The Code Validation Tool (CVT)–

Automatic verification of code generated from synchronous languages*

A. Pnueli, O. Shtrichman and M. Siegel

Contact author: amir@wisdom.weizmann.ac.il

The Weizmann Institute of Science,
Department of Applied Mathematics and Computer Science
Rehovot, Israel

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.

The CVT tool has been 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.

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, [1][2][3] and [4]), 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 is the key enabling technology to allow the usage of code generators in the development cycle of safety-critical and high quality systems. Remarkably, 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.

The work carried out in the SACRES project proves the feasibility of code-validation for the industrial code generators 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 preliminary results from an industrial case study that we are currently working on.

* 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 infra-structure grant from the Israeli Ministry of Science and Art.

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 (which stands for *Safety Critical Real-time Embedded Systems*)[7]. 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, model checking technology and validated code-generation.
The architecture of the SACRES toolset is shown in Figure 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+, the common format for synchronous languages. The design can be mixed: different components can be designed in different tools, as long as these tools are supported in 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 compositionality 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.

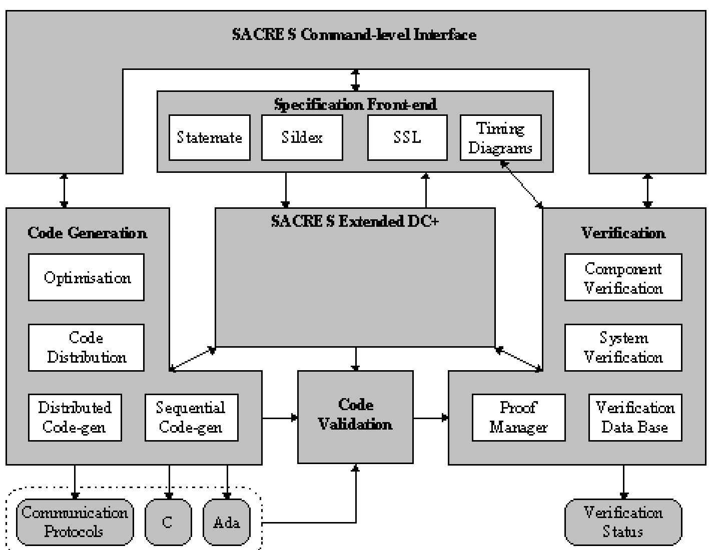

Fig. 1: SACRES Global Architecture

3 The Verification Condition

The theoretical background behind the construction of the verification condition is elaborated in [5]. Following is a brief description of the structure of the verification condition, which, if proven correct, guarantees the correctness of the translation.

In the following, we refer to the DC+ program as the *abstract* system, as it represents the specification, and to the C program as the *concrete* system, as it represents the concrete implementation. We denote the variables, initial condition and transition relation of these systems by V_A, θ_A and ρ_A (abstract) and V_C, θ_C and ρ_C (concrete), respectively. In order to establish that the concrete system correctly implements (*refines)* the abstract system, we use two premises (verification conditions), **R1** – the *base case*, and **R2** – the *induction step*.

The base case requires that θ_C implies θ_A, after performing an appropriate substitution α of each (observable) variable $v \in V_A$ by an expression ε over V_C. Such a substitution induces an (abstraction) mapping between the states of the two systems.

The induction step requires that ρ_C implies ρ_A, once again, after an appropriate substitution α.
Taken all together, the refinement rule has the following structure:

Let α: $V_A \rightarrow \varepsilon(V_c)$ be a substitution
R1: $\theta_c \rightarrow \theta_A[\alpha]$ The base step
R2: $\rho_c \rightarrow \rho_A[\alpha]$ The induction step

C **imp** A

The Verification Condition Generator, which is the first module invoked in CVT, generates these implications from the C and DC+ source codes. ρ_A 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 for more details). These sub-formulas reflect the semantics of the source languages and the mapping between their variables.

4 *The CVT - Architecture*

The code-validation package offers a fully automatic routine 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 can be seen in Figure 2. We will not focus in this (short) paper on the underlying theory behind the verification condition. As mentioned before, a detailed explanation of this can be found in [5]. Rather we explain what is the role of each module and what are its inputs and outputs.

