Certifying an Automated Code Generator Using Formal Tools: Preliminary experiments in the GeneAuto project

N. Izerrouken [1] X. Thirioux [1] M. Pantel [1] M. Strecker [1]

IRIT, University of Toulouse, France

ABSTRACT

This paper reports on the study and early experiments of the available technologies for the formal validation and verification of Automated Code Generator which took place in the GeneAuto project. GeneAuto aims at the development of an ACG for a safe subset of the Matlab/Simulink/Stateflow modelling language which will be used for the development of certified safety critical embedded real time systems in the automobile, aeronautic and space domains and therefore subject to the certification authorities and standards of these domains. The chosen technology is illustrated through the development of a scheduler process for a safe subset of block diagrams. Our purpose is to develop and formally verify some parts of the GeneAuto ACG using the Coq proof assistant. Such a certified code generator guarantees that the correctness properties already proved on the model source code will still hold for the executable code. We focus in this paper on the scheduler part.

KEY WORDS

Automatic Code Generation, Certified systems, Validation, Formal verification, Scheduler, Proof assistant, Coq.

I. INTRODUCTION

Nowadays, safety critical aspects are more and more present in embedded real time systems. In order to ensure the safety of the system users, certification authorities enforce development guidelines both for systems (certification of the systems) and system development tools (qualification of the certified systems development tools). One way to reduce the cost of verifying that the system is correct with respect to its specification and design relies on the use of Automated Code Generator in order to produce the source code automatically from the specification and design models and then to produce the binary executable from the source code (traditional compilers). However, many reports of errors in ACG, especially in compilers which can turn silently a correct safe program into an incorrect unsafe executable code, enforce the awareness that a special attention should be given to ACGs Validation and Verification. Certification authorities usually require that the ACG must be at the same level of safety as the systems it partly generates. The V & V process should then start from the earliest user and tool requirements at the specification phase to reduce the risks of design errors up to the last steps of deployment. Classical approaches of V & V relying mainly on testing the executable code produced by the ACG and reviewing the development items have been applied up to now. However, exhaustive testing is impossible since it is not possible to forecast all possible behaviours of the compiler and usage of the input language. All the more, the cost induced by the high coverage required for high level of safety is also much too heavy. Therefore, such critical systems should rely on rigorous V & V technologies based on formal tools in order to both reduce the cost of testing and improve the safety by going to exhaustive V & V.

Currently, formal tools such as Model Checking or Static Analysis are used for models or programs. However, only the specification or the source code are verified in these formal techniques, compiler can invalidate the correctness of the proved source code because of it errors. ACG developers are aware of this issue and uses various techniques to reduce it, such as manual review of the generated code, automatic tests generation. These techniques are not complete and are costly in terms of development time and program performance. A better approach which has recently been applied to large scale use cases by X. Leroy (see [1]) is to apply formal methods to the compiler itself which preserves the semantics of the source code. Our approach is based on this technique for a formal certification of an ACG.

The development of the Model Driven Engineering technologies requires the use of automated code generators(ACG) from design models to programming languages. Our contribution relates experiments conducted using formal technologies for the verification and validation of an ACG producing real-time certified C code from a safe subset of MatLab/Simulink/Stateflow. Our main purpose is to ensure a zero default result while reducing the costs of test and process based qualification.

Until now, existing ACGs have been verified by standard testing techniques. However, an exhaustive test coverage of such tools still remains impossible, so that industrial research requires formal modelling and verification techniques.

Many formal verification frameworks are available, such as Translation Validation, Proof Carrying Code, static analysis and modelling within proof-assistant. Moreover, verification using a proof assistant seems very promising since the achieve-

ment of the CompCert project, aimed at certifying a C compiler (targeted at the PowerPC assembly language) with the COQ tool.

The purpose of the experiment related in this paper is, on the one hand, to validate an ACG specification, and on the other hand to verify its implementation with respect to its specification. The ACG is relatively different from available works as it ought to translate a safe subset of Simulink/Stateflow to C language instead of C programs to assembly language.

