The project has provided further results and benefits. First, the development of the formal proofs of the semantic equivalence sketched in section 4 revealed that one of the specified translations performed by the compiler was not semantically preserving. More precisely, the translation of GEM instructions of the form v_1 `<-` v_1 `==` v_2, where v_1 is boolean, had been specified as the mapping presented in section 4. This bug was discovered while failing to prove the corresponding case for Theorem 4.1.

Second, by running the Verifier on substantial examples of GEM to RTM translation, implementation bugs were detected in the compiler. These bugs were pinpointed by the Checker failure to certify the correctness of the translation.

The approach followed in this project is completely general and independent of the fact that it is used to certify a particular component of VFRAME. Moreover, in spite of the fact that some of the modules of the Logger and the Checker depend on the compiler, the architecture of the Verifier can be re-used for the certification of different systems. Such an architecture can be devised for several safety critical systems which need a certification of translations of data/programs/models/specifications, such as compilers, translators, and specifications editors.

References

[1] B. Boyer and Yu Y. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In *Proc. of the 11th Conference on Automated Deduction*, number 607 in Lecture Notes in Computer Science, pages 416–430. Springer-Verlag, 1992.

[2] R.S. Boyer and J.S. Moore. A Theorem Prover for a Computational Logic. In M. E. Stickel, editor, *Proc. of the 10th Conference on Automated Deduction*, pages 1–15, Kaiserlautern, Germany, July 1990. Published as Springer LNAI, number 449.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 20(2):107–125, February 1995.

[9] 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.

type of GEM and RTM abstract instruction, the mapping M^c determines the structure of the corresponding concrete instruction. For instance, in the case of `EQUALITY-II`, it holds:

> $M^c($`EQUALITY-II` $i_1\ i_2\ i_3) = v_3$ `<-`v_1 `==` v_2, where v_j is the variable of the i_j-th GEM declaration.

The correctness of the decomposition of the Verifier in the logging/checking schema presented above is guaranteed by proving that, for any possible individual compilation, the Checking Conditions imply the Syntactic Verification Conditions. The proof is done by constructing a mapping M from the M^d, M^{ai} and M^c contained in the Log. This proof is not difficult and we omit to comment it for lack of space.

6 Certification assumptions

As any other form of certification, the certification provided by the Verifier is not absolute. First of all, it depends on the assumption that the formal models of GEM and RTM computation are accurate with respect to the actual GEM and RTM computation. This problem has been tackled with a strict integration between IRST and ANSALDO in the development of the formal model.

Second, it depends on the correctness of the formal results. The proofs have been performed manually and, though not technically complex, they are rather long. In order to improve the confidence on their correctness, a further step will be the mechanization of these proofs by means of an industrial-strength prover (e.g. [8, 5]). A prover can be much more accurate than a human in working out the details of fifty pages of proofs. Of course, the mechanization would not give a 100% certification, because in principle the prover itself could be questioned. Several approaches to this problem are under development. Some of these aim at the development of logging/checking mechanisms for full blown provers (e.g. [6]). Others aim at the development of a provably correct prover, see for instance [2] and [3].

Finally, the certification provided by the Verifier depends on the correct implementation and execution of the Checker. The confidence in this assumption is rather high, as the Checker has been designed to be validated by end users, and given its simplicity also the compilation and the execution platform can be trusted.

However, our approach does not rely on the accuracy of the correspondence between the actual GEM and RTM programs in input to the Verifier and their formal model. This correspondence is guaranteed by the Checker itself. As explained in previous section, a large part of the Checking Conditions are intended to make sure that there is a one-to-one correspondence between the actual programs and their model. This innovative feature is due to the requirement that the Verifier must be integrated in VFRAME, and therefore a manual intervention to generate the formal model would not be feasible.

