STTT, 1998.

The Code Validation Tool (CVT) – Automatic Verification of Code Generated from Synchronous Languages

A. Pnueli,
O. Shtrichman,
M. Siegel The Weizman Institute of Science

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

2010.11.23

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.

Contents

1. Introduction

- 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.

1. Introduction

- The highly desirable alternative : 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, 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.

1. Introduction

- 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.
- 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.

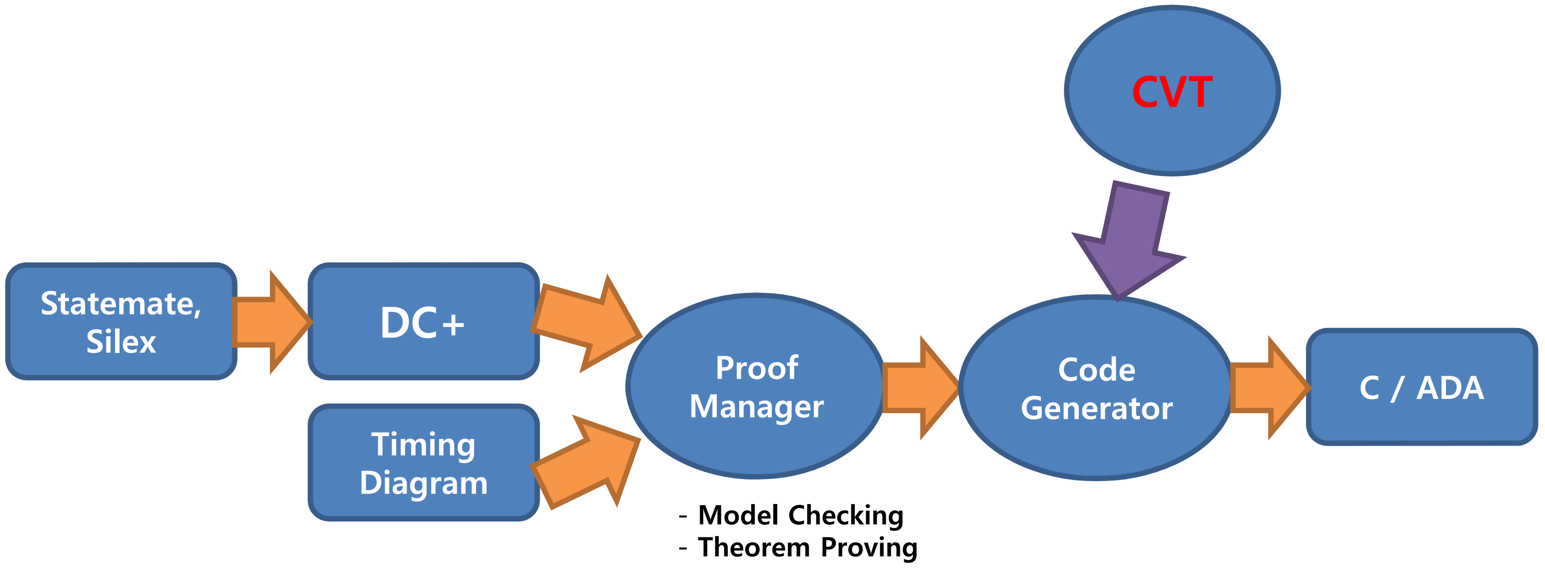
2. Code Validation in the Context of the SACRES Project

- 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 emphasis on formal development of systems, providing
  - formal specification,
  - model checking technology and
  - validated code-generation.

2. Code Validation in the Context of the SACRES Project

- 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.