Due to the unstable and opaque semantics of Simulink/Stateflow, which often changes depending on its version, we first define the semantics of a safe subset of Simulink/Stateflow. Then, we state the correctness properties of ACG and we give the correctness proofs for the ACG specification, inspired by the approach used in the CompCert project. We propose to split the ACG into phases such as typing, clock calculus, scheduling, memory optimisation...etc, and to provide models with a synchronous semantics. This kind of semantics is widespread in the European Embedded Safety Critical Software Industries as advocated by tools such as SCADE, Esterel or RT-builder.

Our work aims to develop a certified code generator in Coq. The Code generator we developed is splited to many intermediate modules before generating the target code (C). In this paper, we focus on the scheduler process; it is an independent module from the other verifications such as typing, clock calculus...etc

we present a simple verified scheduler developed and verified in the proof assistant Coq. Our reasoning is based on block diagram structures. The remainder of this paper is organised as follows. Section II recalls existing methods of verification and validation techniques and their potential application to ACG. Section III presents an introduction to the Coq proof assistant, present our scheduler algorithm and the main theorems for its verification. Section IV discusses the approach followed by concluding remarks and perspectives.

II. Validation and Verification technologies

The purpose of verification is to show that the produced tool fulfil its requirements. Verification is usually done step by step: verify that the tool requirements are correct with respect to the user requirements, then that the design is correct with respect to the tool requirements, and finally that the implementation is correct with respect to the design.

The purpose of validation is to show that the tool fulfil the user needs, this usually means, if the whole development has been verified, that the user requirements are correct with respect to the user needs.

One important point is that every parts from the user requirements to the implementation can be formally defined. The only step going from human informal knowledge to formal definition is the requirement phase which express the user needs using user requirements. Therefore this step can only be validated using tests done by the user. It is possible to check using tools that the user requirements are coherent, even to a certain point complete, but only tests can show that they are correct.

A. Classical approaches

Two main technologies are currently applied in the V & V of critical systems: tests and system reviews. These are the simplest possible approaches which can be applied in almost any context and are very useful for finding errors but they do not scale well to the complexities of systems and in particular ACG. It is usually quite easy to define what are the realistic inputs for a system. It is much more complicated for an ACG as we must define what are the common use of a language and this can change a lot from users to users.

1) System development reviews: A review consists in a human proofreading of the various documents concerning a system development in order to be convinced of the correctness of the development.

2) Tests: A test provides both system inputs and outputs. A system validates a test if it produces the required outputs when it is submitted with the appropriate inputs. A test inputs can be hand-written or generated to validate a given property. A test outputs are usually hand-written, except if an oracle for the system exists which can predict what the outputs will be according to the inputs.

B. Application to the V & V of ACG

One major point in GeneAuto is the V & V of the code generator itself. This point is of tantamount importance for the success of the integration of formal technologies for V & V in GeneAuto. After several rounds of exchange with the industrial partners involved in certification authorities, the following point have been put forward: “Qualification constraints as defined by the certification authorities are more stringent for code generation tools than code validation ones”. In fact:

- a code generation tool takes the place of a human code writer, the produced code must then be qualified with the same constraints as human written code, therefore the code generation tool must be qualified with the same level of constraints;
- a code validation tool helps a user in assessing that the code he has written is correct, it is therefore only another way of asserting the correctness of the code, but it is only one more way which may, or may not, be used, therefore it should not be qualified with the same level of constraints.

The practical guidances which follow from these points and are currently applied in the development of code generators is that code generators should be as simple as possible and that, if this is possible, some code generation services should be expressed as model validation tools which validate that an annotated model is a correct refinement from an initial model. The following architectural principles for GeneAuto can be derived from this point:

- GeneAuto toolkit should be split in several complementary tools: on the one hand model validation tools, and on the other hand code generation tools;

- Several intermediate modelling languages should enable the user to express the various annotations and properties which enable a simple generator to produce optimised target code. The fact that a model annotation are valids and that the annotated model is a correct refinement of the base model should be asserted by validation tools. The code generator should therefore rely on the model annotation in order to produce optimised code instead of using information computed inside the generator itself. The tool requirements are easier to express as it not needed to define how to compute the annotation used for the generation of optimised code inside the code generation specification itself. The implementation is easier to validate. The workload of annotating the models is transferred to the user of the tools which provide the annotated input models. One point is that these annotation may only be required for specific optimisations and the cost is only paid by the user of the optimisation service not by all users.