7 Conclusion and Results

We have developed a Verifier which is able to certificate a real industrial safety critical software component. The Verifier is embedded within the safety critical architecture. As requested by the industrial application, it is fully automatic and efficient. The Verifier has verified the translation of thousands of instructions in a few seconds. The certification requirement has been satisfied. The correctness of the Verifier depends only on an extremely simple and short portion of its code (a few hundred lines of C code) which can be easily understood and validated by end users.

```
-- ABSTRACT GEM INSTRUCTIONS
EQUALITY-II  1 2 3    -- 1. abstract gem instr
-- ABSTRACT RTM INSTRUCTIONS
EQU 6 2 3  -- 1. abstract rtm instr
OR  1 6 6  -- 2. abstract rtm instr
EQU 6 3 5  -- 3. abstract rtm instr
AND 1 1 6  -- 4. abstract rtm instr
```

The log header states the length of the GEM and RTM declarations and instructions. The declaration mapping and the instruction mapping are a description of the M which has been found by the Logger. Variables are (abstractly) referred to in terms of indexes indicating the position of the corresponding declaration in the declaration list. For instance, the GEM integer variable B is referred to as 2, while the corresponding RTM variables B1 and B2 are referred to as 2 and 3. Instruction indexes are defined analogously. For instance, the GEM instruction at index 1, i.e. A <- B == C, is mapped to the four RTM instructions at indexes 1,2,3 and 4. The abstract GEM/RTM instructions provide an abstract description of the (concrete) instructions of the GEM/RTM programs. For instance, EQUALITY-II 1 2 3 is the abstract instruction corresponding to the GEM concrete instruction A <- B == C. EQUALITY-II stands for an equality operation between two integer variables and 1 2 3 are the indexes of the declarations of the variables A,B,C, respectively.

Notice that the Log can be validated by end users which have no experience of logic and theorem proving (part of Requirement 2). Indeed, the logical proof steps actually performed by the Logger (e.g. substitutions, universal bounded quantifications) are presented to the users as information on the syntactic structures of the two programs, e.g. the two programs corresponding instructions (corresponding to substitution), the lengths of the two programs (corresponding to the bound on universal quantification).

Let us consider now the Checking Conditions. Intuitively, these can be thought of as making sure that the Log is syntactically correct. For instance, the Checker must check that abstract instructions are well formed and well typed, that the indexes of the variables in the abstract GEM/RTM instructions are consistent with the log header, and that the abstract GEM and RTM instructions correspond with respect to the declaration and instruction mappings. For instance, in the case of EQUALITY-II, the following condition must hold:

$$M^{ai}(\texttt{EQUALITY-II}\ i_1\ i_2\ i_3) \;=\; < \begin{array}{l} \texttt{EQU}\ M^d_1(i_1)\ M^d_1(i_2)\ rd, \\ \texttt{OR}\ rd\ rd\ M^d(i_3), \\ \texttt{EQU}\ M^d_2(i_1)\ M^d_2(i_2)\ rd, \\ \texttt{AND}\ M^d(i_3)\ rd\ M^d(i_3) > \end{array} \tag{1}$$

M^{ai} is a mapping from GEM abstract instructions to RTM abstract instructions. M^d is the mapping from GEM variable indexes to RTM variable indexes as reported in the declaration mapping. In the example, $M^d(2) = \ < 2, 3 >$. $M^d_1(i)$ and $M^d_2(i)$ are the first and second elements of the sequence $M^d(i)$, e.g. $M^d_1(2) = 2$. rd is the index of the RTM variable tmpequ. Notice that the Log actually satisfies the Checking Conditions, e.g. M^{ai}(EQUALITY-II 1 2 3) is <EQU 6 2 3, OR 1 6 6, EQU 6 3 5, AND 1 1 6>.

However, knowing that the Log is well formed is not enough. The Checker must make sure that the Log is a proof of the theorem we are interested in, i.e. the equivalence of the two GEM and RTM programs in input to the verifier. This amounts to verifying that the Log applies to the actual GEM and RTM programs. For instance, the Checker must check that the actual programs contain as many instructions and declarations as stated in the log header, and that there is a bijective correspondence between GEM/RTM abstract instructions (contained in the Log) and the concrete GEM/RTM instructions (parsed directly from the actual GEM/RTM). For each possible initial states of GEM and RTM. The step case states that, if the sequences of states generated by any input sequence of length n are equivalent, then the sequences of states generated by any input sequence of length $n+1$ are. We prove this by showing that each state transition (e.g. input acquisition, execution of instructions) preserves the equivalence. This was done for each possible GEM instruction and legal typing configuration of GEM operands and results.

