Certified Simulation: What You Get Is What You Want

Vania Joloboff[1], Jean-François Monin[2], and Xiaomu Shi[3]

[1] INRIA - LIAMA
[2] Université de Grenoble - Verimag
[3] Tsinghua University

Abstract. This paper presents an approach to construct a certified virtual prototyping framework of embedded software. The machine code executed on a target architecture can be proven to provide the same results as the initial algorithm, and the proof is verified with an automated theorem prover. This requires the establishment of a tool chain in which each step can be certified. The paper presents the proof of an ARM architecture Instruction Set Simulator, with all of the proofs being verified with the Coq theorem prover.

1 Introduction

In many applications, it is desirable to exhibit a proof of program to certify that the execution of a given algorithm is correct on the specific computer architecture on which the program is executed. For example in the area of distributed algorithms it is interesting to be certain that the implementation on real devices indeed behave the same as the abstract description of the algorithm.

Obtaining such certification is not a one step proof, it consists in a series of steps requiring a tool chain. In this paper, we present an attempt towards building such a certification chain, including a certified simulator of processor architecture. Our approach towards this certification of programs is the following: given the formal specification of an algorithm, one can derive a C program using formal methods such as APTS [18] or event-B [1] or BIP [4]. In this paper we assume that this phase has been achieved, that there is an existing C source code program that is a provably correct implementation of the specification. However, well-known issues concern the specification itself: typically, does it include all desirable features? A number of improvements of the specification can be achieved by reasoning on it: one may state expected conjectures and then try to prove (or disprove) them. This approach is especially relevant for abstract properties, such as the algebraic properties of arithmetics functions. Still, some features are more naturally expressed on the implementation, because they involve interactions with the environment. Alternatively, a piece of software may be well-behaved in some environment, but not so well when the latter changes a little bit. Specifying the precise conditions which are needed or checking that the real environment fits the expectations is not that easy. In all of these situations,

it is important to work with the implementation, not only with the specification. Instead of considering an implementation on real hardware, a *simulated* implementation is more convenient and less expensive. Then, it is important to use a very faithful simulator in order to ensure that no bias is introduced.

Our purpose is to show how it can be certified that the execution of a binary program on the Instruction Set Simulator of a target architecture indeed produces the expected results. This requires sequential steps, to prove first that the translation from C code to machine code is correct, and second that the simulation of the machine code is also correct, that is, they all preserve the semantics of the source code; together with the fact that all of these proofs are verified using a theorem prover or proof checker, not subject to human error in the proof elaboration or verification.

The technique presented here relies on already existing tools, in particular the Coq proof assistant, the Compcert C compiler, a certified compiler for the C language, combined with our own work to prove the correctness of an ARM Instruction Set Simulator, integrated within an existing virtual prototyping framework.

The sequel of the paper presents the background existing tools, related work, our method to combine these tools and our additional work to reach the objectives.

2 Related Work

Program certification has to be based on a formal model of the program under study. Such a formal model is itself derived from a formal semantics of the programming language.

Axiomatic semantics and Hoare logic have been widely used for proving the correctness of programs. For imperative programming languages such as C, a possible approach is to consider tools based on axiomatic semantics, like `Frama-C` [9], a framework for a set of interoperable program analyzers for C. Most of the modules integrated inside rely on ACSL (ANSI/ISO C Specification Language), a specification language based on an axiomatic semantics for C. ACSL is powerful enough to express axiomatizations directly at the level of the C program. State labels can be defined to denote a program control point, and can then be used in logic functions and predicates.

`Frama-C` software leverages off from `Why` technology [7, 14], a platform for deductive program verification, which is an implementation of Dijkstra's calculus of weakest preconditions. `Why` compiles annotated C code into an intermediate language. The result is given as input to the VC (Verification Conditions) generator, which produces formulas to be sent to both automatic provers or interactive provers like Coq.

In our case of verifying an instruction set, we have to deal with a very large specification including complex features of the C language. A framework is required that is rich enough to make the specification manageable, using abstraction mechanisms for instance, and in which an accurate definition of C features

is available. Automated computations of weakest preconditions and range of variation are not relevant in our case. We need to verify more specific properties referring to a formal version of the ARM architecture. It was unclear that `Frama-C` would satisfy those requirements, even with Coq as a back-end.

Operational semantics is a more concrete approach to program semantics as it is based on states, although in contrast with a low-level implementation, operational semantics considers abstract states. The behavior of a piece of program corresponds to a transition between abstract states. This transition relation makes it possible to define the execution of programs by a mathematical computation *relation*. This approach is quite convenient for proving the correctness of compilers, using operational semantics for the source and target languages (and, possibly intermediate languages). Operational semantics are used in `CompCert` to define the execution of C programs, or more precisely programs in the subset of C considered by the `CompCert` project. The work presented in this paper is based on this approach. Interesting examples are given by Brian Campbell in the CerCo project [8], in order to show that the evaluation order constraints in C are very lax and not uniform.