One possible approach to reduce this additional workload is to provide tools which helps the user in annotating the intermediate models. These models are built by the user and validated both by the user and the validation tools. These tools are not annotated model generators but mere helper tools (such as the sophisticated editors available in the current software development environments). The user provides the base model, these tools help the user in annotating the model, the annotated model is then validated and the code is then generated from the valid annotated model. There is then only a small overhead in workload for the user and a much less expensive code generator validation. Let's note that the qualification of these helpers at the same level as the produce models is still possible and these tools can then be completely hidden from the user which will input the base model and the tools will output the generated code hiding the intermediate steps.

Currently, ACG are mainly defined in two steps:

- the user requirements express that a given language (programming or modelling) must be translated to another language
- the tool requirements define precisely how each elements of the source language is to be translated in the target language.

The correctness of the implementation with respect to the tool requirements is currently mainly done by generated code reviews. For tests expressing each elements, and some combination of elements, the generated code is compared with the code predicted using the tool

C. Requirements for the architecture of the ACG

This part will give insights on the GeneAuto ACG architecture which is presented in more details in [2].

The following steps and intermediate languages (mostly the input and output languages which various kind of annotations) are currently in use in the first prototype of the GeneAuto ACG:

- Type annotation for signals in the block diagram. The validation that these annotations are compatible with the types of the blocks input and output ports;
- Scheduling annotation for the blocks. Validation that these annotation are compatible with the data flow constraints expressed by the signal connecting the blocks input and output ports;
- Sharing annotation for the various signal whose values are exclusively required and which may be stored in the same variable in the generated code (the basic generation strategy use one variable for each signal, this allow to reduce the number of variable to the minimum required for the sequential execution of the blocks in the schedule expressed by the annotation in the input model). Validation that each required signal value is available when a block is scheduled;
- Enabling condition annotation which allows to execute only the blocks which computes the signal values required to compute a block output signal values. This is, for example, a backward propagation of enabling condition in the various branches of conditional blocks. Validation that each required signal value is available when a scheduled block is enabled.

The following steps have been introduced for the final version of the ACG to manage requirements currently not taken into account by the current prototype:

- Clock (sampling rate) annotation for blocks in the GeneAuto input language. Validation that each required signal value is available when a block is scheduled, enabled and at the right clock tick (memory or rate adaptation blocks are inserted at the right places);
- Usage of data flow design patterns to express control flow such as proposed by M. Pouzet and J-L. Colaço. In fact, control flow is expressed using common data flow signals and blocks. These specific signals and blocks are then annotated to express their relationship with the implicit control flow diagram. Validation that the annotation expresses a well formed control flow diagram. A helper will allow the user to view the control flow diagram corresponding to the annotated data flow diagram.

D. Formal technologies based improvements

As usual, the validation of a code generator must assert that the implementation of the code generator is compliant to the requirements expressed in the specification. There is usually two kind of requirements:

- User level requirements: These requirements define the input and output languages, their syntax and semantics, mandatory design and coding rules. They also define the set of high level property that the generator must fulfil (for example, the behaviour of the generated code is the same as the behaviour of the input model on both functional and non-functional aspects).
- Tool level requirements: These requirements are derived from the previous ones. They express precisely how

each concept and relation from the input language must be translated to concepts and relations from the output language. More abstractly, they relate the input language and the part of the output language which is mapped from the input language.

