ISTITUTO PER LA RICERCA
SCIENTIFICA E TECNOLOGICA

38050 Povo (Trento), Italy
Tel.: + 39 461 314575 · Fax: + 39 461 314591
e-mail: prdoc@itc.it · url: http://www.itc.it

Certification of Translators via Off-line and On-line Proof Logging and Checking

P. Bertoli, A. Cimatti, F. Giunchiglia, P. Traverso

October 1997

Technical Report # 9710-14

Certification of Translators via Off-line and On-line Proof Logging and Checking

Piergiorgio Bertoli, Alessandro Cimatti, Fausto Giunchiglia, Paolo Traverso
Mechanized Reasoning Group
IRST - Institute for Scientific and Technological Research
38050 - Povo - Trento, Italy
{bertoli,cimatti,fausto,leaf}@irst.itc.it

Abstract

Using non failure-safe components in the implementation of safety-critical systems is desirable because of the extremely high cost of certified components. In order to enhance the safety of such systems, we adopt a solution based on the idea of verifying each single execution of the software running upon them. In particular, we consider the class of translation-based tools used in the development of safety-critical systems. In order perform the verification in an automatic and efficient way, we follow an innovative approach, by distinguishing an off-line and an on-line verification phases. Each proof in the two phases is guaranteed correct thanks to the the use of a logging-and-checking architecture for the tools used to generate them. We describe in detail the off-line and on-line logging-and-checking methodology, its application in the frame of an industrial project, and the ongoing logging-and-checking redesign of a state-of-the-art prover which we intend to use in future applications.

1 Introduction

Safety-critical systems must obey strict correctness constraints, since their failure may imply huge monetary losses, or even put human lives in danger. Nevertheless, implementing them on top of commercial, off-the-shelf hardware/software platforms (COTS), which are not guaranteed to be fail-safe, is desirable because of the extremely high cost of certified components [13]. However, the once-for-all formal verification of the software developed and run on COTS is not sufficient, because of the possible occurrence of platform-induced runtime errors.

The goal of this paper is to propose and describe a solution based on the idea of verifying each single execution of such software rather than the software itself. This approach seems to be applicable to a wide class of tools used in the development of safety-critical systems - thus safety-critical systems themselves. So far we have applied the idea for the certification of a translator from a high-level specification language to binary code [3], and we are in the process of applying it to a system for the generation of telegrams for radio-links, and an editor for the visual programming of safety critical plants. A common feature to these systems is that a series of translation steps is performed to produce a "target" (e.g., the binary executable or the telegram), starting from some sort of "source" (e.g., a program expressed in a high-level language, or a graphical system representation). Every translation

step leads from an expression of a language $\mathcal{L}$ to an expression of another language $\mathcal{L}'$, and is critical to the correctness of the overall translation. In general, we address the problem of proving the correctness of a single translation step:

> Let T be a translator from a source language $\mathcal{L}$ to a target language $\mathcal{L}'$. Let "$\equiv_{sem}$" be a formally defined semantical equivalence, capturing the notion of correct translation between expressions of $\mathcal{L}$ and $\mathcal{L}'$. Let $\overline{P}$ be an expression in $\mathcal{L}$. Let $\overline{P'}$ be the expression produced by the execution of T on $\overline{P}$. Prove the correctness of the execution of T by showing the semantical equivalence between $\overline{P}$ and $\overline{P'}$:
>
> $$\overline{P} \equiv_{sem} \overline{P'} \tag{1}$$

In order to guarantee that the verification is correctly performed, a base requirement is that any tool possibly used to achieve it must be itself certified, i.e. there must be a way to guarantee that it is itself correct. Moreover, in most cases (see for instance [3]), additional requirements arise from the facts that the development tools are operated by end-users which are not experts in logic, and that the translation process is subject to time constraints. We explicitly adopt such constraints: the verification must be performed in a fully *automatic* and *efficient* way, and the proof of correctness of the translations must be easily understood and validated by end users.

In this paper, we present a general methodology for solving the problem stated above. Section 2 describes in detail a two-phase approach to the verification problem. First, a part of the proof is factored out and performed once for all, off-line. This simplifies the remaining proving so that it can be automatically and efficiently performed on-line. Section 3 describes the logging-and-checking architectural solution to the problem of certifying the tools used in the proof phases. In the logging-and-checking architecture, a prover is specified as two independent subsystems, only one being critical, and whose correctness can be trusted. Section 4 motivates the choice of a specific prover, ACL2 [8], for the accomplishment of the off-line proof phase, and describes the ongoing redesign of such tool to adhere to the logging-and-checking architecture. Section 5 provides an example of a real utilization of the approach in the framework an industrial project, describing in detail the specific on-line tool used in the process. The final sections of the papers describe related work and draw some conclusions.

2 The off-line and on-line proof phases

In general, the proof of the semantical equivalence in two expressions of two different languages is too difficult to be dealt with in an automatic and efficient way. A divide-and-conquer style solution consists in identifying some "simplified" verification conditions, which both imply the semantical equivalence, *and* are amenable to an automatic and efficient on-line verification[1].

This leads to the key idea of our approach, depicted in Figure 1; namely, that of splitting the semantical equivalence proof of a source expression $\overline{P}$ and its target $\overline{P'}$ into an off-line and an on-line phase:

Off-line phase: Define an alternative notion "$\equiv_{simp}$" of equivalence "the verification of

[1] As Boyer and Moore stated in [2], "*... it is often easier to prove a theorem that is stronger than the one needed by some particular application*". Of course, proving a stronger conjecture leads to restrict the number of accepted executions of the translator, and liveness issues must be kept in mind, although not explicitly stated in the problem.

