Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools (abstract)

Van-Chan Ngo  Jean-Pierre Talpin  Thierry Gautier  Loïc Besnard  Paul Le Guernic
INRIA Rennes, France
first.lastname@irisa.fr

ABSTRACT

The aim of this presentation is to demonstrate a scalable, modular, refinance methodology to design, assess and improve the trustability of an existing (20 years old), large (500k lines of C), open source (Eclipse/Polarsys IWG project POP) code generation suite using off-the-shelf, open-source, SAT/SMT verification tools (Yices), by adapting and optimizing the translation validation principle introduced by Pnueli et al. in 1998. This methodology results from the ANR project VERISYNC, in which we aimed at revisiting Pnueli’s seminal work on translation validation using off-the-shelf, up-to-date, verification technology. In face of the enormous task at hand, the verification of a compiler infrastructure comprising around 500 000 lines of C code, we devised to narrow down and isolate the problem to the very data-structures manipulated by the infrastructure at the successive steps of code generation, in order to both optimize the whole verification process and make the implementation of a working prototype at all doable. Our presentation outlines the successive steps of this endeavour, from clock synthesis, static scheduling to target code production.

1. INTRODUCTION

Synchronous programming languages like SIGNAL, LUSTRE and ESTEREL propose a formal semantic framework to give high-level specification of safety-critical software in automotive and avionics systems [2, 14, 15, 17]. Safety-critical systems are those systems whose failure could result in loss of life, or damage to the environment. They need to be validated to ensure that their specified safety properties are implemented correctly. Software validation is traditionally done by using testing techniques which, in the case of safety-critical systems, is not sufficient [19]. Since synchronous languages are based on formal semantic models, they provide much higher level of abstraction, expressivity, and clarity at source level rather than once compiled into C code. That makes the application of formal methods much simpler to enforce safety properties. However, a synchronous compiler is still a large and complex program which often consists of hundreds of thousands lines of code, divided into numerous packages. Moreover, compiler modules often interact in sophisticated ways, and the design and implementation of a compiler are substantial engineering tasks. The compilation process involves many analyzers, program transformations and optimizations, some may introduce additional information or constrain the compiled program, some may refine its meaning and specialize its behavior to meet a specific safety or optimization goal, and all these compile-time decisions should additionally be traced.

Formal compiler verification consists of specifying the behavior of the compiler in a formal specification language and build a proof that the compiler satisfies behavioral equivalence or refinement. Formal verification can be done through many approaches. One such approach is deductive verification. It consists of providing deductive proofs that a system behaves in a certain way that is described in the specification, with the aid of either interactive theorem provers (such as HOL [13], Isabelle [16], or Coq [9]), or an automated theorem prover. Another approach is model checking [6, 29]. It involves building an abstract model of the system and ensure it complies with specified requirements by exploring all its accessible states. Requirements are represented in temporal logics, such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) and verification produces a confirmation that the system model conforms to requirements or a counterexample that can be used to locate and eliminate an error. Some techniques can be used to deal with the state explosion problem including abstract interpretation, symbolic simulation and abstract refinement [10]. A variant of model checking, Bounded model checking (Bmc) [5], encodes the fact that potential executions of the system model do not conform to the specification in incremental fashion as propositional satisfiability formulas. The bounded number of evaluation steps is increased as long as the resulting propositional formula is satisfiable. Then a concrete counterexample can be extracted as a trace of system states leading to an error state in the system model. Inductive reasoning can also be used to prove that a system conforms its specification. Advances in Satisfiability Modulo Theories (Smt) have been useful for checking systems inductively. With SMT solvers, systems can be modelled efficiently, require fewer limitations to represent specifications and meet significant performances.
Translation validation was introduced by Pnueli et al. in [27] as an approach to verify the correctness of translators (compilers, code generators). The main idea of translation validation is that instead of proving the correctness of a compiler, each of its individual translations (e.g., run of the code generator) is followed by a validation phase to check that the target program correctly implements the source specification. A translation validator consists of two objects:

- The Model builder is a simple module that formally represents the semantics of the source and target programs of the translator (e.g., labeled transition system, first-order logic formula).
- The Analyzer formalizes correctness as a refinement relation between the models of the source and target of the validator. The analyzer provides an automated proof method on the existence of the refinement between the formal models. If the analyzer succeeds, a proof script is created. If it doesn’t, it generates a counterexample, which can be decompiled to help spot the error. Translation validation does not modify or instrument the compiler. It treats it as a “black box” (as long as there is no error in it). It only considers the input program and its compiled result. Hence, it is not affected by updates and modifications to the compiler, as long as its data-structures remain the same. In general, the validator is much simpler and smaller than the compiler itself. Thus, the proof of correctness of the validator takes less effort than that of the compiler. Plus, verification is fully automated and scales to large programs.