Formal validation technologies can be applied to both kind of requirements:

- Formal specification of the requirements are always a good point as they ease the validation of the implementation using any validation technologies (even only tests). In the case of ACG, the formal specification must express:
  - The input and output languages, that is, their syntaxes, execution semantics (operational), validation semantics (axiomatics), and design rules. This is part of the user requirements.
  - The translation rules (relation between input and output languages). This is part of the tool requirements.
- The assessment that the tool requirements have been taken into account requires to check that the generated code for each model strictly follow the relationship expressed in the translation rules whatever the semantics of the languages. This means that the translation rules are taken as correct hypothesis. To our knowledge, formal technologies have not been commonly used in this purpose. In fact, the translation rules are an executable specification. Therefore an implementation of this executable specification, which might not follow the translation rules, is usually not required. However, if another implementation is required, for example to take into account scalability constraints or to manage some specific constraints introduced by the certification authorities for tools implementing code generators, we propose the following approaches:
  - Oracle based automatic tests: When a code generator is applied to an input model, we also apply the executable specification to the same model if this is possible (scalability constraints) and then we check that both results are syntactically equivalent. The formal specification of the translation is therefore an oracle which allows to validation the implementation of the code generator. The validation procedure can then generate a huge amount of input models, apply both executable specification and code generator and check that both results are equivalent. This procedure can use test coverage technologies in order to produce the right kind of tests and cover most of the input language. We can, for example, rely on the users experience in order to define common model patterns for which the coverage should be better.
  - Enhanced source level comparison of generated code: The checking in finite time that two programs do the same thing is an undecidable and incomplete problem. It is however possible to have less coarse comparison procedure than mere string equality by using a bounded subset of the behaviour equivalence relation between programs. Practically, the behaviour equivalence relation is expressed as rules relating programs in the same language which have the same behaviour (for example, `if C B1 else B2` behaves the same as `if !C B2 else B1`). Then a bounded number of rules can be applied in order to show that two programs are equivalent. This is a kind of equational unification bounded by the number of equation which can be applied. This kind of technology has already been experimented by IRIT for comparing service in SOA architectures on the computing Grid.

E. Formal technologies for V & V of ACG

ACG are complex pieces of critical softwares since they perform delicate code transformations. Bugs in compilers provide incorrect generated code from correct source programs. In order to reduce the error risks, many verification approaches were developed these techniques can be formal and test based techniques. Formal methods such as Model Checking, Static Analysis, Proof Carrying Code, Translation validation, Certified Compiler the technique on which we are inspired.

In this section, we describe the most prevalent techniques.

1) Proven Development: The certification using Coq proof assistant [1], of an optimising back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. The compiler was written mostly in Coq directly in functional style. This technique is very promising in the verification validation domains. The technique was applied to a critical system. The principle of the technique is to divide the compiler to independent successive transformations; each transformation is proved correct in Coq. The formal semantic of all intermediate used languages was defined, and then, an executable code was extracted from the compiler specification.

2) Certified Development: Certified development [1]: This kind of technique allows to build a correct system by construction starting from a specification of the system required properties and then applying iteratively refinement rules which allow to build the system without breaking the properties in the specification. The implementation is then derived from the specification. Each refinement step must then be proven to be correct. Proof requirements are built from each refinement step which ensure this correctness property. These requirements must then be proven using a proof assistant.
The most successful certified development approach is the B methodology which has been applied to the development of many critical systems in railways and automotive industries.

3) Proof Carrying Code: Proof Carrying Code (PCC) [3] is a technique that was developed to be used for safe execution of untrusted code. Initially, the approach was applied to verify mobile code by validating the received code without applying cryptographic techniques.

The principle of the PCC is to return, as well as the generated code produced by the compiler, the proof of the

property *P(S,C)* which is checked independently by the code user. The key idea behind PCC is that the code producer is required to create a formal safety proof that attests to the fact that the code respects the defined safety policy. Then, the code consumer is able to use a simple and fast proof validator to check, with certainty, that the proof is valid and hence the foreign code is safe to execute.

PCC has many uses in systems whose trusted computing base is dynamic, either because of mobile code or because of regular bug fixes or updates.

Its use is extended to be used in many applications such as : extensible operating systems, Internet browsers able to download code, active network nodes and safety-critical embedded controllers. PCC and credible compilation [4] make use of a certifying compiler. Hence, the certifying compiler can produce an incorrect proofs.