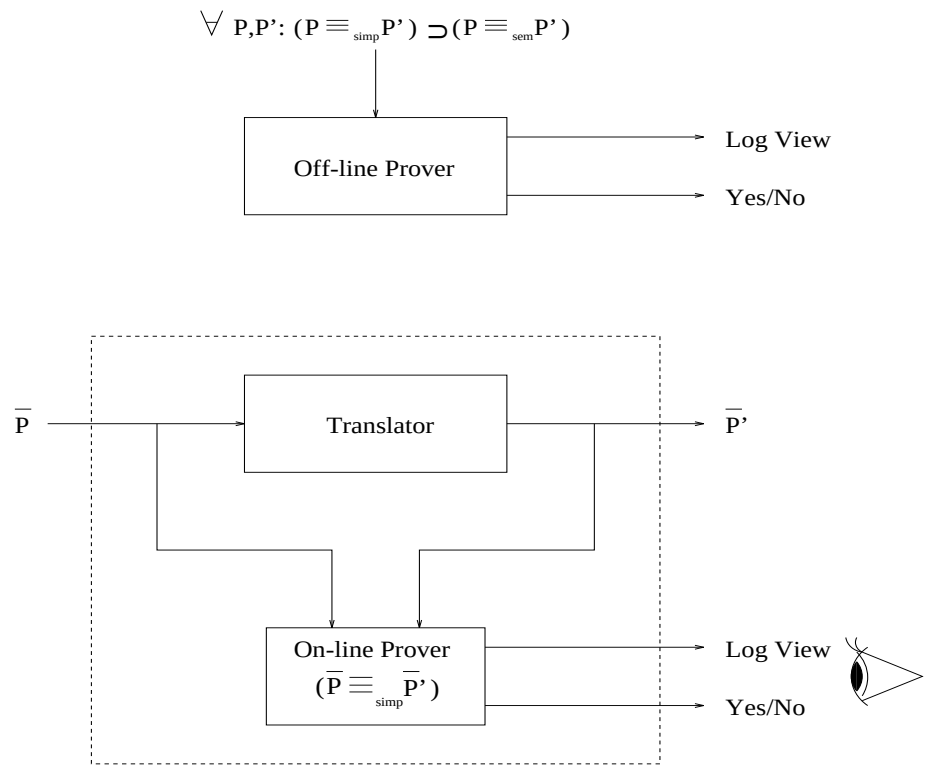

Figure 1: The off-line and on-line approach.

which is simply automatizable", and prove *off-line*, once-for-all, that it implies the given notion of semantical equivalence, namely:

$$\forall P, P' : (P \equiv_{simp} P') \supset (P \equiv_{sem} P') \tag{2}$$

In general, this proof is rather complex, both because of the universal quantification, and because of the gap between the arbitrarily complex "$\equiv_{sem}$" definition and the "easy to check" alternative "$\equiv_{simp}$". Due to the recursive definition of terms in languages, this proof is usually carried out by some sort of induction. Therefore, using a generic, full blown, induction-based state-of-the art prover to perform the proof is practically a must. The interactive proof style characterizing most of such provers, and involving the possibility of failing proof attempts, and consuming large quantities of time, can be afforded, given the once-for-all nature of the proof and the absence of efficiency automaticity constraints. Note that it is not mandatory that an end-user of the translator can understand the proof produced in this phase: this proof is delivered as a part of the system, and is therefore taken care of by the system designers, i.e., by theorem proving experts.

On-line phase: Given the source $\overline{P}$ and its target $\overline{P'}$, prove *on-line* (and obeying the requirements of full automaticity and efficiency) their equivalence according to the simplified definition:

$$\overline{P} \equiv_{simp} \overline{P'} \tag{3}$$

The proof of this theorem, which must be performed for each execution of the translator, is simple "by design", both because it refers to ground expressions, and because of the way "$\equiv_{simp}$" has been defined. This allows for the development of a custom, automatic and efficient on-line prover, "tailored" over the particular equivalence theorem to be proved. The on-line prover must present (other than a yes/no kind of answer describing whether the translator's execution was correct or not) a Log which can be understood and validated by end users with no experience of logic and theorem proving.

It can easily be seen that the conjunction of 2 and 3 implies 1. Notice that, in Figure 1, the provers used in the on-line and off-line proof phases feature some notable architectural differences. The custom, theorem-tailored on-line prover directly reads the source and target expressions, and implicitly interprets such input as the request to prove the simplified equivalence expressed by 3, hardcoding the definition of the one kind of conjecture it has to deal with. Thus, in the figure, the on-line prover box features two separate input lines; moreover, to represent the hardcoding, the logical statement of the simplified equivalence is drawn into the box itself. On the opposite, generic provers, such as that used to perform the off-line proof of 2, are usually designed to read conjectures using some standard logic syntax. In the figure, this corresponds to the off-line prover box featuring a single input line, and an external definition of conjecture 2. The figure also highlights the different end-user Log comprehension requirements, by using a graphical representation of the user's eye in the on-line phase only.