Regarding formalization and proofs related to instruction set, a Java byte code verifier has been proved by Cornelia Pusch[22], the Power architecture semantics has been formally specified in [2], and closer to our work, the computer science laboratory in Cambridge University has used HOL4 to formalize the instruction set architecture of ARM [15]. The objective of their work was to verify an implementation of the ARM architecture with *logical gates*, whereas we consider a ARM architecture simulator coded in C. Reusing the work done at Cambridge was considered for our work. However, as our approach is based on `CompCert`, which is itself coded in Coq instead of HOL4, it was more convenient to develop our formal model and our proofs in Coq.

There is abundant literature covering Instruction Set Simulation (ISS). Using *interpretive simulation*, such as used in Insulin [24], each instruction of the target program is fetched from memory, decoded, and executed. With *static translation*, the target application program is decoded at compile time and translated into a new program for the simulation host. The simulation speed is vastly improved [26, 10], but it is not suitable for application programs that dynamically modify the code, or dynamically load code at run-time. Most ISS'es today use some kind of *dynamic binary translation*, initiated with systems such as Shade [11] and Embra [25], and later enhanced with various types of optimization techniques [5].

Our work is based on the SimSoC simulation framework [16], which uses dynamic binary translation, and is available as open source software at `http://gforge.inria.fr/projects/simsoc`.

3 Background

3.1 Coq

Coq [6] is an interactive theorem prover, implemented in OCaml. It allows the expression of mathematical assertions, mechanically checks proofs of these as-

sertions, helps to discover formal proofs, and may extract a certified program from the constructive proof of its formal specification. Coq can also be presented as a dependently typed λ-calculus (or functional language). For a detailed presentation, the reader can consult [12] or [6]. Coq proofs are typed functions and checking the correctness of a proof boils down to type-checking.

Coq is not an automated theorem prover: the logic supported by Coq includes arithmetic; therefore it is too rich to be decidable. However, type-checking (in particular, checking the correctness of a proof) is decidable. As full automation is not possible for finding proofs, human interaction is essential. The latter is realized by *scripts*, which are sequences of commands for building a proof step by step. Coq also provides built-in *tactics* implementing various decision procedures for suitable fragments of the calculus of inductive constructions and a language which can be used for automating the search of proofs and shortening scripts.

3.2 Compert-C

`CompCert` is a formally verified compiler for the C programming language provided by INRIA [20, 19], which currently targets Power, ARM and 32-bit x86 architectures. The compiler is specified, programmed, and proved in Coq. It aims to be used for programming embedded systems requiring high reliability. The performance of its generated code is often close to that of gcc (version 3) at optimization level O1. The generated assembly code is proved to behave exactly the same as the input C program, according to a formally defined operational semantics of the language provided the input does not contain ill-defined expressions according to the ISO-C specification (e.g., a[i++] = i;). This assumption is expecteed to hold if the C code has been generated using formal specifications.

Three parts of `CompCert` C are used in this work. The first is that we use the correct machine code generated by the C compiler. The second is the C language operational semantics in Coq from which we get a formal model of the program. Third, we use the `CompCert` Coq library for words, half-words, bytes etc., and bitwise operations to describe the instruction set model.

3.3 SimSoC

As mentioned above, our certification target ISS is integrated within `SimSoC` [17], a full system simulator of System-on-Chip that can simulate various processors, available as open source software. `SimSoC` takes as input real binary code and executes simulation models of the complete embedded system: processor, memory units, interconnect, and peripherals. The chip simulator includes a network controller simulator (from Texas Instruments) and this network controller simulator is plugged to the real network through an interface, using the libpcap library, so the simulator can communicate with the real world.

SimSoc is developed in SystemC [21] and uses transaction level modeling (TLM) to model communications between the simulation modules. It includes Instruction Set Simulators (ISS) to execute embedded applications on simulated platforms. Our work introduces a new ISS for the ARM architecture.

4 Certified Simulation

Our general objective to obtain a certified simulatoris illustrated in Figure 1. Considering the ARM architecture (Version 6), we need to have the following:

- a simulator of the ARM instruction set in (`CompCert`) C that we can obtain from the ARM Reference manual.
- a formal operational semantics of the ISS. Given some source code in C, one can obtain through `CompCert` the verified machine code, or alternatively the Coq formal semantics of the compiled program constructed by `CompCert`.
- prove, using a proof assistant, that the resulting ISS semantics indeed implement an ARM processor, to verify that the semantics of the simulator accurately modifies the processor state at each step. For that, we need to prove that the results are compliant with a formal model of the ARM architecture.

