via CVT establishes that the code generator worked as expected and thus the properties which were verified at the specification level are preserved at the implementation level. We expect that the process of code validation will provide the convincing evidence required by the certification authorities in order to allow the use of automatic code generators for the development of safety-critical systems.

3 Code Generation and "Correct Implementation"

The first step in proving the correctness of the compilation process is to define when a generated C program correctly implements its DC+ source program. For this definition, we will briefly explain the semantics of DC+ programs and the basic ideas of the compilation process. Afterwards we motivate a relation between C programs and DC+ programs which will serve us as the 'correct implementation' relation.

DC+ is a synchronous, declarative language. A DC+ program describes a reactive system whose behavior along time is observable as an infinite sequence of states. State changes are triggered by the arrival of new values for the *input* variables. A list of constraints (on program variables), which constitutes the main part of the DC+ program, determines upon such an arrival the values of all the remaining variables. These remaining variables consist of a set of *output* variables, a set of *internal* variables, and a set of *register* variables which store information about the history of the current computation, such as values of expressions at previous time instances. The list of constraints determines the *transition relation* of the system. At each instance in time all constraints have to be satisfied by the values that the variables have at that instance, and their values at the next state.

In order to perform code validation, both the DC+ and the generated C program need to be translated into a common semantic domain. *Synchronous transition systems* (STS) as introduced in [14] turned out to be a convenient candidate for such a translation. An STS $S = (V, \Theta, \rho)$ consists of a finite set V of typed variables, a satisfiable assertion Θ characterizing the initial states of system S, and a transition relation ρ. Basically, a DC+ program D is translated into $S = (V, \Theta, \rho)$ as follows: the set V contains all program variables of D, the initial condition Θ characterizes the initial state of D, and the transition relation ρ is obtained by a one-to-one translation of the list of constraints of D into logical formulas involving primed and unprimed variables. Unprimed and primed variables refer to the values of these variables - the current and the next state, respectively. The result-

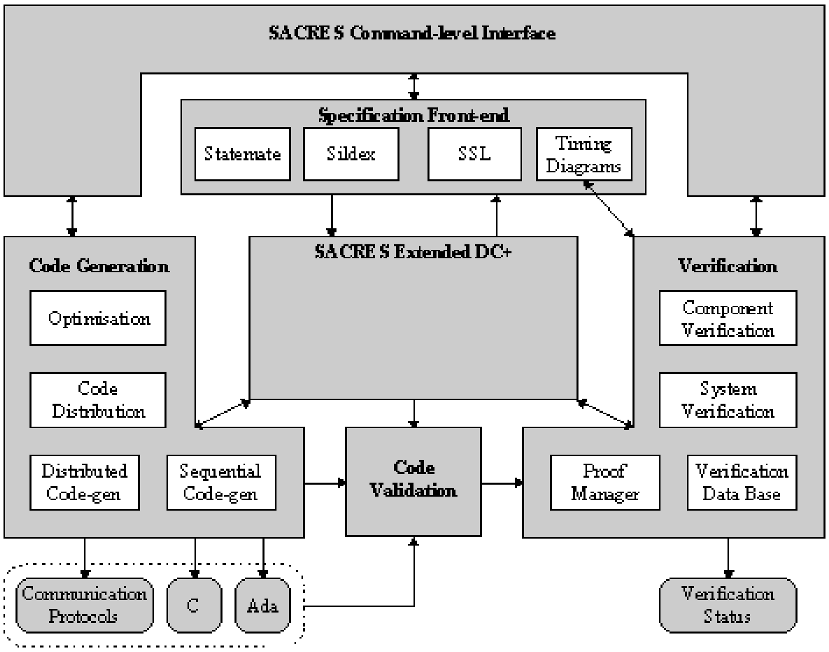
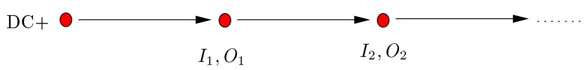


Fig. 1. The SACRES Architecture

ing ρ is a conjunction of these formulas which restrict the possible values of variables in the next state.

The conjuncts in the relation ρ constitutes a *Set of Logical Equations* (*SLE*) on the variables in V. Solutions of this *SLE* for given values of the input variables determine the values of the remaining variables. Conceptually, the observable behavior of such a system can be understood as shown in the figure below where the first dot depicts an initial state of the system.