As an example, let us consider the proof that the GEM instruction v_1 <- v_2 == v_3 and the RTM instruction sequence $M(v_1$ <- v_2 == $v_3)$ preserve equivalence. Part of this proof consists in showing that the value of v_1 in the final state is equal to the value of $M(v_1)$ in the final state. We have two cases. If (the values of) v_2 and v_3 are equal, then v_1 has value 1 and, by induction hypothesis, $M_1(v_2)$ is equal to $M_1(v_3)$ and $M_2(v_2)$ is equal to $M_2(v_3)$. This implies that, after the first RTM state transition, t has value 1, after the second $M(v_1)$ has value 1, after the third t has value 1, and at the end of the final transition $M(v_1)$ has value 1. The other case is similar.

5 Functional specification of the Logger/Checker

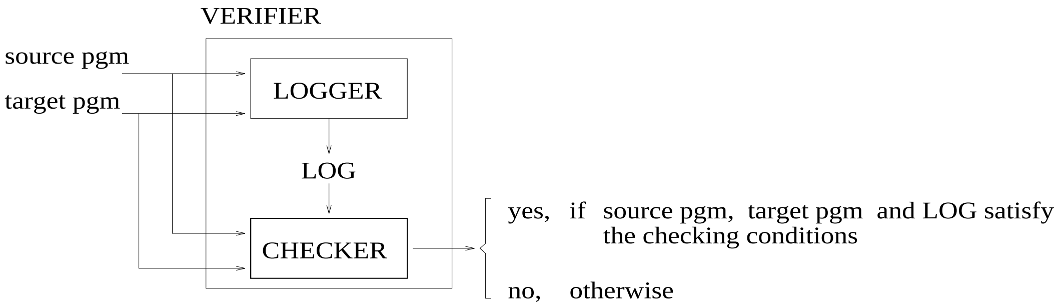
VERIFIER
source pgm
target pgm
LOGGER
LOG
CHECKER
yes, if source pgm, target pgm and LOG satisfy the checking conditions
no, otherwise

Figure 4: The VFRAME Logger/Checker Architecture

The formal specification of the Verifier is refined into the specification of a system (see figure 4) composed of two independent programs, a Logger and a Checker. The Logger generates a Log containing the proof that the Syntactic Verification Conditions are satisfied. The Checker certifies that the proof is correct by checking that some *Checking Conditions* on the source program, on the target program and on the Log, are satisfied. This decomposition allows for the achievement of Requirement 2 (see section 1), being the Checker the only critical component of the Verifier.

The first step is to provide a formal characterization of the output of the Logger, i.e. the Log. Intuitively, the Logger tries to build a proof that the GEM program and the RTM program in input to the Verifier satisfy the Syntactic Verification Conditions, and writes it in the Log. In particular, the Log contains a description of the mapping M from the GEM program to the RTM program, as described in previous section. As an example, let us consider the following Log, generated by the Logger when its inputs are the two programs in figure 2.

```
-- LOG HEADER
3   -- GEM decl length
1   -- GEM instr length
6   -- RTM decl  length
4   -- RTM instr length
-- DECLARATION MAPPING
1 1                -- 1. bool decl
2 <2,3>            -- 2. int decl
3 <4,5>            -- 3. int decl
-- INSTRUCTION MAPPING
1 <1,2,3,4>      -- 1. int instr
```

variable declarations and instructions. The following are some of the conditions which must hold of the GEM and RTM programs. If v is an integer variable in g, then $M(v)$ must be a pair $\langle v_1, v_2 \rangle$ of distinct variables in r (in the following we write v_i as $M_i(v)$). For each integer input [output, resp.] variable v in g, $\langle M_1(v), M_2(v) \rangle$ are input [output] variables in r, and for each input [output, resp.] variable v' in r, there exists an input [output] variable v in g such that, either $M_1(v) = v'$ or $M_2(v) = v'$. The GEM and RTM programs in figure 2 satisfy the conditions above. For instance, $M(\texttt{B}) =< \texttt{B1}, \texttt{B2} >$ and all and only GEM input/output variables are mapped to RTM input/output variables.