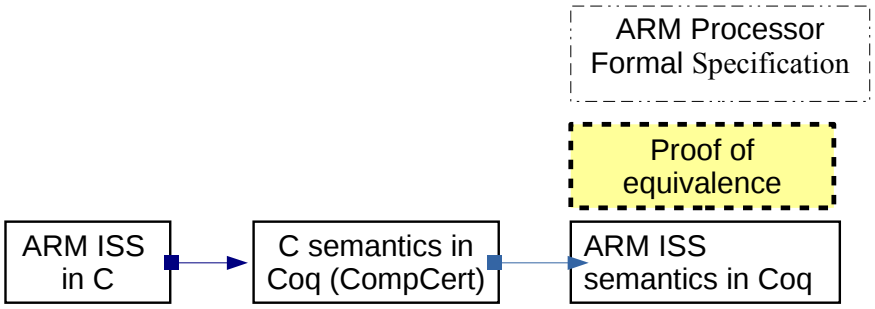

Fig. 1. Overall goal

These steps are described in the following paragraphs.

4.1 Constructing the ISS

The whole process starts with the ARM reference manual available from ARM web site (version 6) `ARM DDI 0100I` [3]. Here are the main relevant chapters.

- `Programmer's Model` introduces the main features in ARMv6 architecture, the data types, registers, exceptions, etc;
- `The ARM Instruction Set` explains the instruction encoding in general and puts the instructions in categories;
- `ARM Instructions` lists all the ARM instructions in ARMv6 architecture in alphabetical order and the `ARM Addressing modes` section explains the five kinds of addressing modes.

There are 147 ARM instructions in the ARM V6 architecture. For each instruction, the manual provides its encoding table, its syntax, a piece of pseudo-code explaining its own operation, its exceptions, usage, and notes. Except the semi-formal pseudo-code, everything else is written in natural language. Three files are extracted from the reference manual: a 2100 lines file containing the pseudo-code, a 800 lines file containing the binary encoding tables, and a 500 lines file containing the assembly syntax. Other than these three extracted files, there is still useful information left in the document which cannot be automatically extracted, such as validity constraints information required by the decoder generator. The corresponding information is manually encoded into a 300 lines file. The overall architecture of the generator is given in figure 2.

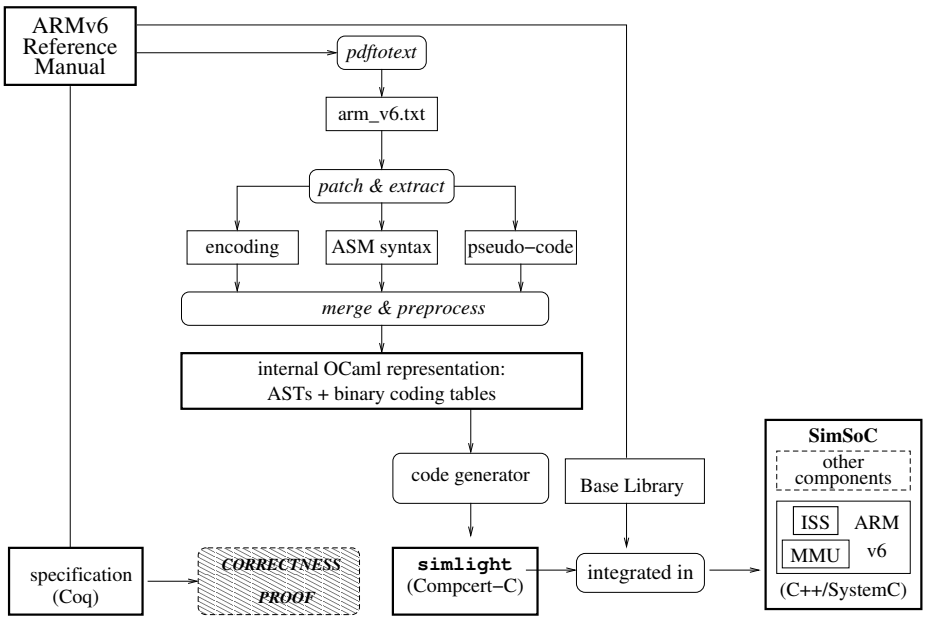

Fig. 2. Overall generation chain

Three kinds of information are extracted for each ARM operation: its binary encoding format, the corresponding assembly syntax, and the instruction semantics, which is an algorithm operating on the various data structures representing the state of an ARM processor, mostly registers and memory, according to the purpose of the instruction considered. This algorithm may call basic functions defined elsewhere in the manual, for which we provide a `CompCert` C library to be used by the simulator and a Coq library defining their semantics. The latter relies on libraries from the `CompCert` project that allows us, for instance, to manipulate 32-bits representations of words. The result is a set of abstract syntax

trees (ASTs) and binary coding tables. These ASTs follow the structure of the (not formally defined) pseudo-code.