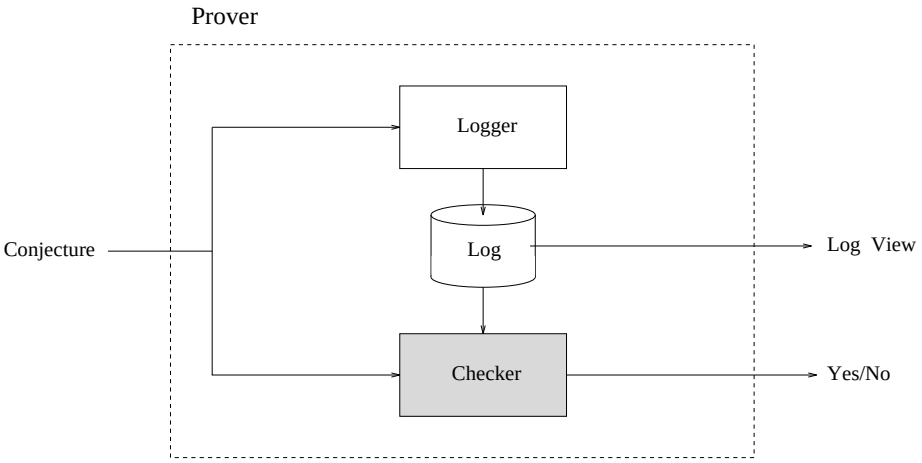

Figure 2: The logging-and-checking approach.

3 Certifying correctness

The requirement that each tool involved in the verification process can be itself certified forces us to adopt an automated reasoning system for both phases of the verification. It would otherwise have been infeasible to formally validate the (at least partially) informal proof accounts which might be manually produced in the off-line phase by a mathematician. Therefore, we are faced with the problem of certifying the automated reasoning tools we use in both phases - which we cannot trust because, due to their possibly complex design,

they might be subject to flaws just as well as the translators they are intended to verify[2]. We call this problem, the *tool certification problem*. We uniformly adopt the logging-and-checking architecture, shown in Figure 2, as a solution to the this problem both in the on-line and off-line phase. In the logging-and-checking architecture, each proving component is specified in terms of two independent programs, a Logger and a Checker. The Logger generates a Log, containing (a description of) the proof of the conjecture. The Checker certifies the correctness of the proof by checking that the Log actually represents a proof of the conjecture. The representation in the figure is a conceptual rather than a structural one; when instantiating the logging-and-checking architecture to the on-line prover, it must be considered that the single input theorem is substituted, in the concrete, by the two expressions $\overline{P}$ and $\overline{P'}$. The different shadowing of the Logger and the Checker highlights their different criticalities w.r.t. the safety of the overall system, that is w.r.t. the possibility of accepting a non-proved conjecture, or a conjecture for which a flawed proof is generated. Namely:

- the Logger is *not* critical. Every error introduced by the Logger is caught by the Checker, which causes the Logger's execution to be rejected.
- the Checker is critical; this can be trivially argued.

Of course, there still remains the "semantic gap"; that is, both the logging and the checking must be performed upon concrete data structure representations, whose communication and handling may introduce errors which may corrupt the overall system's correctness. Thus, the software interfaces allowing the Checker and the Logger to receive and output data must be considered when discussing criticality issues. In detail:

- the interface allowing the Checker to read the conjecture is critical, because it may cause the Checker to accept an incorrect Logger's execution.
- the interface allowing the Checker to output the yes/no answer is critical; this can be trivially argued;
- every remaining interface (including that allowing the Checker to read the Log) is not critical; this derives from the fact that they may only modify the Logger's external behavior w.r.t. the user or the Checker; but the Logger's correctness is already not guaranteed.

Thus, the correct execution of the overall system just depends on that of the Checker (and of all of its input/output interfaces, but the one reading the Log). A desirable and expected feature of checkers is that their implementation is much simpler than those of the corresponding loggers. This is due to the fact that a Checker does not need to search a proof, rather it parses an existing one and only checks that its deduction steps comply to some set of rules[3]. Validating such a simple piece of code is a task which is often feasible by simple code inspection, and can be assumed to be correctly performed; therefore, the Checker can be assumed to be correct. Of course, it can be argued that an incorrect execution of the Checker is possible due to a COTS-induced runtime failure, leading to an unsafe behavior of the overall system: no absolute guarantee of safety can ever be achieved. Furthermore, our achievement is nevertheless very significant: by making the correctness of the overall system dependent only on a selected, trusted section of code, we enhance its level of safety. Intuitively, the

[2] In fact, running them on COTS would make them untrustable anyway.

[3] Note that the size of a checker linearly depends upon the number of rules it deals with, and upon their size. Ideally, a Checker should base on a small set of "easy to implement" rules.

probability of an error occurring in the execution of the (independent and trusted) Checker such that it annihilates a correspondent error occurred in the corresponding execution of the Logger is much lower than that of an error occurring in the (untrusted) Logger itself.

4 A tool for the off-line verification phase

Building a complex reasoning system, powerful enough to tackle problems such as the off-line proof stated in Equation 2, is an extremely difficult and time-consuming task. Existing state-of-the-art provers embed several man years in design, and refined heuristics whose fine tuning results from a vast experience accumulated by wide groups of users. Thus, in our opinion, it is sensible to base on an existing tool to accomplish the off-line of the phase, rather than implement one from scratch. Our choice fell on the ACL2 prover [8] because of a number of features which make it one of the most powerful existing reasoning systems, and especially suit our needs. We summarize below the most relevant:

- It implements a quantifier-free first-order logic of tree-structured data and functions defined by recursion on well-founded orderings, which is especially suitable to the representation of hardware systems and programs alike;
- It provides direct support for an important set of primitive data types, whose explicit axiomatization would otherwise be necessary to prove properties related to expressions, such as 2: strings, rational numbers, characters, lists, symbols;
- It provides a water-clean language and logic, Acl, whose syntax and semantics corresponds to an applicative subset of Common Lisp; this allows the executability of logic statements, which is of enormous importance when checking that the logic definitions appropriately model the phenomena being described;
- It includes several complex heuristics, including simplification by distinct classes of rules, type-set reasoning, a linear arithmetic decision procedure, generalization, destructor elimination, elimination of irrelevance, cross-fertilization, and induction. These extremely powerful heuristics are of great help in the proof process;
- It is a very robust tool.