4) Translation Validation: Translation validation was introduced by Pnueli in 1998 [5], [6], [7], [8] as a technique to detect translators bugs at compilation process.

Instead of proving in advance that translators (compilers, code generators) produce correct target from a source code, each run of the compiler is validated; the validation process consists to verify that the generated code implements correctly the source code.

Technically, the translation validation approach complements the compiler by a verifier *Verif(S,C)* which verifies a correctness property *P(S,C)* by static analysis of *S* the source code and *C* the generated code.

The following formula must be verified to certify the verifier.

$\forall SC, Verif(S,C) \rightarrow Prop(S,C)$

As the verifier is another tool which is also error prone, there is no guarantee that the generated code, be it correct or not, will or not pass through the verifier.

J-B.Tristan and X.Leroy have recently [9] developed 2 validator corresponding to 2 instruction scheduling optimisations: list scheduling and trace scheduling.

Most of the verifier currently in use rely on model checking or static analysis technologies.

5) Model Checking: Model checking is a fully automated technology which rely on building explicitly or implicitly the reachable state space for all the possible execution for a given system in all possible execution contexts and then to check that the required properties are valid in all these states. If the property is not valid, it is then possible to build a counter-example, that is an execution going from an initial state to the state which breaks the property. This approach is usually complete and decidable if the state space is finished and if enough resources (computing power and storage) are available to build it. If not, the models or properties should be simplified (abstraction) manually or automatically in order to reach completeness and decidability. The technique of model checking is used in systems with finite tests. However, in the case of critical systems, the infinite cases to test disable the use of Model Checking technique. Even in finite cases, model checking do not scale well and all experiments conducted previously for ACG could not be applied on real size industrial cases.

6) Static Analysis: One potential approach to reduce the number of execution contexts and execution steps is to approximate the semantics of the system and of the fix-point computations required to compute it. This approach, which can be used for programs and models, will either produce false alarms (the property is not valid for the abstract system but is valid for the concrete one), or be unsound (the property is valid for the abstract system but is not valid for the concrete one). This depends on the way the systems are approximated by the abstraction relation. Only the first kind of static analyses can be used for critical systems, that is a static analysis should not be unsound. Several technologies have been devised in order to define static analyses tools. Some require the proof of soundness (for example, type inference or type checking), some rely on a correct by construction soundness (for example, abstract interpretation). Type based static analyses [10], [11] are in use in most currently available compilers. Abstract interpretation is at the basis of the successful tools PolySpace, AbstInt and ASTREE. Static analysis have been used for ACG V & V by X. Rival (see [12], [13]) in a variation of translation validation where static analysis is used instead of model checking. However, it is very complex to prove the correctness of static analysers.

F. Synthesis for GeneAuto

The validation of the refinement of the user requirement to the tool requirement consist in assessing that the translation rules expressed in the tool requirement are correct with respect to the semantics of the input and output languages. For example, it must be asserted that the execution of the code resulting from the translation of a given model is the same as the execution of this model. The comparison relation can take into account both functional and non-functional aspects. Formal technologies are well adapted to this kind of validation and many medium to full size experiments have already been conducted in previous projects for programming language to assembly language code generators.

In the first year of the GeneAuto project, we have compared the various V & V technologies both classical and formal in order to choose the best suited technologies and show the improvements it could bring to the usual industrial practices in the field.