In the end, two files are generated: a `CompCert` C file to be linked with other components of `SimSoC` (each instruction can also be executed in stand-alone mode, for test purposes for instance) and files representing each instructions in `CompCert` C abstract syntax to be used for correctness proof.

The ARM V6 code generator not only generates the semantics functions, it also generates the decoder of binary instructions supported in V6 architectures. This decoder is obtained by compiling the opcodes information. The generated decoder is not optimal in performance, but as `SimSoC` uses a cache to store the decoded instructions, the performance penalty is marginal.

4.2 Simulator Semantics

In order to formally reason on the correctness of the simulator, we need to have a formal model of the C implementation of the ARM architecture as described above. It is provided by `CompCert`, which defines operational semantics of C formalized in Coq. As the simulator C program also has the objective to achieve a high speed simulator, the generator makes optimizations. In particular, states in the model of the C implementation are complex, not only due to the inherent complexity of the C language memory model, but also because of optimization and design decisions targeting efficiency. In more detail:

- The C implementation uses large embedded *struct*s to express the ARM processor state. Consequently the model of the state is a complex Coq record type, including not only data fields but also proofs to guaranteed access permission, next block pointer, etc.
- Transitions are defined with a relational style. In general, the relational style is more flexible but functional definitions have some advantages: reasoning steps can be replaced by computations; existence and unicity of the result are automatically ensured. However, the functional style is not always convenient or even possible. It is the case here, where the transitions defined by the C implementation are relations which happen to be functions. This comes first from the operational semantics, which needs to be a relation for the sake of generality. Furthermore in our case, the kind of record type mentioned in the previous item is too complex to execute calculation with it, so it is more convenient to describe the state transformation for memory with a relation.
- The global state is based on a complex memory model with load and store functions that are used for read/write operations.

4.3 Proof

As a result of the generator described above, we have a C implementation of the ARM instruction set. To prove that it is correct we need to have a formal specification of that architecture, and prove that, given the initial state of the system, the execution of an instruction as implemented by a C function results

in the same state as the formal specification. As mentioned in the related work section, we considered using the ARM formal model defined in [15], but it would have required us to translate all of the C operational semantics as well, which would not have been error prone, not to mention the effort. So, we chose to define a formal model of ARM architecture in Coq as well, derived from the architecture reference manual. As Coq models are executable, we could validate the model with real programs.

The complete proof is too lengthy for this article, and we only provide here an outline of the method. The state of the ARM V6 processor defined in the formal model is called the *abstract state*. On the other hand, the same state is represented by the data structures corresponding to C semantics that we shall call the *concrete state*. In order to establish correctness theorems we need to relate these two models. Executing the same instruction on the two sides produces a pair of new processor states which should be related by the same correspondence. Informally, executing the same instruction on a pair of equivalent states should produce a new pair of equivalent states, as schematized by Figure 3.

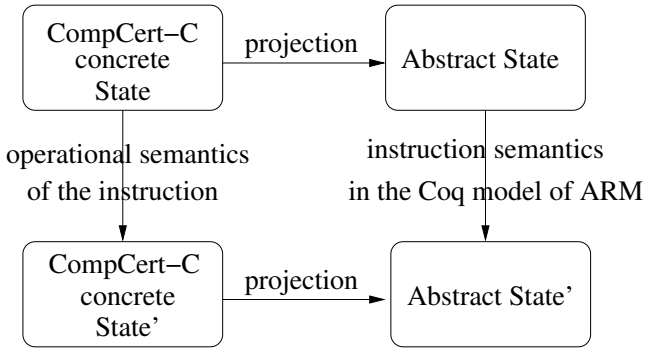

Fig. 3. Theorem statement for a given ARM instruction

Projections from the concrete state to the abstract state are represented in Figure 4. According to the type of the argument of the projection, the definitions of projections are different. For example, the projection of a register performs a case analysis on a value of type `register`, whereas the projection of SPSR (Save Processor Status Register) depends on the type of exception modes. We define a specific projection for each type. The proofs start from the abstract state described by the formal specification. In order to verify the projection of the pair of original states, we need the following data: the initial memory state, the local environment, and the formal initial processor state. The projection is meaningful only after the C memory state is prepared for evaluating the current function body representing a ARM instruction. In the abstract Coq model, we directly use the processor state `st`. But on the C side, the memory state must provide the contents of every parameter, especially the memory representation