Each instruction in the GEM program must correspond to a sequence of instructions in the RTM programs. Not only does this correspondence depend on the operator of the instruction (e.g. `==`), but also on the types of GEM variables (e.g. integer or boolean). For instance, if v_2 and v_3 are integer variables, a GEM instruction of the form v_1 `<-` v_2 `==` v_3 must be mapped by M on the sequence of RTM instructions reported below, where t is an RTM (temporary) variable which does not correspond to any GEM variable. Comparison between boolean variables would be translated in a different way.

$$\begin{array}{lcrcccc} M(v_1 \texttt{ <- } v_2 \texttt{ == } v_3) & = & t & \texttt{<-} & M_1(v_2) & \texttt{EQU} & M_1(v_3), \\ & & M(v_1) & \texttt{<-} & t & \texttt{OR} & t, \\ & & t & \texttt{<-} & M_2(v_2) & \texttt{EQU} & M_2(v_3), \\ & & M(v_1) & \texttt{<-} & M(v_1) & \texttt{AND} & t \end{array}$$

The instructions of GEM and RTM programs in figure 2 satisfy the above conditions, the (temporary) variable t being `tmpequ`.

The functional specification of the Verifier through the Syntactic Verification Conditions makes it possible to satisfy Requirement 1 (see Section 1). Indeed, the problem of verifying the Syntactic Verification Conditions is decidable and not computationally complex. As a consequence, the Verifier can be implemented as a fully automatic and efficient system.

The next step was to make sure that this specification is actually such that correct compilations generate target programs which are semantically equivalent to source programs. Two (GEM and RTM) programs are semantically equivalent when, given two sequences of "equivalent" input vectors, they compute two sequences of "equivalent" output vectors. The notion of equivalent (sequences of) input vectors is formalized by extending the mapping M to input vectors. If I is a GEM input vector, $M(I)$ is an equivalent RTM input vector. The precise definition of M on input vectors depends on some details of the encoding of RTM variable values into RTM variable values, which, for lack of space, we do not explain in this paper. The notion of equivalent output vectors and equivalent states is formalized similarly.

We formally prove that, for any possible individual compilation, the Syntactic Verification Conditions imply semantic equivalence. This result is a direct consequence of the following theorem, stating that the Syntactic Verification Conditions imply that the GEM and RTM programs are state equivalent. The intuitive meaning is that the execution of the programs when given corresponding inputs proceeds through sequences of pairwise corresponding states.

Theorem 4.1 (State equivalence between programs) *Let g and r be a* GEM *and an* RTM *program, such that $g \sim_M r$. Then they are state equivalent, i.e. for every input vectors sequence $\overline{I}$ for g,*

$$[\![g]\!](\overline{I}) \;\sim_M\; [\![r]\!](M(\overline{I}))$$

The proof of Theorem 4.1 is done by induction on the length of sequences of inputs $\overline{I}$. The base case, corresponding to the null sequence of inputs, follows from the equivalence between the

For simplicity and readability, in this paper we have written RTM programs "symbolically". RTM programs are, actually, sequences of hexadecimal numbers. A somewhat more "realistic" presentation of the instructions of the RTM program in figure 2 is reported below (lines starting with `--` are comments). For instance, consider `operation #1`. `0x00000002` and `0x00000014` identify the variables `B1` and `C1`, respectively, `0x0000000d` identifies the operator `EQU`, and `0x00000019` identifies `tmpequ`.

```
-- operation #1 --
0x00000002 0x00000014 0x0000000d 0x00000019
-- operation #2 --
0x00000019 0x00000019 0x0000000b 0x00000001
-- operation #3 --
0x00000003 0x00000015 0x0000000d 0x00000019
-- operation #4 --
0x00000001 0x00000019 0x0000000a 0x00000001
```