Unfortunately, although brilliantly engineered, ACL2's design is that of a tool whose proof reports solely consist of an informal commentary which describes the proof construction. Therefore, in order to fulfill our requirements, ACL2 needs to be reconfigured as a Logger, and a Checker must be built for it[4]. This is the object of a long-term project, started in Fall 1996, where the Open Mechanized Reasoning Systems (OMRS) technology [5] has been selected as the methodology to proceed to the ACL2 logging-and-checking redesign.

Synthetically, OMRS is a general architecture for the specification and design of reasoning systems as Logical Services [14], featuring a three-layered structure which allows for a clear separation of the logic aspects of a prover from its strategies, and its interaction capabilities:

$$\text{OMRS} = \mathit{Logic} + \mathit{Control} + \mathit{Interaction}$$

[4] Although, actually, a limited Emacs-based proof tree facility is available, together with a companion checker named Acl2-Pc. Neither tools are satisfactory w.r.t. our aims, both because of the Checker's size, and because of the fact that the Checker is strictly integrated with the Logger, rather than being an independent piece of code.

Namely, the logical behavior of an OMRS is specified by its *reasoning theory* component, which counterparts the logical notion of formal system by defining a sequent-based notion of OMRS rule; the *control* component describes the strategies implemented by the system, via primitive tactics, composed using a language of tacticals; finally, the *interaction* component provides a description of the way the system interacts with other systems, including OMRSs and human users. OMRS sequent and rules are extended to deal with control and interaction annotations to represent heuristic and I/O information respectively. Two aspects of OMRS are particularly relevant w.r.t. our objectives. First, the correctness of the reasoning theory component of an OMRS and of its interaction level[5] is sufficient to guarantee that the overall system is correct: a flawed strategy can not lead to unsound proofs [4]. This allows us to focus on selected sections of an OMRS code when dealing with correctness issues. Moreover, the execution of an OMRS results in a structured account of the proof, called a Reasoning Structure. A Reasoning Structure is basically a graph describing the proof construction, and from which a proof Log is easily obtainable by appropriately pruning the logically irrelevant control and interaction information. For more details, see [5] and [1]. In the following subsections we describe (what we have developed of) the ACL2 Logger, and the ACL2 Checker.

4.1 The ACL2 Logger and Checker

Provided that the whole proof building process is captured within the OMRS specification, the result of the OMRS re-engineering of ACL2 is a Logger - regardless of the level of detail at which the proof construction has been specified. Of course, the granularity of the specification influences the features of the Logger's implementation, and those of the Logs it produces. For instance, a Logger implementing a coarse specification, adopting few complex rules, is likely to generate compact Logs; on the opposite, a very detailed specification would cause Logs of a bigger size to be produced.

ACL2's top-level proof strategy is often referred to as the ACL2 waterfall, because its behavior can be represented in analogy to the famous Escher picture (see [5]). Synthetically, the waterfall strategy consists in recursively applying a set of top-level heuristics, often referred to as the ACL2 processes, to the clauses whose conjunction represent the current conjecture to be proved. An induction scheme is applied to those clauses which the processes could not transform in any way, producing new clauses to be resubmitted to the waterfall. When no clauses are left to be proved, the initial conjecture is accepted; otherwise, the waterfall may loop forever, or stop and reject the conjecture according to a set of heuristic conditions.

In order to obtain an ACL2 Logger, we started by considering a coarse grained OMRS formalization of the waterfall, where each ACL2 process is specified as a single OMRS rule. This resulted in a set of 25 OMRS rules, 9 of which reflecting the black-box specification of processes, and in a set of complex tactics. In order to discuss the features of both the logic and control description in OMRS, we analyze in detail an example of an OMRS rule and an example of a complex tactical expression.

OMRS rules represent the behavior of reasoning modules, by explicitly mentioning the complex data structures they handle - including control and interaction info. This is a major difference w.r.t. rules which only describe relations amongst abstract logical entities, and whose utilization in representing actual reasoning systems would thus be extremely difficult. Consider the following abridged[6] `Wat1LstRule` rule definition:

[5] Actually, only those interfaces which may lead critical components of the system to an unsafe externally observable behavior are critical.

[6] For reasons of space and ease of understanding.

$$\texttt{Wat1LstRule}\quad \frac{\begin{array}{c} evh \vdash_{wat1} cl_1; \widetilde{hstEl}_1, pspv_0 \rightsquigarrow \widetilde{cls}_1; \{Proc_1\} \circ \widetilde{hstEl}_1, pspv_1 \\ evh \vdash_{wat1} cl_2; \widetilde{hstEl}_2, pspv_1 \rightsquigarrow \widetilde{cls}_2; \{Proc_2\} \circ \widetilde{hstEl}_2, pspv_2 \\ \vdots \\ evh \vdash_{wat1} cl_n; \widetilde{hstEl}_n, pspv_{n-1} \rightsquigarrow \widetilde{cls}_n; \{Proc_n\} \circ \widetilde{hstEl}_n, pspv_n \end{array}}{\begin{array}{c} evh \vdash_{wat1Lst} cl_1 \wedge \ldots \wedge cl_n; \widetilde{hstEl}_1 \circ \ldots \circ \widetilde{hstEl}_n, pspv_0 \rightsquigarrow \\ \widetilde{cls}_1 \wedge \ldots \wedge \widetilde{cls}_n; \{Proc_1\} \circ \widetilde{hstEl}_1 \circ \ldots \circ \{Proc_n\} \circ \widetilde{hstEl}_n, pspv_n \end{array}}$$