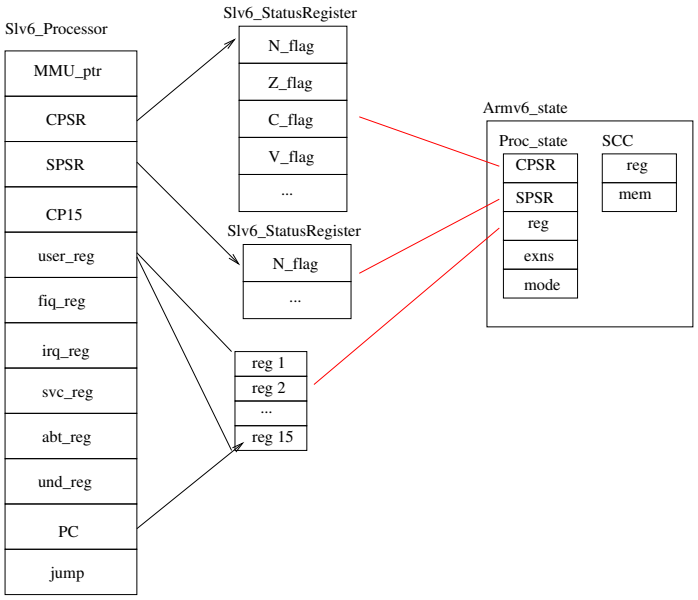

Fig. 4. Projection

of the processor state. We also need to observe the modification of certain blocks of memory corresponding to local variables.

The semantics of `CompCert` C consider two environments. The global environment *genv* maps global function identifiers, global variables identifiers to their blocks in memory, and function pointers to a function definition body. The local environment *env* maps local variables of a function to their memory blocks reference. It maps each variable identifier to its location and its type, and its value is stored in the associated memory block. The value associated to a C variable or a parameter of a C function is obtained by applying `load` to the suitable reference block in memory. These two operations are performed when a function is called, building a local environment and an initialized memory state. When the program starts its execution, *genv* is built. On the other hand, *env* is built when the associated function starts to allocate its variables. Therefore, on the concrete side, a memory state and a local environment is prepared initially using two steps. First, allocate function variables: from an empty local environment, all function parameters and local variables ara allocated into the memory state, yielding a new memory state and the local environment. Second, initialize function parameters: using a function `bind_parameters` to initialize parameters with a list of argument values and a new memory state is created.

Next, we need to consider the execution of the instruction. In the C instruction set simulator, there is a standalone C function for each ARM V6 instruction. Each function (instruction) has its own correctness proof. Every function is composed of its return type, arguments variables, local variables, and the function

body. The function body is a sequence of statements including assignments and expressions. Let us consider as an example the ARM instruction `BL` (`Branch and Link`). The generated C code is:

```
void B(struct SLv6_Processor *proc,
       const bool L,
       const SLv6_Condition cond,
       const uint32_t signed_immed_24){
 if (ConditionPassed(&proc->cpsr, cond)){
  if ((L == 1))
   set_reg(proc,14,address_of_next_instruction(proc));
   set_pc_raw(proc,reg(proc,15)+(SignExtend_30(signed_immed_24)<<2));
 }
}
```

`CompCert` has designed semantics for `CompCert` C in both small-step and big-step. The big-step inductive type for evaluating expression is enough for our proof. The semantics is defined as a relation between an initial expression and an output expression after evaluation. Then the body of the function is executed. On the `CompCert` C side, the execution is yielding a new memory state **mfin**. On the abstract side, the new processor state is obtained by running the formal model. We have to verify that the projection from the concrete state **mfin** is related to this abstract state.

The proof is performed in a top-down manner. It follows the definition of the instruction, analyzing the expression step by step. The function body is split into statements and then into expressions. When evaluating an expression, we search for two kinds of information. the first one is how the memory state changes on `CompCert` C side; the other is whether the results on the abstract and the concrete model are related by the projection. To this effect, we use lemmas in the following five categories.

1. *Evaluating a `CompCert` expression with no modification on the memory state.*
   Such lemmas are concerned with the expression evaluation on `CompCert` C side and in particular the C memory state change issue. Asserting that a memory state is not modified has two aspects: one is that the memory contents are not modified; the other is that the memory access permission is not changed. For example, evaluating the boolean expression $Sbit \ == \ 1$ returns an unchanged memory state.

   $$\begin{array}{l}\text{if } \ G, E \ \vdash \texttt{eval\_binop}_c \ (Sbit \ == \ 1), \ M \ \stackrel{\varepsilon}{\Longrightarrow} \ v, \ M' \\ \text{then } \ M = M'.\end{array}$$

   In Coq syntax, the relation in premise is expressed with `eval_binop`. In this lemma and the following, E is the local environment, G is the global environment and M is the memory state; ε is the empty event; usually here we have a series of system events; v is the result. The evaluation is performed under environments G and E. Before evaluation, we are in memory state M. With no event occurring, we get the next memory state M'. According to the definition of `eval_binop`, an internal memory state will be introduced.