GEM and RTM programs were given semantics by formalizing the cyclic execution process informally presented above. The basic entities are *computation states*, mapping the variables of a program on the corresponding values. In the following, we write the value of a variable v in the computation state s as $v(s)$. Each execution step - initialization, input acquisition, execution of instruction, output delivery - is formalized as an operation on the (computation) state. Initialization is a function mapping a program to the initial state s such that for each variable v, $v(s)$ is the initial value of v. Input acquisition is the function mapping the computation state s and an input vector $I =< i_1, ..., i_n >$, i.e. a tuple of (input) values, to the computation state s', such that, for all input variables v_j, $v_j(s') = i_j$ and for every other variable w, $w(s) = w(s')$. Output delivery is a function from a state to an output vector, i.e. a function from a state s to the tuple of the values $< v_1(s), ..., v_m(s) >$, where $v_1, ..., v_m$ are the output variables of the program. Each (GEM and RTM) instruction is semantically interpreted as a function mapping state to state. For instance, the interpretation of the GEM operation v_1 `<-` v_2 `==` v_3 is the function $[\![v_1 \texttt{ <- } v_2 \texttt{ == } v_3]\!](s) = s'$, where $v_1(s')$ is `1` iff $v_2(s)$ and $v_3(s)$ are equal, `0` otherwise, and for all v other than v_1, $v(s') = v(s)$

Intuitively, the (state) semantics of an embedded program p, written $[\![p]\!]$, is a function from the set of all finite sequences of input vectors to state sequences, such that for every input sequence of length k, $\overline{I} = I_1, \ldots, I_k$, the sequence of states is obtained by composing initialization, input acquisition and the execution of the sequence of instructions.

4 Functional specification of the Verifier

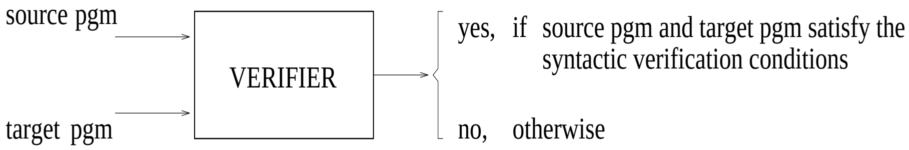
source pgm
target pgm
VERIFIER
yes, if source pgm and target pgm satisfy the syntactic verification conditions
no, otherwise

Figure 3: The Verifier

A correct compilation is defined by a set of *Syntactic Verification Conditions*, i.e. conditions on how source programs are mapped syntactically into target programs. The Verifier is thus formally specified as a system which, given in input the two programs, answers "yes" only if the two programs satisfy the Syntactic Verification Conditions (see figure 3). Let g and r be a GEM and an RTM program, respectively. Intuitively, g and r satisfy the Syntactic Verification Conditions (written as $g \sim_M r$) iff they correspond through a mapping M from GEM to RTM

3 GEM and RTM programs

GEM and RTM programs can be thought of as "embedded programs", i.e. programs which are embedded in an external environment (at different abstraction levels). After (variable) initialization, embedded programs are executed cyclically. At each cycle, values are acquired from the external environment (e.g. information from sensors such as train speed) and stored in input variables. Then, the instructions of the program are executed (e.g. to compute a control algorithm), and the computed values are stored in output variables and then delivered to the external environment (e.g. information for actuators such as control to a breaking device).

```
-- GEM PROGRAM                -- RTM PROGRAM
-- DECLARATIONS               -- DECLARATIONS
1. A, 0, Output, bool         1. A1, 0, Output
2. B, 0, Input,  int          2. B1, 0, Input
3. C, 0, Input,  int          3. B2, 0, Input
                              4. C1, 0, Input
                              5. C2, 0, Input
                              6. tmpequ, 0
-- INSTRUCTIONS               -- INSTRUCTIONS
1. A <- B == C                1. tmpequ <- B1 EQU C1          -- operation #1 --
                              2. A1     <- tmpequ OR tmpequ   -- operation #2 --
                              3. tmpequ <- B2 EQU C2          -- operation #3 --
                              4. A1     <- A1  AND tmpequ     -- operation #4 --
```

Figure 2: Examples of GEM and RTM programs.