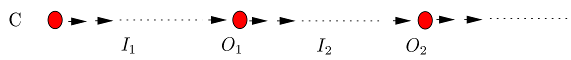


Note, that in accordance with the synchrony hypothesis there is no time delay between the reception of new values I_j for input variables and the generation of corresponding output values O_j satisfying *SLE*, i.e. all variables are updated *simultaneously*.

3.1 The Compilation Schema

The task of the code generator is to derive from a given DC+ program D a C program which computes, for given values of the input variables, a solution of the *SLE* of D.

The obtained C program belongs to a fragment of ANSI-C and has the typical structure of control systems: exactly one control-loop, where one iteration corresponds to one step of the DC+ program. Whereas the values for all variables in DC+ were updated simultaneously, the control-loop first consumes new values for input variables and afterwards successively computes (one-by-one) the values of the remaining variables as shown in the next figure.

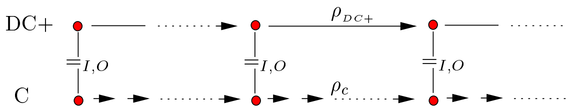


The states marked with a bullet, corresponding to the begin (and also end) of the control-loop, match the states of the original DC+ program. Intermediate states, where only some variables have been updated, are not depicted since they do not correspond to any state of the DC+ program.

As stated before, the C program is also translated into an STS representation for the purpose of a semantical comparison. In the rest of this section the letters C and DC+ will denote the STS representation of the respective programs.

3.2 The "Correct Implementation" Relation

The notion of *correct implementation* used in CVT is a variation of the standard *language inclusion* relation as commonly used in refinement theory. Program C *implements* DC+ denoted by $C \,\mathsf{ref}\, \mathrm{DC+}$, if for every computation σ of C there exists a computation τ of DC+ such that σ and τ agree state-wise on the values of observable variables, i.e. input and output variables, as depicted below.



It follows that we can identify a set of observable variables $\mathcal{O}$ which are common to the C and DC+ programs. For simplicity and clarity we prefer to keep the sets of variables of the two programs disjoint. Consequently, we rename the concrete C-version of every observable variable $x \in \mathcal{O}$ to x_c. Thus, the requirement of correct implementation can be stated as:

For every computation of C (concrete computation) $\sigma : s_0, s_1, \ldots,$ *there exists a computation of DC+ (abstract computation)* $\sigma^a : S_0, S_1, \ldots$ *such that, for every* $i = 0, 1, \ldots$ *and every* $x \in \mathcal{O}$, $S_i[x] = s_i[x_c]$, *i.e.,* S_i *and* s_i *assign to* x *and* x_c *identical values.*

We use proof techniques from refinement theory in order to establish that C ref DC+ indeed holds for two given systems. The standard constructive proof technique requires to devise an abstraction mapping f, which maps computations σ of the concrete system C to computations $f(\sigma)$ of the abstract system DC+ such that σ and $f(\sigma)$ agree on the values of the observable variables. Typically, f is induced by a state mapping α from concrete states to abstract states. Syntactically, α is specified by a substitution $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ which assigns to every abstract variable x_i a term over the concrete variables. For a concrete state s, we define the corresponding abstract state $S = \alpha(s)$ as the state in which, for all x_i, $S[x_i]$ equals $s[t_i]$, i.e. the value of the term t_i in the state s. Thus, we can view α both as a substitution and a state mapping.

The fact that f_α indeed is a an abstraction mapping is proven by induction on the length of computations. The proof obligations are:

0. The substitution α replaces every observable variable $x \in \mathcal{O}$ by its concrete version x_c, i.e. $x[\alpha] = x_c$.
1. The mapping α maps initial concrete states to initial abstract states.
2. Concrete transitions are mapped by α to possible abstract transitions. That is, if s_2 is a C-successor of s_1, then $\alpha(s_2)$ is a (DC+)-successor of $\alpha(s_1)$.

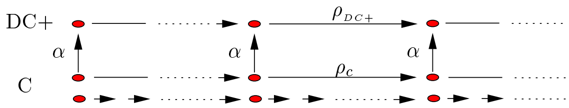
Usually, finding such a mapping α is left to the ingenuity of the verifier. In the context of translation validation it is essential that α can be *automatically constructed* from the source and target programs.