   $$\frac{G, E \ \vdash a_1, M \Rightarrow M' \quad G, E \ \vdash a_2, M' \Rightarrow M''}{G, E \ \vdash (a_1 \ binop \ a_2), M \Rightarrow \ M''}$$

   Now, in our example, expression a_1 is the value of $Sbit$ and a_2 is the constant value 1. By inverting the hypothesis of type `eval_binop`, we obtain several new hypotheses, including on the evaluation of the two subexpressions and the introduction

of an intermediate memory state M''. Evaluating them has no change on the C memory state. Then we have $M = M'' = M'$. In more detail, from the `CompCert` C semantics definition, we know that, evaluation of an expression will change the memory state if the evaluation contains uses of `store_value_of_type` in `CompCert` versions before 1.11. In `CompCert`-1.11, the basic store function on memory is represented by an inductive type `assign_loc` instead of `store_value_of_type`.

2. *Result of the evaluation of an expression with no modification on the memory.* Continuing the example above, we now discuss the result of evaluating the binary operation $Sbit \ == \ 1$ both in the abstract and the concrete model. At the end of evaluation, a boolean value $true$ or $false$ should be returned. in both the `CompCert` C model and the abstract model, using the projection definition.

$$\begin{array}{l}\text{if } \texttt{Sbit\_related}\ M\ \texttt{Sbit},\\ \text{and } G, E\ \vdash \texttt{eval\_rvalue\_binop}_c\ (Sbit\ ==\ 1), M \Rightarrow\ v, M'\\ \text{then } v = (Sbit\ ==\ 1)_{coq}\end{array}$$

Intuitively, if the projection corresponding to the parameter `sbit` in the C program yields the right information from the abstract state, then the evaluation will return the same value both in the abstract and in the concrete model. Here, the expression is a so-called "simple expression" that always terminates in a deterministic way, and preserves the memory state. To evaluate the value of simple expressions, `CompCert` provides two other big-step relations `eval_simple_rvalue` and `eval_simple_lvalue` for evaluating respectively their left and right values. The rules have the following shape:

$$\frac{G, E\ \vdash a_1, M \Rightarrow v_1 \quad G, E\ \vdash a_2, M \Rightarrow v_2 \quad \texttt{sem\_binary\_operation}(op, v_1, v_2, M)\ =\ v}{G, E\ \vdash (a_1\ op\ a_2), M \Rightarrow v}$$

In order to evaluate the binary expression $a_1\ op\ a_2$, the sub-expressions a_1 and a_2 are first evaluated, and their respective results v_1 and v_2 are used to compute the final result v.

3. *Memory state changed by storage operation or side effects of evaluating expression.* As mentioned before, evaluating some expressions such as `eval_assign` can modify the memory state. Then we need lemmas stating that corresponding variables in the abstract and in the concrete model will evolve consistently. For example, considering an assignment on register Rn, the projection relation `register_related` is used. Expressions with side effects of modifying memory are very similar.

$$\begin{array}{l}\text{if } \texttt{rn\_related}\ M\ rn\\ \text{and } G, E\ \vdash \texttt{eval\_assign}_c\ (rn := rx), M\ \Rightarrow\ M', v\\ \text{then } \texttt{rn\_related}\ M'\ rn\end{array}$$

4. *Internal function call.*
Internal functions are described in an informal manner in the ARM V6 reference manual. No pseudo-code is available for them, which means that the corresponding library functions, both in the abstract model and in C, are coded manually. In order to get a suitable `CompCert` C AST to reason about, we use the parser provided in `CompCert`. When combining the simulation code of an instruction with the code of library functions, we need to take care of the memory allocation problem. In `CompCert` C representation, identifiers are unique positive numbers which indicate the memory block where corresponding variables are allocated. Currently, the extra

identifiers introduced by library functions are added manually and assigned with fresh block numbers.

> if `proc_state_related` M st
> and $G, E \vdash$ `eval_funcall`$_c(copy\_StatusRegister)_c, M \Rightarrow\ v,\ M'$
> and $st' = (copy\_StatusRegister)_{coq}\ st$
> then `proc_state_related` M' st'.

After an internal function is called, a new stack of blocks is allocated in memory. After the evaluation of the function is performed, these blocks will be freed. Unfortunately, this may not bring the memory back to the previous state: the memory contents may stay the same, but the pointers and memory organization may have changed. For lemmas on evaluation of internal functions, we can observe the returned result on variables, compare it with the corresponding evaluation in the formal specification, and verify some conditions. For example, the lemma above is about the processor state after evaluating an internal function call `copy_StatusRegister` which reads the value of CPSR and then assigns it to SPSR. The evaluation of `copy_StatusRegister` should be protected by a check on the current processor mode. If it is neither system mode nor user mode, the function `copy_StatusRegister` can be called. Otherwise, the result is "unpredictable", which is defined by ARM architecture
It is then necessary to reason on the newly returned states, which should still be related by the projection. This step is usually easy to prove, by calculation on the two representations of the processor state to verify they match.