Embedded programs are composed of (variable) declarations and instructions. A GEM variable declaration contains a variable identifier, its initial value, information about whether it is an input or an output variable, and its type (e.g. boolean or integer). For example, in figure 2, the GEM program contains the declarations of the variables `A,B,C`. All of them are initialized to the value `0` (in the case of boolean variables `0` and `1` stand for `false` and `true`, respectively). `A` is a boolean output variable and `B,C` are integer input variables. GEM instructions have type restrictions, and can have multiple arguments and multiple results. The GEM program in figure 2 contains only one instruction, `A <- B == C`, which tests if the two integer variables `B` and `C` are equal and assigns the boolean result to `A`.

RTM programs are not typed. In order to prevent undetected data corruptions, information is manipulated in form of codewords, uniform data structures containing, among other things, a numerical value, the identifier of the variable where the codeword is stored, and a CRC protection. RTM variables can be thought of as the result of an "expansion" of GEM variables. For instance, in the programs in figure 2, the integer variables `B` and `C` are expanded into two pairs of variables, `B1,B2` and `C1,C2`, respectively [2]. The boolean variable `A` is translated into the single variable `A1`. RTM instructions are expansions of GEM instructions as well. In the example, the GEM instruction `A <- B == C` gets expanded into the four RTM instructions. The intuition underlying this expansion is that the two GEM variables `A` and `B` are equal if and only if the RTM corresponding variables are pairwise equal, i.e. `B1` is equal to `C1` and `B2` is equal to `C2`. `operation #1` compares the first two pairs (`B1` and `C1`) and the boolean result is assigned to a variable introduced by the compiler (`tmpequ`). The result is stored in `A1` (`operation #2`). The second pair is compared and the result is stored in `tmpequ` (`operation #3`). The result (`A1`) is the conjunction of the results of the two equality comparisons (`operation #4`).

[2] The actual details of this expansion can not be discussed here. The expansion depends, among other things, on some configuration parameters of the compiler and can change from compilation to compilation.

machine is called the cyclic Runtime Executive. Figure 1 shows an overview of VFRAME. At the software level, VFRAME is partitioned into the Off-line and Runtime systems.