In order to facilitate the generation of suitable refinement mappings, we perform a transformation of the transition relation of C. Originally, the transition relation in C updates variables successively till finally the result of the simultaneous update of variables in DC+ has been computed. We construct a new transition relation which reflects the *accumulated* effect of the individual steps in the execution of the loop's body, as illustrated in the figure below.

$$\begin{array}{lll}
\textbf{R1}.\ \Theta_C & \Rightarrow\ \Theta_{DC+}[\alpha] & \text{Initiation} \\
\textbf{R2}.\ \rho_C & \Rightarrow\ \rho_{DC+}[\alpha] & \text{Propagation} \\
\hline
& C \text{ ref } DC+ &
\end{array}$$

Rule ref: Proving Refinement

Fig. 2. The proof obligations of Rule ref



While standard state mappings in refinement theory reconstruct the values of *all* abstract variables from the values of their concrete counterparts, this was not possible in our case due to the more than 100 optimization rules applied by the code generator. These optimizations eliminate for example (whenever possible) internal variables of the DC+ program such that a complete reconstruction of abstract states is impossible without additional (reliable!) information about the code generation process. We dealt with this variable elimination process by hiding all internal variables in the formulation of the proof obligation by means of existential quantification. For more technical details on the automatic construction of suitable state mappings solely from the given programs, we refer to [13].

All in all, CVT automatically generates the two proof obligations corresponding to the premises of Rule ref (see Fig. 2). If both of these proof obligations are found to be valid, we can conclude that C is a correct refinement of the corresponding DC+ program.

The Verification Condition Generator, which is the first module invoked in CVT, generates these implications from the C and DC+ source codes. The formulas ρ_{DC+} and ρ_C are both large conjunctions of atomic sub-formulas, where typically (but not always) each sub-formula corresponds to an assignment line in the code or a constraint imposed by the abstraction (see Section 4.3). These sub-formulas reflect the semantics of the source languages and the mapping between their variables.

4 Architecture of CVT

The code-validation tool offers a fully automatic utility which establishes the correctness of the generated code individually for each run of the code generator. Therefore, there is no user-interface to this tool - just configuration parameters and a command line. The overall architecture of CVT is presented in Fig. 3. In the next sections we will explain what is the role of each module and what are its inputs and outputs. In general, only the first module (the verification-condition module) is dependant on the specific languages and compiler we

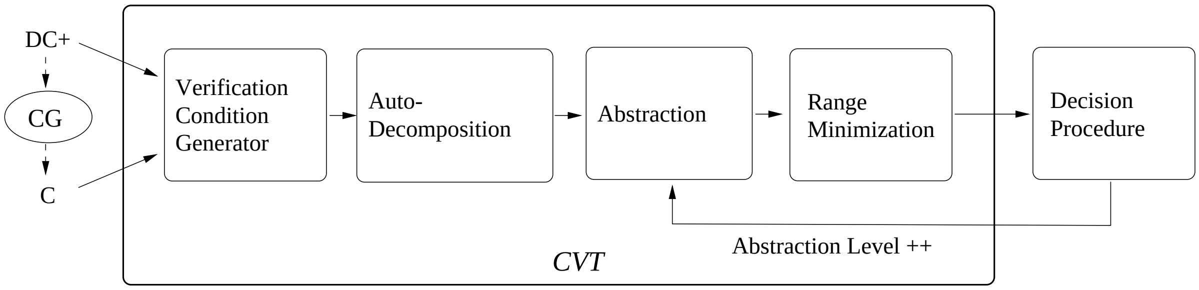


Fig. 3. The CVT Architecture

considered. All the other modules serve as the decision procedure of CVT, and can be reused (under minor adjustments) in validation of other compilers.

4.1 The Verification Condition Generator module

CVT receives as input the DC+ and C source codes. These are the source and target code for the code generator. Two separate sub-modules (combined into the Verification Condition Generator module in Fig. 3) generate the verification condition (which is actually a large logical implication) by means of various translations and transformations. The validity of this logical implication implies the correctness of the generated code w.r.t. the source code while its invalidity indicates a potential mistake in the code generation process. Each of the conditions is separated into two files representing the left and right hand side of the implication (in R2 these are ρ_C and ρ_{DC+}). Since at the end of this process we use TLV [12], the verification condition is generated in the appropriate format (the models TLV expects are compatible with the more broadly used SMV model-checker [1]).