3. SCALABLE TRANSLATION VALIDATION

Our approach is to scale translation validation not only in a modular fashion, by decomposing the problem into the successive transformations performed by the compiler on the intermediate representation of a program [22], but by narrowing it further to the actual data-structure that are being used to represent the transformation problem and the actual algorithmic operations performed on it. In all cases, we show that translation validation is amenable to simple SAT/SMT verification (the semantic inclusion of one data-structure into another) instead of the more general problem of simulation-based conformance-checking of the transformed program w.r.t. the input specification [23–25]. In the case of the synchronous data-flow language Signal, a 500k-lines big code generation infrastructure, the compilation process can be divided into three phases depicted in Figure 1 (top row). A source Signal program is the synchronous composition of discrete equations on signals, e.g., $x := y + 1 \mid y = x \land 1$ defines $x$ by $y + 1$ at all times and $y$ by the value of $x$ delayed by 1 evaluation tick: its previous value. Its compilation may be seen as a sequence of morphisms that refine and rewrite the source specification with information gained from analysis. C or Java code production is performed on ultimately transformed program, e.g., $y = x; x := y + 1$.

- **Clock calculation.** This stage determines the clock of all signals in the program and defines a Boolean abstraction of the program. The clock of a signal defines when the value of the signal shall be evaluated.
- **Static scheduling.** Based on the clock information and the Boolean abstraction obtained at the first stage, the compiler constructs a Conditional Dependency Graph (Cdg), which represents the schedule of signals’ evaluations.
- **Code generation.** Sequential C or Java code is directly generated from the structure of the clocked and scheduled Signal program.

A proof of semantic-preservation can be decomposed into the preservation of clock semantics at the clock calculation phase and that of data dependencies at the static scheduling phase. Value-equivalence of variables can be then checked at the code generation phase. Figure 1 shows the integration of this verification framework into the compilation process of the Signal compiler. For each phase, the validator takes the source program and its compiled counterpart, and constructs the corresponding formal model of the program. Then, it checks the existence of the refinement relation to prove semantic-preservation.

3.1 Preservation of clock models [24]

The first verification stage focuses on proving that all clock relations with signals in the source and transformed program are equivalent. A clock model is a first-order logic formula that
characterizes the presence/absence status of all signals at all times in a Signal program. Given two clock models, a clock refinement relation is defined to express the semantic preservation of clock semantics. The existence of a clock refinement is defined as a satisfiability problem which is automatically and efficiently proved by an Smt solver.

**Example**  Consider the Signal program DEC which counts through the output N from the value of input FB to 1, ZN := N$\|$ init 1 defined ZN as the previous value of N, N := FB default (ZN-1) assigns FB to N when the input FB is present, and ZN-1 otherwise, FB $\approx$ when (ZN<=1) synchronizes FB to the condition (ZN=1). The clock model of the source program is:

$$\Phi(DEC) = (\bar{FB} \iff \bar{ZN}_1 \land \bar{ZN}_2)$$

$$\land (\bar{ZN}_1 \iff v^1_{<z} \iff \bar{ZN}) \land (\bar{ZN}_2 \iff v^1_{<z} = v^1_{<z})$$

Terms $\bar{a}$, resp. $\bar{b}$, represent the clock resp. value of a signal $x$. Checking it a correct refinement amounts to establish a variable mapping $X_{DEC} = \alpha(X_{DEC} \cdot X_{DDG})$ between DEC and DEC' and delegate the checking validity of the formula $\Phi(DEC') \land X_{DEC} = \alpha(X_{DEC_{BASIC_TCP}} \cdot X_{DDG}) \iff \Phi(DEC)$, named $\Phi$, to the Smt solver under the logical context defined by the variable mapping and the following assertions:

$$(\bar{ZN} = \bar{ZN}^2 \land 1 = 1) \iff (v^1_{<z} = v^1_{<z}) \land (v^1_{<z} = v^1_{<z})$$

### 3.3 Preservation of data dependency [25]

The goal of this stage is to prove that the existence of a data dependency between two signals in the source specification is a property preserved by the target program which, in addition, makes a sequential schedule of computations explicit. Along the way, the validator further checks the target program deadlock-free. Data dependencies among signals are represented by a Synchronous Data-flow Dependency Graph (SDDG). An SDDG is a labeled directed graph in which node are signals and clocks and edges represent dependencies between nodes. Edge are labeled by clocks. An edge clock tells when the dependency is effective: when its clock is present. The correctness of a schedule is formalized as a dependency refinement relation between the source and target SDDGs. It is implemented by Smt-checking the existence of the refinement relation.

**Example**  The SDDGs of DEC and the scheduled DEC' are the following:

Checking formula construction establishes the variable mapping much like for clock models. Next, it amounts to finding all cycles and dependency paths from FB to N to generate the formulas for checking the dependency refinement, the validity of which is delegated to the Yices solver:

$$\bar{FB} \Rightarrow (\bar{FB}^1 \land \bar{FB}^1)$$

### 3.4 Value-equivalence of variables [23]

This stage focuses on proving that every output signal in the source program and the corresponding variable in the generated C program are assigned the same values at all times. The defining equation of a signals and its C translation are represented by a shared value-graphs, called Synchronous Data-flow Value-Graph (Svvg). To prove that a signal and a variable have the same value with an Svvg, we just need to check that they are represented by the same sub-graph, meaning that they point to the same graph node.