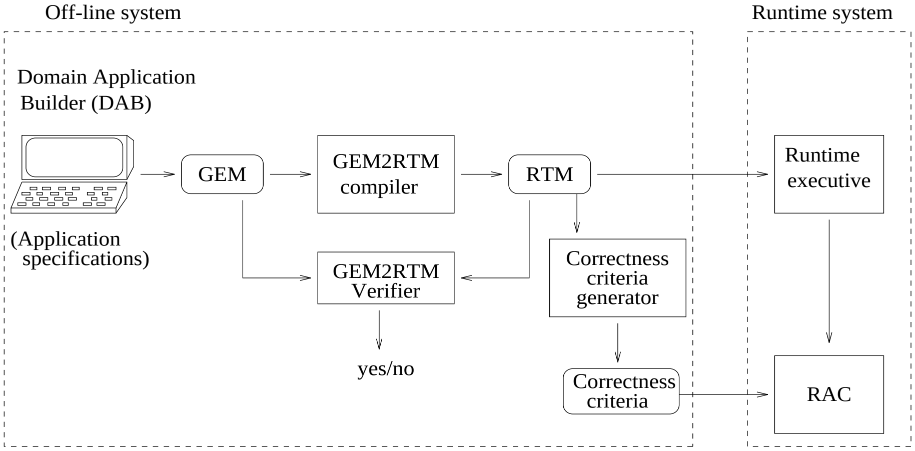

Figure 1: VFRAME overview

The Off-line system provides an application programming environment in the form of a Domain Application Builder (DAB), i.e. a graphical interface designed to allow specification of both the system hardware and application algorithms. A visual language compiler translates graphical specifications into a unique generic and domain independent form, called the Generic Entity Model (GEM), which represents the application in terms of standard operations, like boolean and arithmetic operations, access to tables, etc. The GEM is not yet in a form which can be executed by the Runtime Executive. A compiler, called the GEM2RTM compiler, translates a GEM into a loadable and executable program, called a Run Time Model (RTM). The compiler decomposes the GEM into a sequence of primitive executable operations. At the RTM level, information is stored in form of "codewords", i.e. words protected with CRC to detect data corruptions. The RTM is loaded onto the Runtime Executive. A correctness criteria generator independently processes the RTM and generates precomputed data describing what the correct result of each primitive operation should be. These correctness criteria are loaded onto a Realtime Application Checker (RAC), a simple, fail-safe hardware checker which performs concurrent checking of the results of the runtime execution against the correctness criteria. The on-line architecture allows for safety quantification: a "probability of undetected error" [9] can be determined which depends on the length of the CRC in the codewords. As a consequence, run time execution can be guaranteed to be performed with a small probability of undetected error by means of techniques based on information encoding and concurrent checking by independent hardware.

The remaining problem is the correctness of the off line translations, where the techniques used for the runtime execution cannot be applied. A project under development is dealing with the formal verification of the correctness of the visual compiler [9, 4]. In this paper we describe the project on the certification of the GEM2RTM compiler. The goal is to embed the (GEM2RTM) Verifier (see figure 1) within the off-line part of the vital architecture. The Verifier can be thought of as a black box which takes in input the source and target programs and answers "yes" only if the target program (generated by the compiler) is a semantically equivalent correct translation of the source program (in input to the compiler). A future project will apply the same methodology developed in this project to the formal verification of the correctness criteria generator.

The design of the Verifier was characterized by the following key steps:

Step 1. A formal semantics has been defined for the source and target languages of the VFRAME compiler, and a notion of semantic equivalence has been devised.

Step 2. The Verifier has been functionally specified as a system capable of proving a set of "Syntactic Verification Conditions" over source and target programs. These conditions have been formally proved to imply the semantic equivalence of the programs.

Step 3. The architecture of the Verifier has been specified in terms of two independent programs, a Logger and a Checker. The Logger generates a Log, containing the proof that the Syntactic Verification Conditions are satisfied. The Checker certifies the correctness of the proof by checking that some "Checking Conditions" hold of the Log, the source program and the target program. The Checking Conditions have been formally proved to imply the Syntactic Verification Conditions.

Requirement 1 was addressed by reducing the (hard) task of proving the semantic equivalence of the input programs to the (easier) task of proving the Syntactic Verification Conditions (step 2 above). Indeed, the direct proof of the semantic equivalence of the two programs would require complex theorem proving techniques, and therefore interaction with a user and high computation time. On the contrary, the Syntactic Verification Conditions can be analyzed automatically and efficiently. The proof of the correctness of this step was performed a priori, once for all.

Requirement 2 was achieved through the decomposition of the Verifier into the Logger and the Checker (step 3 above). The independence of the Logger and the Checker guarantees that the Logger is non critical, and the correctness of the Verifier relies only on the correctness of the Checker. Indeed, if the Logger generates a wrong proof, and the Checker is correct, the Verifier will not accept the compilation. This decomposition is motivated by the fact that the task performed by the Checker (i.e. checking a proof) is in general much simpler than the task of the Logger (i.e. finding a proof). Hence the Checker is a small portion of the Verifier, and can be easily validated. The proof of the correctness of this decomposition was performed a priori, once for all. Since the task of the Verifier has been reduced to proving some syntactic properties of the two programs, the proof steps in the Log (e.g. substitutions) are presented to end users as information on the syntactic structures of the two programs (e.g. the two programs corresponding instructions). As a consequence, the logical steps performed by the Checker can be presented to end users with no experience of logic or theorem proving as simple tests on the syntactic structures of the two programs.