4.2 The Auto-Decomposition module

The next step is Auto-Decomposition. We are interested in handling industrial-size programs, and therefore decomposition is essential. As will be demonstrated in Section 5, the auto-decomposition is one of the key enabling steps for scalability. The Auto-Decomposition module takes advantage of the fact that the right hand side of the implication is in the form of a conjunction (typically of hundreds of sub-formulas), and simply breaks it into smaller conjuncts which can be verified independently.

Since the time it takes to verify a program using BDD based tools is worst-case exponential in the size and complexity of the formula, it is the size of the single formula that has to be verified that determines the bottleneck of the validity checking. The complexity of solving each task may depend exponentially on the number of conjuncts appearing in the antecedent in the case that most of them are conditional. This is because each condition (if-then-else) introduce a case splitting which together are compounded into exponentially many cases. Note that conditional statements arise not only from conditions in the C-program, but also from the definition of the internal variables in the (DC+)-program, and from the α substitution, which also appear in the antecedent of the implication.

The Auto-Decomposition module breaks a program that is n times longer into n times more separate files to validate. Each of these files represent a smaller verification task, although in most cases it is larger than $1/n$ of the original formula, as will soon be explained. Thus, although there is a *linear* increase in the number of validation tasks, there is an exponential decrease in the validation time of each of these tasks. This phenomena, which is characteristic of verification tasks, is the reason why decomposing the formula is so important, and why the possibility to split the formula to as many sub-formulas as the number of conjuncts is a significant factor in the attained scalability of CVT.

The size of the decomposed verification condition is optionally set by a configuration parameter (called the 'chunk size'), and can range from 1 (a single conjunct) to the total number of conjuncts. In the later case the entire formula will be verified at once, which is only possible for relatively small files. After breaking the right hand side, the Auto-Decomposition module returns to the left hand side of the implication, and calculates the *Cone of Influence* (COI), i.e. the portion of the formula in the left hand side that is needed for proving the selected conjuncts on the right-hand side.

This is the way the COI is calculated: as a first step CVT makes a list of all the variables that are used in the right hand side of the implication (obviously the smaller the right hand side is, the shorter the list). Assume x_i is such a variable. CVT looks for the definition of x_i on the left hand side, which is an expression over other variables $x_1 \ldots x_k$. It then erases x_i from the list and adds each of the variables $x_i \ldots x_k$ that were not in this list before, instead. This procedure is repeated until the list is empty. At the end, the only conjuncts retained on the left-hand-side are the defining equations for the variables that were considered throughout the computation.

Example 1. Consider the following implication, where x_1 and x_7 are input variables:

stage	variables list	now looking at...	cone of influence
1	x_4, x_5, x_6	x_4	$x_4 = x_1$
2	x_5, x_6, x_1	x_5	$x_4 = x_1 \wedge x_5 = 2$
3	x_6, x_1	x_6	$x_4 = x_1 \wedge x_5 = 2 \wedge x_6 = x_3$
4	x_1, x_3	x_1	$x_4 = x_1 \wedge x_5 = 2 \wedge x_6 = x_3$
5	x_3	x_3	$x_4 = x_1 \wedge x_5 = 2 \wedge x_6 = x_3 \wedge x_3 = x_7$
6	x_7	x_7	$x_4 = x_1 \wedge x_5 = 2 \wedge x_6 = x_3 \wedge x_3 = x_7$

Fig. 4. Calculating the Cone of Influence

$$x_4 = x_1 \wedge x_2 = x_6 \wedge x_6 = x_3 \wedge x_3 = x_7 \wedge x_5 = 2 \quad \rightarrow \quad x_4 = x_5 + x_6 \wedge x_6 = x_7 + x_5$$

The Auto-Decomposition module will split the right hand side into two files, one for each conjunct. The calculation of the COI for the first one appears in Fig. 4.

Thus, the cone of influence of $x_4 = x_5 + x_6$ is made of four conjuncts, excluding the conjunct $x_2 = x_6$. If we did not decompose the file, the formula would have consisted of two conjuncts in the right and five in the left. ┛

After repeating this process until all conjuncts are covered, we are left with (possibly hundreds) pairs of files, each significantly smaller than the original ones. There is an obvious tradeoff between having files with very small right hand side, which leads to significantly shorter verification time, and the number of these files which incurs an additional invocation overhead cost associated with each file. It is therefore left to the user to decide on the chunk size which may be optimal for her case.