The criteria chosen for the comparison were the following ones:

- SL Cost of the specification of the input and output languages: This is the cost of providing the supplementary information required for using the technology with respect to the standard specification of the languages. A mark means that no additional input is required apart from the input model simulator and the output language processor. E mark means that everything must be redefined;
- ToU Cost of the use of the validated tools: This is the cost introduced by the use of the validated tools with

respect to the use of the invalidated code generator. A mark means that there is no additional costs;

TD Tools developments costs: This is the additional cost of developing a validated tool with respect to the development of an invalidated one. A mark means that there is no additional cost;

QT Qualification cost: This is the additional cost for validating the tools with respect to the development of non qualified tools. A mark means that there is no additional cost;

TeU Technology usage costs: This is the additional cost of learning how to use the validation technology with respect to the classical development of the tools. A mark means that there is no additional cost;

TS Scalability: This shows the ability of applying the tools to real size models. A mark means that any models can be managed;

UC Real size use case: This shows that this technology has already been applied to real input and output languages. A mark means GeneAuto level complexity of languages;

FT Fault tolerance: This shows the risk of validating tools which are in fact invalid. A mark means that the technology ensure that a validated tool is valid. This score points out that the validation may rely on external validation tools which may be themselves invalids;

CA Certification authorities practices: This is a first rating provided by the GeneAuto WP6 partners relating the V & V technology to the current practises in the certification authorities. This is the cost required to convince the authorities that the technology can be used for qualification. A mark means mostly no additionnal costs.

These criteria have been used to compare the following technologies:

- Proven development (PD), Certified development (CD), Testing with oracles (TO), Translation validation (TV), Static analysis (SA) which are the same technologies which have been described previously in the communication.
- Assessment translation (AT) is a technology which is specific to code generator. It relies on a translation of the assessment which have convinced the user that its input model is correct. The principle is then to translate the model and then to check that the assessments still hold and convince the user that the generated code is still correct. This approach can be applied very easily to the test scenario designed for the model. If these tests have been expressed as models, they can be translated as tests at the code level and applied on the generated code for the model. Notice that the whole model execution environment should then be described in the modelling language.

It led to the synthesis expressed in table I.

	PD	CD	TO	AT	TV	SA
SL	3	4	0	2	1	4
ToU	0	0	0	2	3	3
TV	2	4	0	1	2	4
TQ	0	0	4	1	2	2
TeU	3	3	0	1	2	4
TS	0	0	4	2	3	3
UC	2	1	0	4	2	2
FT	0	0	4	1	2	3
CA	1	0	0	1	2	3

TABLE I
SYNTHESIS OF V & V TECHNOLOGIES

The previous synthetic table allowed to choose the technologies which would be experimented first in GeneAuto. A very strong point was the fact that the technology used should be the most error-prone as no certification authorities would allow to choose an error-prone technology event if it was better for all the other criteria.

To conclude, it is required to provide formal specifications of the input and output languages and of the translation between them. Then, two kind of validation should be considered using as far as possible formal technologies. Then, both kind of validation should be considered: validation of tool implementation with respect to tool requirement and validation of tool requirement with respect to user requirement. The first kind is related to the validation currently in use for this kind of tools in the industrial partners practise. The second kind validation the refinement between the requirements and can provide much higher confidence in the produced tools. It also correspond to the current state of the art in academic research. We therefore advocate to experiment both kind of validation with the most appropriate technologies.

One last important point is the chosen architecture for the toolset. There should be several intermediate steps for the building of the GeneAuto input language annotated models with helpers and validation procedure in order to reduce the workload and ensure the correctness of the model refinements. The code generator should apply on fully annotated models in order to be the most simple as possible and therefore more easy to validate.

III. PRELIMINARY EXPERIMENT: SCHEDULING ALGORITHM

Before presenting our approach, let us introduce the system Coq and how it is used.

A. The Coq Proof Assistant

In this section we briefly introduce some principles of the Coq proof assistant and illustrate them with block diagrams specifications. The system is described in detail in [14].

There are various proof assistants, namely, *Nqthm* [15], *Nuprl* [16], *Isabelle* [17],*Lego* [18], *Hol* [19], *PVS* [20] and *ACL2* [21]. They are distinguished by different criterions defined by De Bruijn. For further information see [22]. Coq is a proof assistant in which we can express specifications and develop applications that respect these properties. It is being used to develop safe applications in various domains such as algorithms, automata theory, logic and in real-life critical applications. Proofs are built in an interactive way with the aid of automatic search when possible called *tactic*. Coq departs from the proof assistant family by its possibility to generate certified code and relatively efficient functional programs by extracting them from the proofs of their specifications. The functional languages available as output are currently Objective CaML, Haskell and Scheme.