**Example**  The code generated from DEC consists of the following step function (left). The value graphs of DEC and DEC_step are constructed and composed (right).

```c
logical DEC_step() {
    if (C_FB)
        return TRUE;
    else
        DEC_step_finalize();
        return FALSE;
}
```

The shared Svvg are then normalized by employing a series of rewriting rules to merge nodes referring to the same value. This yields the following graph, proving that the generated step function conforms its specification:

$$\{N, N, N, \phi \}
\{C_FB \land \phi\}
\{N, N, N, \phi \}
\{C_FB \land \phi\}$$

### 4. RELATED WORKS

The notion of translation validation was introduced in [27, 28] by Pnueli et al., using a symbolic model of Synchronous Transition Systems (S) to represent both source and target programs. An S is a set of logic formulae which describes the functional and temporal constraints of the whole program and its generated C code. Bao [4] implement the symbolic S models. Our approach improves over standard translation validation by explicitly capturing the clock semantics in the model, which also results in much smaller models. Another related work is the static analysis of Signal programs for efficient code generation [12], where linear relations among clocks and values are represented by first-order logic formulas with the help of numerical interval abstraction techniques. The objective is to make generated code more efficient by detecting and removing the dead-code segments (e.g., segment of code to compute a data-flow which is always absent). They determine the existence of empty clocks, mutual exclusion of two or more clocks, or clock inclusions, by reasoning on the formal model using an Smt solver. Related works have also adopted the translation validation approach in verification of transformations, and optimizations. In [21], the translation validation is used to verify several common optimizations such as common subexpression elimination, register allocation, and loop inversion. The validator checks the existence of a simulation relation between two programs. Leroy et al. [7, 18] used this technique to develop the ComCort high-assurance C compiler. The programs before and after the transformations and optimizations of the compiler are represented in a common intermediate form, then the preservation of semantics is checked by using symbolic execution in the proof assistant Coq. Tristan et al. [32] recently
We have presented a technique based on S value graphs) considerably reduces the size of the refinement or value-graph. The graph is Eclipse project POP with the Polarsys Industry Working Group [26]. A compiler. The individual modules designed in the context of this libraries [11] to achieve the expected goals: traceability and formal problem to solve. For that purpose, we represent the desired program e and its transformed result. Hence, it is not a not modify or instrument the compiler. It treats it as a "black box" on a reduced graph to check value equivalence. Our validator does source specification and the generated code to a couple of SMT SAT-code patterns from and a specification's syntax tree. This reduces / transformers performed by the compiler using the simplest structures to represent them: SAT/SMT formulas represent the refinement of clock models and the reinforcement of data-flow graphs, value graphs are used to represent the production of target code patterns from and a specification’s syntax tree. This reduces the whole process of proving a refinement relation between the source specification and the generated code to a couple of SMT SAT-checking on formulas of minimal size, and to a symbolic rewriting on a reduced graph to check value equivalence. Our validator does not modify or instrument the compiler. It treats it as a “black box” (as long as there is no error in it). It only considers an input program and its transformed result. Hence, it is not affected by an update an modification made to this or that compilation stage, as long as its principle and data-structure remains the same. The validator is much simpler and smaller than the compiler itself. Proving its correctness (the model builder, the verifier) would take a lot less effort than for the compiler as well. Verification is fully automated and scales to large programs very well by employing state-of-the-art verification tools and by minimizing the representation of the problem to solve. For that purpose, we represent the desired program semantics using a scalable abstraction and we use efficient Srr libraries [11] to achieve the expected goals: traceability and formal evidence. We believe that this approach provides a, both technically and economically, attractive alternative to developing a certified compiler. The individual modules designed in the context of this project are being integrated in the open-source environment of the Eclipse project POP with the Polarsys Industry Working Group [26].

5. CONCLUSION

We have presented a technique based on Srr solving to prove the preservation of clock semantics during the compilation of a synchronous data-flow compiler. Our approach focuses on the transformations performed by the compiler using the simplest structures to represent them: SAT/SMT formulas represent the refinement of clock models and the reinforcement of data-flow graphs, value graphs are used to represent the production of target code patterns from and a specification’s syntax tree. This reduces the whole process of proving a refinement relation between the source specification and the generated code to a couple of SMT SAT-checking on formulas of minimal size, and to a symbolic rewriting on a reduced graph to check value equivalence. Our validator does not modify or instrument the compiler. It treats it as a “black box” (as long as there is no error in it). It only considers an input program and its transformed result. Hence, it is not affected by an update an modification made to this or that compilation stage, as long as its principle and data-structure remains the same. The validator is much simpler and smaller than the compiler itself. Proving its correctness (the model builder, the verifier) would take a lot less effort than for the compiler as well. Verification is fully automated and scales to large programs very well by employing state-of-the-art verification tools and by minimizing the representation of the problem to solve. For that purpose, we represent the desired program semantics using a scalable abstraction and we use efficient Srr libraries [11] to achieve the expected goals: traceability and formal evidence. We believe that this approach provides a, both technically and economically, attractive alternative to developing a certified compiler. The individual modules designed in the context of this project are being integrated in the open-source environment of the Eclipse project POP with the Polarsys Industry Working Group [26].

REFERENCES