Another configuration parameter is called 'Reverse Cone (RC)'. When this flag is set, after calculating the cone of influence, the program returns to the right hand side and looks for additional conjuncts that can be proven with the same cone that was just calculated. This option is useful for reducing the number of files and reducing the over-all time for performing the proof (the time TLV takes mainly depends on the transition relation, i.e. the left hand side. Thus if we use the same formula for proving more conjuncts, we save time). When setting this option, the 'chunk size' is no longer an exact number of conjuncts taken each time, rather it is the size of the initial set of conjuncts, which possibly grows after the RC calculation. The efficiency of the RC calculation obviously depends on the ordering of conjuncts we are investigating. An optimal ordering would be such that if $cone(C_i) \subseteq cone(C_j)$ then C_i and C_j are verified together (with simple sequential ordering this will happen only if C_j appears first or if $cone(C_i) = cone(C_j)$). This ordering can be achieved, for example, by calculating all the cones and then partitioning the files accordingly. We did not implement this because we suspected that the overhead of this calculation will be larger than the saving resulting from the better ordering.

4.3 The Abstraction Module

After decomposing the files, CVT invokes the Abstraction module. The underlying theory of the abstraction is detailed in [13]. Basically, abstraction is needed since we are trying to verify a formula which contains integer and float variables, as well as functions over these variables using a BDD-Based decision procedure for finite-state models. The abstraction module treats these functions as *uninterpreted functions*, replacing them by new symbols. The faithfulness of this technique depends on the way that the compiler manipulates these functions and the kind of functions we leave interpreted. The more we interpret, the more faithful the model is. On the other hand, the less we interpret, the smaller the model is.

The abstraction works in an incremental manner, following an *abstraction hierarchy* designed according to the specific optimizations the compiler performs. We begin with maximum abstraction (called *Level-0 abstraction*) where all functions except equalities, Boolean operators and if-then-else are left uninterpreted. If the proof fails, CVT invokes the abstraction module again, asking for *Level-1 abstraction* where, additionally, comparisons operators on integers ('>', '<', etc) are now interpreted.

If, for example, the compiler reads '$a < b$' in the abstract system and transforms it to '$b > a$' in the concrete system (which are obviously semantically equivalent), Level-0 abstraction will result in a false negative (i.e. the abstracted formula is pronounced invalid, while the concrete formula is valid) where as level-1 will succeed.

The function encoding scheme works as follows: assume we are given a formula φ, and let f be a function symbol occurring in φ. Then the function encoding scheme for f looks as follows.

- Replace each occurrence of the form $f(t_1,..,t_k)$ in φ by a new variable v_f^i of a type equal to that of the value returned by f. Occurrences $f(t_1,..,t_k)$ and $f(u_1,..,u_k)$ are replaced by the same v_f^i iff t_j is identical to u_j for every $j = 1,\dots,k$.
- For every pair of newly added variables v_f^i and v_f^j, $i \neq j$, corresponding to the non-identical occurrences $f(t_1,..,t_k)$ and $f(u_1,..,u_k)$, add the implication $t_1 = u_1 \wedge \ldots \wedge t_k = u_k \;\rightarrow\; v_f^i = v_f^j$ as an antecedent to the transformed formula.

Example 2. Assume that a source program contained the statement $z := (x_1 + y_1) \cdot (x_2 + y_2)$ which the translator we wish to verify compiled into the following sequence of three assignments:

$u_1 := x_1 + y_1;\ u_2 := x_2 + y_2;\ z := u_1 \cdot u_2,$

introducing the two auxiliary variables u_1 and u_2.

For this translation, the abstraction module will construct the verification condition

$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 \cdot u_2 \quad \to \quad z = (x_1 + y_1) \cdot (x_2 + y_2),$

whose validity we wish to check.

The abstracted version of the above implication is:

$u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2) \quad \to \quad z = G(F(x_1, y_1), F(x_2, y_2))$

Clearly, if the abstracted version is valid then so is the original concrete one.