The paper is structured as follows. Section 2 is a brief overview of VFRAME. Section 3 describes, through an example, the source and target programs of the VFRAME compiler and their semantics. Section 4 describes the functional specifications of the Verifier. Section 5 describes how the Verifier specifications are refined into the specifications of the Logger and the Checker. Section 6 discusses some issues and assumptions about the certification requirements of the Verifier. Since the work done involved proprietary information not all of the details can be disclosed in this paper.

2 Overview of VFRAME

VFRAME (Vital Framework) [9] is one of ANSALDO software driven vital architectures used to develop safety-critical applications from commercial, off-the-shelf hardware and software. VFRAME can be thought of as a virtual logical and arithmetic machine that executes a vital algorithm on a vital platform. This virtual machine is a cyclic, finite state machine designed to have fail safe behavior independent of the physical implementation. In this way system vitality does not depend on knowing how the processor might fail. The software which implements this virtual

A Provably Correct Embedded Verifier for the Certification of Safety Critical Software

Alessandro Cimatti[1], Fausto Giunchiglia[1], Paolo Pecchiari[1], Bruno Pietra[2],
Joe Profeta[2], Dario Romano[2], Paolo Traverso[1], Bing Yu[2]

[1]IRST - Institute for Scientific and Technological Research
38050 - Povo - Trento, Italy
[2]Ansaldo Trasporti Spa
Via dei Pescatori, 35 - 16100 - Genova, Italy

Abstract. VFRAME is one of ANSALDO's software driven vital architectures for safety critical products. This paper describes a project whose result is the development of an "embedded verifier", i.e. a system integrated within VFRAME and able to certify the correctness of one of VFRAME components, a compiler. The embedded verifier satisfies two precise requirements. First, the compiler must be certified in a fully automatic and efficient way. Second, the embedded verifier must be itself certified, in a way which can be easily understood and validated by end users.

Submission Cathegory: A - Case Study.

Contact Author: F. Giunchiglia. IRST - Via Sommarive 38050, Povo, Trento, Italy
Phone: +39 461 314517. Fax: +39 461 314591. Email: fausto@irst.itc.it

1 Introduction

This paper describes the results of a project where theorem proving techniques have been applied to the certification of the correctness of a component of VFRAME [9], one of ANSALDO's software driven vital architectures for safety critical products. VFRAME is used for the development of rail transportation industrial applications. The VFRAME component under consideration can be thought of as a compiler which, given in input a (source) program, has to generate a "semantically equivalent" (target) program as its output. The goal of the project was the development of an "embedded verifier", i.e. a system integrated within the VFRAME architecture and able to certify the correctness of the compiler. More precisely, every time a compilation is performed, the Verifier must take in input the actual source program, the actual target program generated by the compiler, and prove that the latter is a correct translation of the former[1]. Being part of VFRAME, the Verifier had to meet the following requirements.

Requirement 1. The Verifier must be fully *automatic*, since it must be used by VFRAME end users, and *efficient*, as in general the final installations on-the-field are subject to time constraints.

Requirement 2. As any other component of the vital architecture, the Verifier must be itself certified, i.e. there must be a way to guarantee that the Verifier is itself correct. An additional requirement is that the certification of the Verifier and the proof of the correctness of the compilations must be easily understood and validated by end users.

[1] A well known alternative approach, the (mechanical) "once for all" verification of the correctness of the VFRAME compiler for all its possible inputs (see for instance [7, 1]), has been rejected since the compiler's environment platform is not guaranteed to be fail-safe and can not assure the correct execution of verified software.

IRST

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

A Provably Correct Embedded Verifier for the Certification of Safety Critical Software

Cimatti A, Giunchiglia F., Pecchiari P., Pietra B.,
Profeta J., Romano D., Traverso P., Yu B.

January 1997

Technical Report # 9701-04

© Istituto Trentino di Cultura, 1997

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of ITC and will probably be copyrighted if accepted for publication. It has been issued as a Technical Report for early dissemination of its contents. In view of thetransfer of copyright to the outside publisher, its distribution outside of ITC prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article.