In terms of logical content, `Wat1LstRule` is as simple as a standard cut rule, and describes the way several transformations upon logical entities (namely, clauses) may be composed together. Each schematic sequent defined into the rule is typed; the type appears as a subscript to the dash symbol. Each sequent of type $wat1$ describes the transformation (represented by "$\rightsquigarrow$") of a clause cl_i into a set of clauses $\widetilde{cls}_i$, to be meant in conjunction, under a fixed theory evh. A lot of control information is used and transformed as well. This kind of information is separated from the logical information by a semicolon. Namely, a history list $\widetilde{HstEl}_i$, containing the description of the processes used to derive the clause, is used and updated by appending to it a process identifier $Proc_i$; also, the complex structure named $pspv_i$, which contains several heuristic-driving data (the "prover's special proof vars"), is exploited and modified in the process. The sequent whose type is $wat1lst$ describes the way a conjunction of clauses $cl_1 \wedge \ldots \wedge cl_n$ is transformed into a conjunction of conjunctive sets of clauses $\widetilde{cls}_1 \wedge \ldots \wedge \widetilde{cls}_n$. Correspondent history lists are also transformed, and a modification occurs upon an initial value $pspv_0$ of the prover's special vars. For a detailed description of the semantics of `wat1` and `wat1lst` sequents, and of the structures they involve, see [1].

The OMRS definition of tactics closely resembles that used in tactical provers, e.g. LCF or HOL; some significant extension (namely, a `MEMOIZE` statement) has been introduced to efficiently represent the strategies implemented in actual reasoning systems. In [1] it is shown how the lack of such additional definition would cause hard efficiency problems. Consider the following complex tactical expression, where tactical language constructs are capitalized to distinguish them from the tactic names:

```
Waterfall0 = ( MEMOIZE SkipMiss IN
                ( (SkipMiss THEN Waterfall0Ph1)
                  ORELSE
                  ( (NotHit)
                    ( (SkipMiss THEN Waterfall0Ph2)
                      ORELSE
                      ( (NotAbort)
                        ( (SkipMiss THEN Waterfall0Ph3) ) ) ) ) ) )
```

`Waterfall0` performs a case analysis by an `ORELSE`-based chain, in order to choose between three different `ANDTHEN`-composed tactics. The initial `MEMOIZE` statement causes a hash table to be created for the `SkipMiss` tactic, in order to possibly reuse its future executions. Then, the `(SkipMiss THEN Waterfall0Ph1)` tactics is invoked; if it fails, generating a `NotHit` failure message, the `(SkipMiss THEN Waterfall0Ph2)` is invoked; if also in this case a failure message is generated (namely, `NotAbort`), then `(SkipMiss THEN Waterfall0Ph3)` is called. Each `(SkipMiss THEN Waterfall0Phx)` tactic invocation results in the invocation of the primitive `SkipMiss` tactic (possibly reusing some previous result), followed (unless a failure is generated) by the application of `Waterfall0Phx` upon the sequents it generates, according to the standard `THEN` tactical definition.

Codewise, the resulting Logger features an extended code, w.r.t. 19KBytes of the original waterfall. It includes an interpreter for the language of tacticals (approx. 70 KBytes of source code) and a set of structured functions corresponding to the compiled primitive tactics (approx. 50 KBytes). Note that the tactical language interpreter is a constant price to pay in terms of software weight, regardless of the coarseness of the specification.

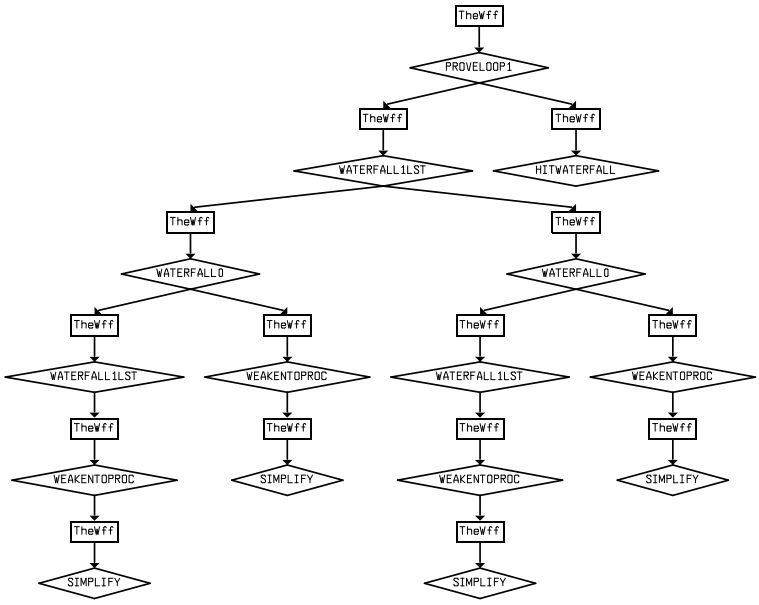

Figure 3: The proof Log for associativity-of-append.