Following the abstraction schema, the abstraction module now replaces each functional term by a fresh variable but adding, for each pair of terms with the same function symbol, an extra antecedent which guarantees the functionality of these terms. Namely, that if the two arguments of the original terms were equal, then the terms should be equal. It is not difficult to see that this transformation preserves validity. We thus obtain the following equality formula:

$$\varphi\colon \quad \begin{pmatrix} (x_1 = x_2 \wedge y_1 = y_2 \to f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \to g_1 = g_2) \wedge \\ u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \end{pmatrix} \to z = g_2 \tag{1}$$

Note the extra antecedent ensuring the functionality of F by identifying the conditions under which f_1 should equal f_2 and the similar requirement for G. This shows how equality formulas such as φ of Equation (1) arise in the process of translation validation. ▟

The reason we first interpret the comparison operators on moving from level-0 to level-1 is that the compiler we are considering employs these kinds of optimizations frequently. To handle commutativity of the '+' function, for example, we need another abstraction level. However, so-far Level-0 and Level-1 abstractions proved to be sufficient for the purposes of code-validation of the study-cases we have considered.

This leaves us with a quantifier-free first-order logic formula which enjoys the small model property (i.e. it is satisfiable iff it is satisfiable over a finite domain). Therefore the next issue the abstraction module handles is the calculation of a finite domain, such that the formula is valid if and only if it is valid over all interpretations into this domain. The latter can be checked algorithmically, using BDD techniques. The domain that is many times taken when using these techniques 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 ordering of 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 orderings, is sufficient regardless of the formula's structure.

4.4 *The Range-Minimization module*

The size of the state-space imposed by the $[1..n]$ range as suggested in the previous section is n^n. For many industrial-size programs this state-space is far too big to handle. For example, a program with 100 variables requires a state space of size 100^{100} which obviously can not be handled in reasonable time. But apparently there is a lot of redundancy in this range that can be avoided. The $[1..n]$ range is given without any kind of analysis of the formula's structure. Note that the informal soundness proof we described before, is independent of the structure of the formula we try to validate, and thus the range is sufficient for all formulas with the same number of variables. But in fact, there is no reason why we should treat the validated formula as arbitrary. Analyzing the structure of the formula we wish to validate makes it possible to significantly decrease the ranges and therefore decrease the state space. This analysis is performed by the 'Range Minimization' module (RMM), using the *range allocation algorithm*, which significantly reduces the range of each of these (now enumerated type) variables, and thus increase the size of programs we can handle. By invoking this module 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 results 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 a few seconds.

The range allocation algorithm is somewhat complex and its full description is beyond the scope of this paper. We refer the reader to [11] for more details, and describe here only the general idea. The algorithm is relevant at this point to Level-0 abstraction only, hence it analyzes formulas where all functions are abstracted except the equality function and the standard boolean operators.

The range allocation algorithm tries to solve a satisfiability (validity) problem efficiently, by determining a *range allocation* $R : \mathit{Vars}(\varphi) \mapsto 2^{\mathbb{N}}$, 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, 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 natural, 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.

4.4.1 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 $\rightarrow$, $\leftrightarrow$ 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 $\psi_1 \wedge \cdots \psi_k$ 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. Variable x_i is called a *mixed variable* iff $(x_i, x_j) \in A_=$ and $(x_i, x_k) \in A_{\neq}$ for some $x_j, x_k \in Vars(\varphi)$.

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 3. Let us illustrate these concepts on the formula φ of Equation (1), whose validity we wished to check.

Since our main algorithm checks for satisfiability, we proceed by calculating the positive form of $\neg\varphi$, which is given by:

$$\neg\varphi\colon \quad \left(\begin{array}{c} (x_1 \neq x_2 \ \vee\ y_1 \neq y_2 \ \vee\ f_1 = f_2) \wedge \\ (u_1 \neq f_1 \ \vee\ u_2 \neq f_2 \vee\ g_1 = g_2) \wedge \\ u_1 = f_1 \ \wedge\ u_2 = f_2 \ \wedge\ z = g_1 \end{array}\right) \ \wedge\ z \neq g_2,$$

and therefore

$$\begin{aligned} A_= &= \{(f_1 = f_2),\ (g_1 = g_2),\ (u_1 = f_1),\ (u_2 = f_2), \\ &\quad (z = g_1)\} \\ A_{\neq} &= \{(x_1 \neq x_2),\ (y_1 \neq y_2),\ (u_1 \neq f_1),\ (u_2 \neq f_2), \\ &\quad (z \neq g_2).\} \end{aligned}$$

Note that u_1, u_2, f_1, f_2, g_2 and z in this example are mixed variables.