2. Code Validation in the Context of the SACRES Project

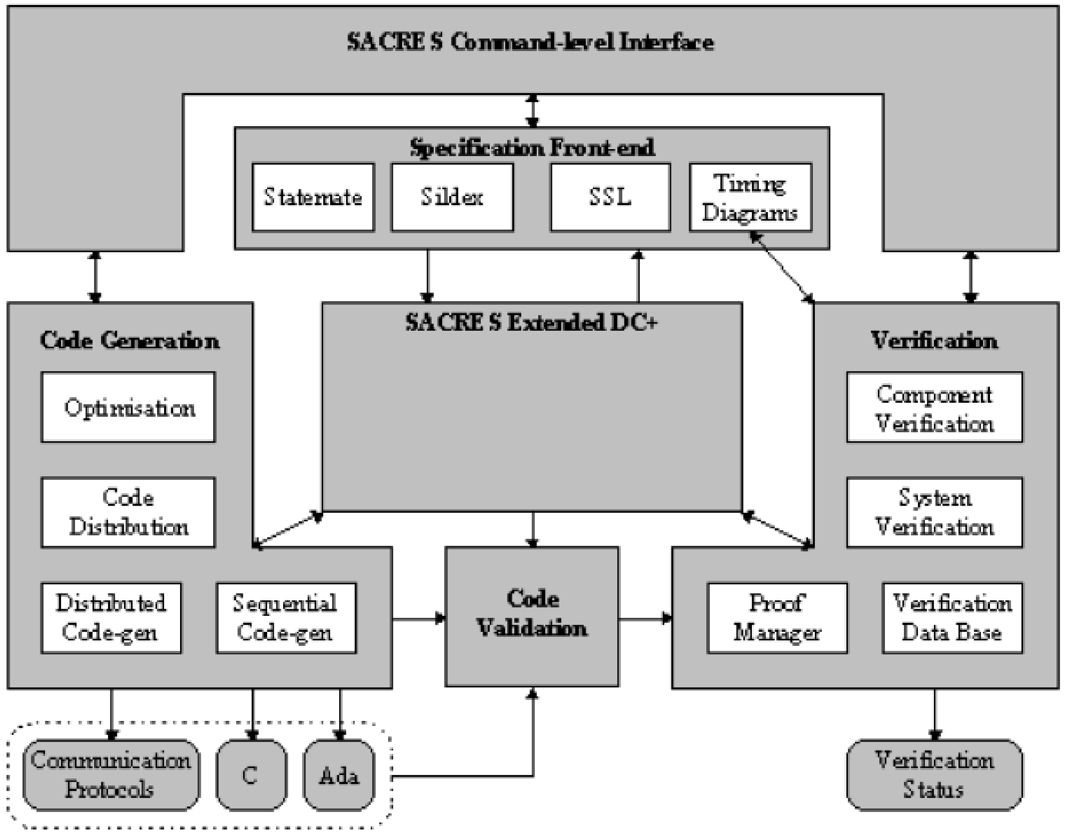


Fig. 1: SACRES Global Architecture

3. The Verification Condition

- This section is a brief description of the structure of the verification condition, which, if proven correct, guarantees the correctness of the translation.

- DC+ program : abstract system
- C program : concrete system
  - V_A , V_C : variables
  - θ_A , θ_C : initial conditions
  - ρ_A , ρ_C : transition relations

- We use two premises (verification conditions)
  - $R1$: the base case
  - $R2$: the induction step

3. The Verification Condition

- Appropriate substitution α
  - 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 $\alpha: V_A \rightarrow \varepsilon(V_c)$ 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

3. The Verification Condition

- 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 each sub-formula corresponds to an assignment line in the code or a constraint imposed by the abstraction.
  - 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.

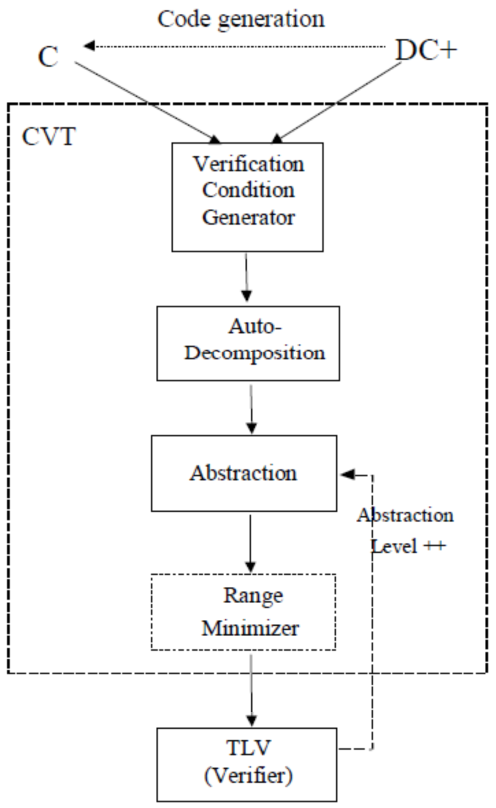


Fig. 2 : CVT Architecture (The 'Range Minimizer' module is not yet part of CVT)

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 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.
- Since at the end of this process we use TLV [6] as a decision procedure, the verification condition is generated in the appropriate format.

4.2 The Auto-Decomposition Module

- 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.
  - Chunk size
  - Back calculation

4.3 The Abstract Module and the Range-Minimizer Module

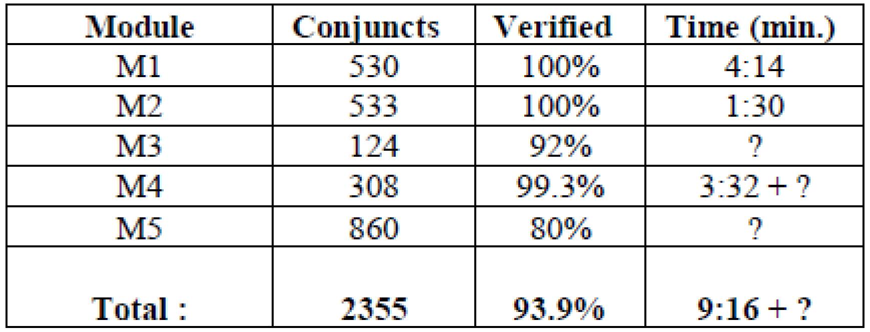
- After decomposing the files, CVT invokes the Abstraction module.
- Abstraction is needed since we are trying to verify a model which contains integer and float variables, as well as functions over these variables using a BDD-Based decision procedure for finite-state models.
- The abstraction module 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.
  - The less we interpret, the smaller the model is.

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
  - Has been developed mainly for finite-state deductive proofs
  - and thus convenient in our case for expressing the refinement rule.

- 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



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.