Figure 3 shows the graphical Log representation produced by the Logger for a theorem stating the associativity of the standard `append` operation on lists. The original report consisted of some 70 lines of informal commentary. The rhombi in the Log represent the application of primitive tactics; the name used to refer to the tactics is represented within the rhombi (e.g., `Waterfall1Lst` is the primitive tactic corresponding to the `Waterfall1LstRule` rule). The rectangles in the Log represent the sequents manipulated by the tactics. In order to keep the graphical representation compact in the figure, we have hidden the actual sequent representation, as well as some additional informations, e.g., the instantiations produced by the applications of schematic primitive tactics.

An advantage of using a specification framework such as OMRS is that the specifications of the Logger can be effectively reused to implement a companion ACL2 Checker. Thus, the level of detail in the specifications influences the features of the Checker in a similar way to how it influences those of the Logger. Namely, a coarse grained specification, such as that currently adopted in the Logger, would result in a Checker based upon a small set of very complex rule-implementing functions. Such a Checker could not be easily validated, nor claimed to be trustable. This is why we plan the implementation of a Checker as a successive step to a refinement of the current ACL2 specification and logging.

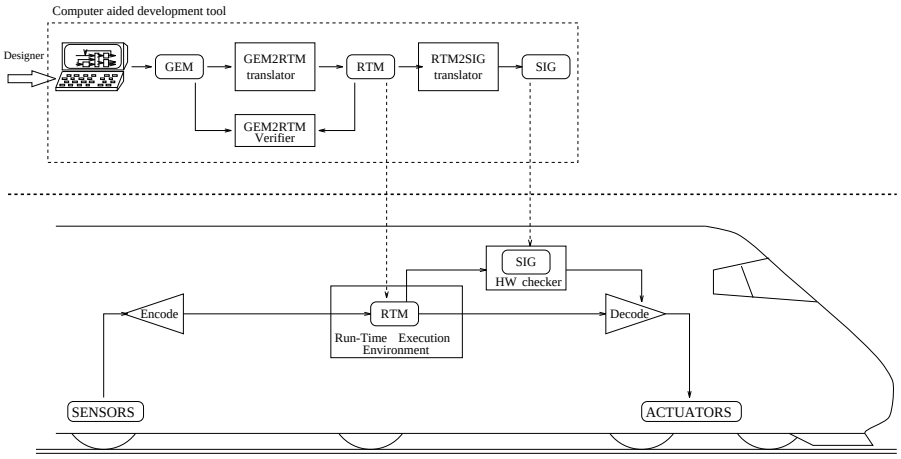

Figure 4: The computer aided development tool for the reactive run-time system

5 An industrial case study

The methodology presented so far was applied in a technology transfer project [3], aiming at the on-line certification of a translator of a safety critical computer aided development tool which is part of one of UNION SWITCH & SIGNAL's next-generation products [13]. The tool is used to develop safety critical reactive systems, such as Automatic Train Protection systems (ATPs), which cyclically read the train speed and position from sensors, checks whether the speed is consistent with a "safety speed curve", and drives the train breaking devices accordingly.

Figure 4 depicts the architecture instantiated on a safety critical railways application. Two subsystems can be identified: the on-board subsystem (below the dotted line), and the Computer Aided Development tool (above). On board, the "Run-Time Execution Environment" is a platform for the execution of reactive programs in RTM format (which is a binary, loadable machine language). The "Run-Time Execution Environment" is based on COTS technology. The correctness of the run-time execution is guaranteed by a fail-safe, very simple (and thus cheap) Hardware Checker. This Checker compares information on the status of RTM execution with a set of criteria which must be satisfied for the execution to be correct (SIG in figure).[7] Data from sensors and to actuators are encoded/decoded (with Cyclic Redundant Checksums and Residue Number Systems) to protect from data corruption by means of simple, fail-safe components (see figure).

At development time, the user interacts with the Computer Aided Development tool, by specifying an application program on the graphical interface. The program is stored in GEM (Generic Entity Model) format, a simple high level language. The Computer Aided Development Tool is responsible for the translation of the application program into RTM format, which can then be safely executed. The Computer Aided Development tool is also based on COTS technology. It is a complex software package, written in a high level language, including graphical libraries, compiled on a conventional compiler, running on a portable PC, and intended to be operated on the field by design engineers. The project addressed the on-

[7] The details of the HW Checker cannot be disclosed in this paper since they involve proprietary information. However, the details are not necessary to the goal of this paper.

line verification of the (safety critical) GEM2RTM Translator (in figure 4), generating RTM programs from GEM programs [8].

The resulting on-line prover (GEM2RTM Embedded Verifier in figure 4) was developed according to the methodology described in this paper.

GEM and RTM programs can be thought of as cyclically acquiring data from sensors, executing the corresponding instructions, and delivering outputs to actuators. (The actual details are out of the scope of this paper, but can be found in [3].) GEM and RTM programs are given semantics as functions mapping finite sequences of inputs into sequences of outputs. Semantic equivalence between a GEM program G and a RTM program R (i.e. the instance to this particular domain of equation 1) is defined as follows:

$$G \equiv_{sem} R \quad \stackrel{\text{def}}{=} \quad \forall I.([\![G]\!]_{I/O}(I) = Decode([\![R]\!]_{I/O}(Encode(I))) \tag{4}$$

Intuitively, G and R are equivalent if they feature the same input/output behavior in all possible environments, i.e. for all possible sequences of inputs. I is an input for G, and $Encode$ and $Decode$ represent the "translation" of data from the input spaces of the different programs. $[\![.]\!]_{I/O}$ is defined in terms of a standard, state based semantics for GEM and RTM programs. Intuitively, acquisition of data and execution of instructions are mappings from program states to program states. Output delivery is defined as a mapping from state sequences to output sequences.