5. *External function call.*
The `CompCert` C AST of an external function call contains the types of input arguments and of the returned value, and an empty body. `CompCert` provides the expected properties of a few built-in external functions such as `printf`, `malloc` and `free`. We have proceeded similarly for the external functions of the ARM simulator. The general expected properties of an external call are that (i) the call returns a result, which has to be related to the abstract state, (ii) the arguments must comply with the signature. (iii) after the call, no memory blocks are invalidated, (iv) the call does not increase the access permission of any valid block, and finally that the memory state can be modified only when the access permission of the call is granted. For each external call, we verify such required properties.

In addition to the above lemmas we had to prove a fair number of more trivial lemmas that are omitted here. Most of them are related to the semantics of `CompCert` C. Details are given in [23]. With these lemmas we can build the proof scripts for ARM instructions. For that, we are decomposing the ARM instruction execution step by step to perform the execution of the C programs . `CompCert` C operational semantics define large and complex inductive relations. Each constructor describes the memory state transformation of an expression, statement, or function. As soon as we want to discover the relation between memory states before and after evaluating the C code, we have to invert the hypotheses of operational semantics to follow the clue given by its definition, to verify the hypotheses relating concrete memory states according to the operational semantics.

During the development of a proof, if a hypothesis is an instance of an inductive predicate and we want to derive the consequences of this hypothesis, the general logical principle to be used is called *inversion*. An *inversion* is a kind of forward reasoning step that allows for users to extract all useful information contained in a hypothesis. It is

an analysis over the given hypothesis according to its specific arguments, that removes absurd cases, introduces relevant premises in the environment and performs suitable substitutions in the whole goal. The practical need for automating inversion has been identified many years ago and most proof assistants (Isabelle, Coq, Matita,...) provide an inversion mechanism. To this effect, the Coq proof assistant provides a useful tactic called `inversion` [12].

Every instruction contains complex expressions, but each use of `inversion` will go one step only. If we want to find the relation between the memory states affected by these expressions, we have to invert many times. For illustration, let us consider the simple example from the ARM reference manual `CPSR = SPSR`. As the status register is not implemented by a single value, but a set of individual fields, the corresponding C code is a call to the function `copy_StatusRegister`, which sets the CPSR field by field with the values from SPSR. Lemma `same_cp_SR` below states that the C memory state of the simulator and the corresponding formal representation of ARM processor state evolve consistently during this assignment.

```
Lemma same_copy_SR :
  ∀ e m l b s t m' v em,
  proc_state_related m e (Ok tt (mk_semstate l b s)) →
    eval_expression (Genv.globalenv prog_adc) e m expr_cp_SR t m' v  →
    ∀ l b, proc_state_related m' e
                     (Ok tt (mk_semstate l b (Arm6_State.set_cpsr s
                                              (Arm6_State.spsr s em))))
```

From this, we have to invert generated hypotheses until all constructors used in its type are exhausted. On this example, 18 consecutive inversions are needed... The first proofs scripts we wrote were becoming unmanageable, and not robust to version changes of Coq or `CompCert`. In order to reduce the script size and get better maintainability, we studied a general solution to this problem, and developed a new inversion tactics in Coq . We have first expanded the small inversion mechanism from Coq and added a new inversion tactic for inductive types in `CompCert`. The semantics of `CompCert` C tells us how the memory state is transformed by evaluating expressions. Using the built-in constructs of the tactics language, we can define a high-level tactic for each inductive type, gathering all the functions defined for its constructors.

This tactic has two arguments corresponding to the C memory states. The first step of the tactic introduces generated components with new names. The second step is related to previously reverted hypotheses, ensuring that the new names introduced are correctly managed by Coq. The tactic then proceeds as follows. First, it automatically finds the hypothesis to invert by matching the targeted memory states. Then the related hypotheses are reverted and an appropriate auxiliary function is called (all auxiliary functions are gathered into the tactic) and meaningful names are given to derived variables and hypotheses. Next, all other related hypotheses are updated according to the new names, and finally new values and useless variables or hypotheses are cleaned up. The above steps are then repeated until all transitions between the two targeted memory states are discovered. As a result, considering the former example of `same_copy_SR` where 18 standard `inv` were used in the first proof script we developed, our tactics reduced them into one step: `inv_eval_expr m m'`. Thanks to this new tactics, the size of the proofs has become smaller and the proof scripts are more manageable. The size vary with the instructions complexity from less than 200 lines (e.g 170 for LDRB) to over 1000 (1204 for ADC). As a result, for each ARM instruction, we have established a theorem proving that the C code simulating an ARM instruction is

equivalent to the formal specification of the ARM processor. All of these lemmas and theorems are verified by the Coq theorem prover [23].