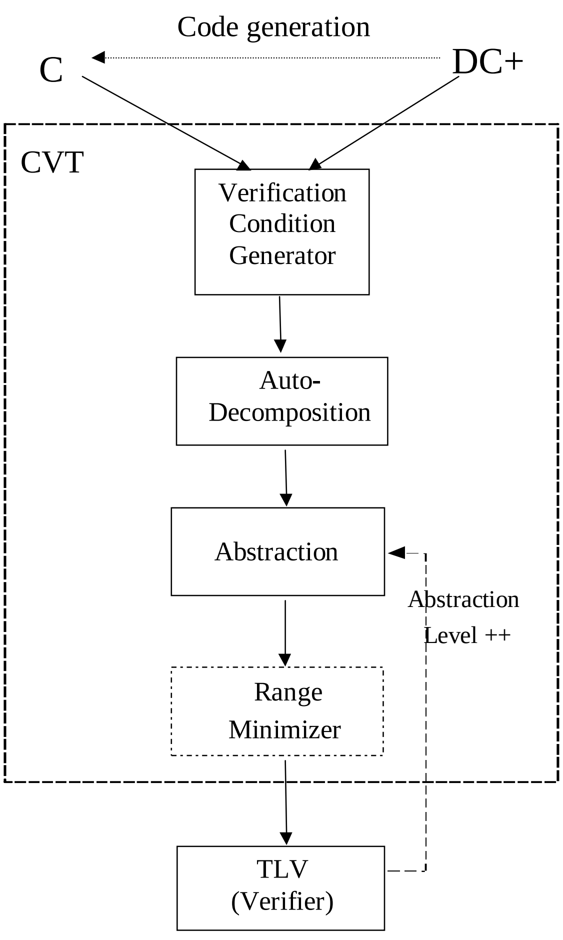

Fig. 2 : CVT Architecture (The 'Range Minimizer' module is not yet part of CVT)

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 (appearing united in Figure 1 as the Verification Condition Generator module) generate the verification conditions (which are 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_A) . Since at the end of this process we use TLV [6] as a decision procedure, the verification condition is generated in the appropriate format (the models TLV expects are compatible with the more broadly used SMV model-checker).

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 premises are in the form of a conjunction (typically of hundreds of expressions), and simply breaks it into smaller conjuncts which can be verified independently.

The size of the decomposed conjuncts is 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*, 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. 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 option in the Auto-Decomposition module is called 'back calculation'. 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 of the model, i.e. the left hand side. Thus if we use the same model 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 back calculation. The efficiency of the back calculation obviously depends on the ordering of conjuncts we are investigating. An optimal ordering would be such that if $\texttt{cone}(C_i) \subseteq \texttt{cone}\ (C_j)$ then C_i and C_j are verified together (with simple sequential ordering this will happen only if C_j appears first or if $\texttt{cone}(C_i) = \texttt{cone}\ (C_j)$). 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 suspect that the overhead of this calculation will be larger than the saving resulting from the better ordering.

4.3 *The Abstraction Module and the Range-Minimizer module*

After decomposing the files, CVT invokes the Abstraction module. Once again, the underlying theory of the abstraction is detailed in [5]. Basically, abstraction is needed since we are trying to verify a model 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 module 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 uninterpreted. If the proof fails, CVT invokes the abstraction module again, asking for *Level-1 abstraction*, where additionally comparisons operators on integers ('>', '<', etc) are left 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 where as level-1 will succeed.

The reason we first interpreted 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 comutativity 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 examples we have considered.

After we replace the appropriate terms by new variables, we impose additional constraints on the verification conditions to ensure functionality. 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 iff it is valid over all interpretations into these domains. The latter can be checked algorithmically, using BDD techniques. The domain that is currently taken is simply a finite set of integers whose size is the number of (originally) integer/float variables, although smaller domains can be achieved by analyzing the structure of the formula considered. This analysis is performed by the 'Range Minimizer' module, not yet implemented, which we expect to significantly reduce the range of each of these (now enumerated type) variables, and thus increase the size of programs we can handle. Preliminary results from this approach show that most models can be verified by using a state-space which is orders of magnitude smaller than the state-space resulting from using our current method.

4.4 The Verifier module (TLV)

The validity of the verification conditions is checked in TLV [6], 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-Composition 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

Currently we are working on the validation of 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 subsection 4.3, the abstraction module replaces function symbols with new variables). Following is a summary of the results achieved so far:

Module	Conjuncts	Verified	Time (min.)
M1	530	100%	4:14
M2	533	100%	1:30
M3	124	92%	?
M4	308	99.3%	3:32 + ?
M5	860	80%	?
Total :	**2355**	**93.9%**	**9:16 + ?**

As can be seen, about 6.1% of the conjuncts in our case could not be verified in reasonable time using the current implementation of CVT. We hope that after installing the Range-Minimizer this problem will be solved.

References

[1] 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.

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

[3] P. Curzon. A verified compiler for a structured assembly language. In *proceedings of the 1991 international workshop on the HOL theorem Proving System and its applications*. IEEE Computer Society Press, 1992.

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

[5] A. Pnueli, O. Shtrichman and M. Siegel. Translation Validation for Synchronous Languages. To appear in *ICALP' 98*.

[6] A. Pnueli and E. Shahar, A Platform for Combining Deductive with Algorithmic Verification. *8th Conference on Computer Aided Verification*, Springer-Verlag, 1996

[7] The SACRES web page: http://www.ilogix.co.uk/ilogix/technica.html