The simplified conditions for a GEM program G and an RTM program R are defined as follows:

$$G \equiv_{simp} R \quad \stackrel{\text{def}}{=} \quad \exists M, \mathcal{T}, \mathcal{R}_M.\Phi(G, R, M, \mathcal{T}, \mathcal{R}_M) \tag{5}$$

M is a mapping from the variables of the GEM program to the variables of the RTM program. Intuitively, M induces a correspondence between the computation states of the GEM and RTM programs. Corresponding states imply corresponding input/output behavior. $\mathcal{T}$ is a mapping associating each GEM instruction i_G in G to the sequence $\mathcal{T}(i_G)$ of RTM instructions. Intuitively, i_G and $\mathcal{T}(i_G)$ lead to corresponding states if executed in corresponding states. $\mathcal{R}_M$ is a sequence of instances of translation rules. The translation rules can be used to prove that a GEM instruction and a sequence of RTM instructions correspond. These rules, based on the informal description of the behavior of the GEM2RTM translator, are sufficient for equivalence. Φ expresses a number of conditions on $G, R, M, \mathcal{T}, \mathcal{R}_M$. For instance, G and R must be syntactically correct programs, M must relate the variables of G and R appropriately (e.g. it must be functional and invertible), $\mathcal{T}$ must take into account all the instructions of both programs. A basic feature of Φ is that it does not involve semantic notions, but only structural properties (e.g. type of variables, number of instructions). If two particular programs $\overline{R}$ and $\overline{G}$ are such that $\overline{G} \equiv_{simp} \overline{R}$, we say that they satisfy the Syntactic Verification Conditions. The reduction given by the off-line/on-line decomposition is that the Syntactic Verification Conditions are much simpler to prove than semantic equivalence.

The correctness of the off-line/on-line decomposition is guaranteed by proving Theorem 2, instantiated with Definitions 4 and 5:

$$\forall G, R.(G \equiv_{simp} R \quad \supset \quad G \equiv_{sem} R) \tag{6}$$

[8] A project under development is dealing with the formal verification of the correctness of the visual compiler [13, 7]. A future project will apply the same methodology developed in this project to the formal verification of RTM2SIG translation. These problems are not addressed here. However, notice that the decomposition of the certification problem yields the problem of certifying the composition of the different certification steps. This might require further off-line proofs.

The proof goes by induction on the length of input sequences and on the structure of programs. The off-line verification was carried out manually. Nevertheless, a bug was found in the design of the translator during the off-line phase. In particular, one of the translation rules $\mathcal{R}_M$ was not semantics preserving. Since $\mathcal{R}_M$ is derived from the informal specification of the translator, it turned out that the translator itself implemented a non semantically preserving transformation. In the future we plan to mechanically verify the off-line proofs in ACL2.

The Logging and Checking approach to the certification problem for the on-line phase was very valuable in this particular problem. Indeed, proving the Syntactic Verification Conditions requires to determining witnesses for the existential quantifiers (namely $M, \mathcal{T}$ and $\mathcal{R}_M$), and may require a lot of search. Since the Logger is not critical, it was possible to exploit the custom nature of the problem and use the data structure provided by the translator (e.g. the symbol table) in order to drive the search (e.g. the correspondence M from GEM to RTM variables).

The project required the design of a Checker which could be understood and validated by the user, and efficient. This deeply influenced the design. A possible choice was having a proof Log specified in terms of primitive rules (e.g. and introduction, equality substitution, bounded quantification rules). This would have allowed for the use of a Checker for a general purpose logic, implementing a few simple inference rules. However, a proof presentation in terms of such low level inference rules is hardly understandable by a user which is not an expert of logic. Furthermore, the proof of the Syntactic Verification Conditions can easily blow up for large programs (which is often the case with real-world applications). Some conditions in the Syntactic Verification Conditions (e.g. the fact that M is functional and invertible over GEM variables) amount to an aliasing problem, and therefore expand to a quadratic number of simple checks. Therefore, the Checker was designed as a custom program, containing a number of special purpose decision procedures. For instance, the aliasing problem is simply solved by a subroutine implementing a nested loop. This choice results in a more understandable Log, and the code of the Checker can be directly mapped to its specifications.

The developed Checker is a few hundred lines of C code, with an extremely simple control structure. Interestingly enough, the largest part of the Checker is not the one devoted to checking the correctness of the proof generated by the Logger, but rather the interface to the external environment, which needs to compare the formal model of the GEM and RTM programs (contained in the Log) with the actual GEM and RTM programs. Indeed, not only does the Checker need to make sure that the proof generated by the Logger is a correct proof, but also that the Syntactic Verification Conditions hold for the particular GEM and RTM programs it has in input.

The developed Verifier is extremely efficient. It is able to verify the translation of thousands of instructions in a few seconds. By running the Verifier on substantial examples of GEM to RTM translations, slight implementation bugs were detected in the translator. These bugs were pinpointed by the Checker failure to certify the correctness of the translation.

6 Related Work

As far as we know, no previous work has proposed the idea of decomposing the verification task into two distinct phases, the off-line proof done "once for all" and the on-line proof for each individual translation.