As explained above, the sets $A_=$ and $A_{\neq}$ can be computed directly by counting the number of negations enclosing the atomic formulas in φ without transforming to positive form or even explicitly negating φ. For example, the comparison $x_1 = x_2$ in φ is contained within two negations implied by appearing on the left hand side of two (nested) implications. Since we are considering $\neg\varphi$, this amounts to 3 negations. Since 3 is odd, we add $x_1 \neq x_2$ to $A_{\neq}$. In a similar way, the comparison $f_1 = f_2$, being under 2 negations, is added to $A_=$. ∎

This example would require a state-space of 11^{11} if we used the $[1..n]$ range, where $n = 11$. The range allocation algorithm will find ranges adequate for this formula, with a state space of 16.

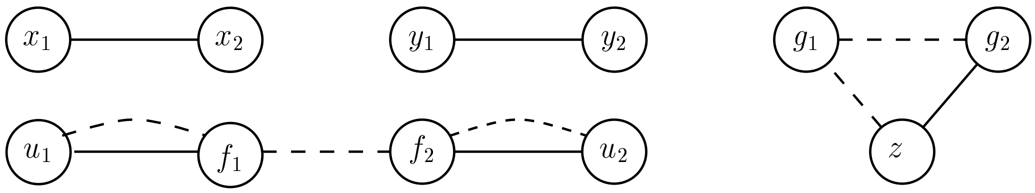


Fig. 5. The Graph $G : G_{\neq} \cup G_{=}$ representing $\neg\varphi$

4.4.2 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. Each vertex in these graphs represents a variable, and therefore some of them represent mixed variables. We refer to these vertices as *mixed vertices*.

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 graphs 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. We once more refer the reader to [11] for further details.

4.5 The Verifier module (TLV)

The validity of the verification conditions is checked by TLV [12], an SMV-based tool which provides the capability of BDD-programming and has been developed mainly for finite-state deductive proofs (and 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. CVT invokes TLV for each pair of files generated by the Auto-decomposition module. A proof log is generated as part of this process, indicating which files were proved, at what level of abstraction and when.

5 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 DC+ 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 4.3, the abstraction module replaces function symbols with new variables). Following is a summary of the results achieved by CVT:

Module	Conjuncts	Verified	Time (min.)
M1	530	100%	1:54
M2	533	100%	1:30
M3	124	100%	0:27
M4	308	100%	2:22
M5	860	99.8%	3:31 + ?
Total :	2355	99.9%	9:44 + ?

As can be seen, only a fragment of a percent (3 conjuncts out of 2355) could not be verified in reasonable time using the current implementation of CVT. These 3 conjuncts had several characteristics that made them hard to check:

- The cone of influence for these conjuncts was very big, and in fact included most of the left hand side of the formula.
- As a result, there was a very large number of functions that had to be abstracted. As explained in Section 4.3, each abstraction requires the addition of constraints. For a two argument function like '+', to abstract n occurrences of the function requires the addition of $O(n^2/2)$ constraints (because the arguments of each pair of functions is compared). In the case of these 3 conjuncts, typically there were more than 50 of each kind of function ('+', '-'...'>' '<' etc) and thus thousands of constraints had to be added.

We are currently working on additional optimizations and improvements of our basic algorithms that, hopefully, will enable us to handle these three cases of last resistance. The progress so far seems to be most encouraging.

References

1. J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

2. B. Buth, K. Buth, M. Franzle, B. Karger, Y. Lakhneche, H. Langmaack, and M. Muller-Olm. Provably correct compiler development and implementation. In *Compiler Construction 92*, 1992.
3. D. Clutterbuck and B. Carre. The verification of low-level code. *Software Engineering Journal*, pages 97–111, 1998.
4. T. S. Consortium. Safety critical embedded systems: from requirements to system architecture, 1995. Esprit Project Description EP 20.897, URL http://www.tni.fr/sacres.
5. 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.
6. J. D. Guttman, J. D. Ramsdell, and V. Swarup. The VLISP verified scheme system. *Lisp and Symbolic Computation*, 8:33–110, 1995.
7. J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
8. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
9. D. P. Oliva, J. D. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8:111–182, 1995.
10. I. M. O'Neill, D. L. Clutterbuck, and P. Farrow. The formal verification of safety-critical assembly code. In *IFAC Symposium on safety of computer control systems*, 1988.
11. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. An efficient algorithm for the range minimization problem. Technical report, Minerva Center for Verification of Reactive Systems at the Weizmann Institute, Dec. 1998.
12. 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.
13. A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 235–246. Springer-Verlag, 1998.
14. 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.