5 Conclusion

Using the approach presented in this paper, we can construct a tool chain that makes it possible to certify that the simulation of a program is conformant with the formal definition of the algorithm, by leveraging off from three existing tools namely, Compcert-C, that has defined formal C semantics and a formally proven the code generator, Coq and SimSoc, to which we have added a proven generated simulator of the ARM instruction set. There is no limit on the size of the C code that can be certified as long as the target instruction set has been certified. This provides a very useful complement to formal methods producing C components from formal specifications, for sequential as well as for distributed systems: since we prove the execution of programs over a simulator that is itself a proved implementation of the architecture, it means then that the execution of programs that are themselves generated from formal specifications on that architecture are proved to be correct with regards to their specification.

Broader usages of certified simulation make sense as well. For instance, considering data-centric distributed algorithms written in Netlog and proven correct using the method described in [13], it is interesting to check, using certified simulation, that the implementation of Netlog on chips is correct in order to get a high confidence that the future implementation of the system will behave as expected.

The quality of our "certified simulation" relies on the faithfulness of our formal model of the ARM processor to the reference manual. The latter is unfortunately not formal enough for our needs but we did our best: as Coq specifications are executable, we have been able to validate our ARM formal model by checking that we obtain identical results with real test programs.

In order to certify more complete systems, a further step is to certify additional parts of SimSoC (other than the instruction set simulator) using again the technologies presented here. For instance, for a System on Chip relevant to distributed systems, we could consider the network controller mentioned in 3.3. This is for further study, but given our current achievements it is clear that our approach could cover this part as well.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of power and ARM multiprocessor machine code. In *DAMP'09*, pages 13–24, New York, NY, USA, 2008. ACM.
3. ARM. *ARM Architecture Reference Manual DDI 0100I*. ARM, 2005.
4. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.
5. F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC'05*, pages 41–41, Berkeley, 2005. USENIX Association.

6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
7. F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. *Boogie*, 2011:53–64, 2011.
8. B. Campbell. An executable semantics for Compcert C. In *Certified Programs and Proofs*, pages 60–75. Springer, 2012.
9. G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *SCAM'09*, pages 123–124. IEEE, 2009.
10. M.-K. Chung and C.-M. Kyung. Improvement of Compiled Instruction Set Simulator by Increasing Flexibility and Reducing Compile Time. In *RSP'04*, pages 38–44, Washington, 2004. IEEE.
11. B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS'94*, pages 128–137, New York, May 1994. ACM.
12. Coq Development Team. *The Coq Reference Manual, Version 8.2*. INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.
13. Y. Deng, S. Grumbach, and J.-F. Monin. A Framework for Verifying Data-Centric Protocols. In R. Bruni and J. Dingel, editors, *FMOODS/FORTE*, volume 6722 of *LNCS*, pages 106–120, Reykjavik, Iceland, June 6-9 2011. Springer.
14. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07, LNCS 4590*, 2007.
15. A. C. J. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP*, pages 243–258, 2010.
16. C. Helmstetter and V. Joloboff. SimSoC: A SystemC TLM integrated ISS for full system simulation. In *APCCAS'2008*, pages 1759–1762. IEEE, 2008.
17. C. Helmstetter, V. Joloboff, and H. Xiao. SimSoC: A full system simulation software for embedded systems. In IEEE, editor, *Open-source Software for Scientific Computation (OSSC), 2009 IEEE International Workshop on*, pages 49–55, Sept 2009.
18. E. I. Leonard and C. L. Heitmeyer. Automatic Program Generation from Formal Specifications using APTS. In O. Danvy and al., editors, *Automatic Program Development*, pages 93–113. Springer, 2008.
19. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
20. X. Leroy. *The CompCert C verified compiler. Documentation and user's manual*. INRIA Paris-Rocquencourt, March 2012.
21. Open SystemC Initiative. *SystemC v2.2.0 Language Reference Manual (IEEE Std 1666-2005)*, 2006. `http://www.systemc.org/`.
22. C. Pusch. Proving the Soundness of a Java Bytecode Verifier Specification in Isabelle/HOL. In *TACAS'99*, pages 89–103. Springer, 1999.
23. X. Shi. *Certification of an Instruction Set Simulator*. Phd thesis, Université de Grenoble, July 2013.
24. S. Sutarwala, P. G. Paulin, and Y. Kumar. Insulin: An instruction set simulation environment. In *CHDL '93: 11th IFIP WG10.2 International Conference*, pages 369–376, Amsterdam, 1993. North-Holland.
25. E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *SIGMETRICS'96*, pages 68–79, New York, 1996. ACM.
26. J. Zhu and D. D. Gajski. An ultra-fast instruction set simulator. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):363–373, 2002.