In the off-line phase, we can of exploit all the techniques previously developed, for instance the work on the "CLI short stack" [10]. In this phase, we adopt a logging-and-checking

mechanism to address the problem of the certification of the ACL2 proof. To this extent, the closest work to ours is that on the EVES prover [9]. Even if similar in spirit, our work is in practice rather different: we aim at logging and checking a complex and powerful prover such as ACL2 in a principled way by using the OMRS technique. According to the OMRS approach, the theorem proving strategies are expressed as tactics. This resembles the idea underlying tactic-based theorem provers (e.g. HOL [6]). In fact, an alternative approach to the certification of the off-line proof is the use of a tactic-based theorem prover. Indeed, in principle, proofs generated by means of tactics are guaranteed to be sound given the correctness of primitive tactics. Nevertheless, state-of-the-art tactic based provers, when used to verify complex systems, e.g. in hardware verification, have required a rather onerous development of several special purpose tactics which would be hardly applicable in our application domain. We have adopted ACL2 since we want to fully exploit its powerful theorem proving heuristics and decision procedures in the off-line phase. Moreover, we need to extend the notion of tactic w.r.t. that of current tactic-based provers in order to log efficiently some of the ACL2 strategies [1].

To our knowledge, the idea of coupling a translator with an on-line prover which compares for individual translator runs inputs and outputs is novel too[9]. To this extent, some similarities exist with the work on Proof Carrying Code (PCC) [12, 11]. The PCC technology has been devised for application domains which are very different from ours, namely the problem for a fail-safe host platform to execute code provided by an untrusted code producer. The producer must couple each program with a proof which is then proof checked by the host. The checker must check the proof against the actual untrusted code. This idea is similar to our idea of an on-line logging-and-checking of the proof log against the actual inputs and outputs of the translator (see Section 5 for a discussion about this problem in the industrial case study). In PCC, the checker is general, for instance based on simple extensions of Natural Deduction encoded in LF. Our on-line checker is based on special purpose decision procedures in order to check efficiently the proof log, to keep the log size reasonable, and to have both the checker and the log which can be understood and validated by end-users with no experience in logic. Finally, the current PCC framework does not seem to address the problem of the generation, the logging and the checking of complex proofs, even if some extensions in this sense are currently planned.

7 Conclusions and future work

In this paper, we have described a methodology for the automatic verification of safety-critical translators, run on top of non-failure-safe platforms. The methodology involves the reduction of the on-line verification phase through an off-line phase, and the design of proving tools following the logging-and-checking approach. We have showed the application of the methodology in the frame of an industrial project; in such case, off-line proofs were conducted by hand rather than exploiting an automated tool. In order to be able to fully exploit the technology in other projects, we pursue the aim of obtaining a logging-and-checking version of a state-of-the-art prover, ACL2, by re-engineering it through OMRS. We have already designed a coarse ACL2 specification, and obtained an ACL2 Logger implementation from it. This was performed in few man-months of work, inclusive of a startup time needed to design a number of re-engineering support tools.

As a part of our ongoing ACL2 redesign project, we intend to refine the OMRS specification, in order to build a finer grained Logger; to design and implement the correspondent Checker,

[9] Actually, a similar approach has been independently promoted by Amir Pnueli for the verification of a compiler (personal communication)

and to test them in proving conjectures such as that defined in the off-line phase of the project shown in section 5. Moreover, we intend to test our methodology by verifying other translators, possibly more complex than the GEM2RTM we showed, and possibly commercial compilers, where individuating automatizable verification conditions is likely to become a major issue.

References

[1] Piergiorgio Bertoli. *Using OMRS for real: a case study with* ACL2. PhD thesis, Universitá di Roma 3, 1997. Forthcoming.

[2] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[3] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, and P. Traverso. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *Proc. Computer-Aided Verification (CAV'97)*, Haifa, Israel, June 1997. Also IRST-Technical Report 9701-04, IRST, Trento, Italy.

[4] F. Giunchiglia, P. Pecchiari, and A. Armando. Towards provably correct system synthesis and extension. *Journal of Future Generation Computer Systems*, 12:123–137, 1996.

[5] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning Theories: Towards an Architecture for Open Mechanized Reasoning Systems. Technical Report 9409-15, IRST, Trento, Italy, 1994. Also published as Stanford Computer Science Department Technical note number STAN-CS-TN-94-15, Stanford University. Short version published in Proc. of the First International Workshop on Frontiers of Combining Systems (FroCoS'96), Munich, Germany, March 1996.

[6] M.J. Gordon. A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis.* Kluwer, 1987.

[7] D. Guaspari, C. Barbash, and D. Hoover. Checking critical code. Technical Report ORA TM-95-0081, Odyssey Research Associates, Ithaca, NY 14850 USA, September 1995.

[8] M. Kaufmann and J S. Moore. Design Goals for ACL2. Technical Report 101, Computational Logic Inc., Austin, Texas, 1994.

[9] S. Kromodimoeljo, B. Pase, M. Saaltink, D. Craigen, and I. Meisels. The EVES system. In *Proceedings of the International Lecture Series on "Functional Programming, Concurrency, Simulation and Automated Reasoning" (FPCSAR).* McMaster University, August 1992.

[10] J.S. Moore, editor. *Special Issue on Systems Verification, Journal of Automated Reasoning.* Vol. 5, n. 4, 1989.

[11] George Necula. Proof-carrying code. In *24th Annual ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, 1997.

[12] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, 1996.

[13] J. Profeta, N. Andrianos, B. Yu, B. Jonson, T. DeLong, D. Guaspari, and D. Jamsek. Safety Critical Systems Built with COTS. *Computer*, 29(11):54–60, November 1996.

[14] Ian Sutherland and Richard Platek. A plea for logical infrastructure. In *TTCP XTP-1 Workshop on Effective Use of Automated Reasoning Technology in System Development*, pages 1–3, 1992.