The Coq language is based on the Calculus of Constructions [23], completed with inductive and co-inductive type definitions: inductive types are used to handle finite values with primitive recursion, while co-inductive types are used to express infinite values (such as streams). There is a relation between proofs and programs:

$$\begin{array}{ccc} Term & \Leftrightarrow & Type \\ \Downarrow & & \Downarrow \\ Proof & \Leftrightarrow & Program \end{array}$$

In the underlying logic of Coq, proofs are considered as programs and types as terms. Proving an expression is equivalent to giving a program belonging to this type.

Terms can be built using quantifiers, pattern matching, functional abstractions and applications, as well as many other classical constructions found in functional programming languages.

As types are terms, they have themselves types called sorts. Types can represent programming datatypes or logical propositions. this is expressed using the special sort Set as the type of datatypes and $Prop$ as the type of propositions.

B. Language specification

The idea is to develop the whole code generator within the Coq framework, specify all needed properties and build proofs in order to verify their correctness. Every module of the generator will be implemented and verified using Coq. This approach guarantees a correct extracted code, preserving the correctness properties proved in the Coq source code.

We will focus in this paper on the Scheduler module, which is a first step before any execution or code generation. Giving a block diagram we must prove that all blocks are executed in a correct order. The correct order should be the same as given by Matlab/Simulink.

We have specified the language of block diagrams in the Coq tool and have written a verified scheduler. In this section, we will illustrate our specification of block diagrams.

The primitive functional blocks of Matlab/Simulink are defined as instances of the inductive type `blockop` denoting the block operators. For the sake of conciseness, we consider in our case study a very small subset of the Matlab/Simulink library :

```
Inductive blockop : Set  :=
| InputExt  : nat -> blockop
| OutputExt : nat -> blockop
| Sum       : nat -> blockop
| Mult      : nat -> blockop
| Div       : nat -> blockop
| Delay     :        blockop

⋮

end.
```

The type $blockop$ is of type Set, it introduces all constructors of the considered subset of Matlab/Simulink blocks, such as Sum the constructor of addition operators.

Two particular blocks play a special role in our block diagram specification : $InputExt$ and $OutputExt$ which respectively correspond to the global inputs and the global outputs of the circuit. Most operators take a natural number as an extra parameter, it represents the number of input arguments because this number may vary for a given operator. The delay block always has a single input argument, therefore this number is not mentioned in its type.

Base blocks are defined using the $record$ construction. The $record$ type is an inductive type that represents data clusters, like tuples in mathematics or records in programming languages.

```
Record blockBase : Set :=
{b        : blockop;
inportB  : list evaltype;
outportB : list evaltype;
indexB   : nat}.
```

Each base block gathers 4 fields : a block operator, a list of input types, a list of output types and an integer which is used to index the base block, which makes easier the distinction between duplicated blocks. Notice that nothing is connected to the $InputExt$ block and nothing is connected from the $Output$ block.

First, we define block interfaces, representing a specific input or output ports. They are given as pairs of natural numbers, each pair respectively denoting the block number and the port number for that particular block. Then, we define a connection between a block output and a block input. This connection is defined with 2 interfaces corresponding respectively to the source and to the target interface.

```
Inductive intport (A :evaltype) : Set :=
InternalPort : nat -> nat -> intport A.

Inductive extport (A:evaltype) : Set :=
ExternalPort : nat -> nat -> extport A.

Inductive connexion (A : evaltype): Set :=
CNX : extport A -> intport A -> connexion A.
```

A diagram is defined using a `record` construction with 2 principal fields: the list of circuits that constitute the blocks of the diagram, and the list of all connections of the diagram which describes how blocks are connected to each others.

```
Record diagram (circuit : Set) : Set  :=
{block_list     : list circuit ;
 connexion_list : list (connexion EV) ;
 inportD        : list eval_type ;
 outportD       : list eval_type ;
 indexD         : nat}.
```

Finally, block diagrams are constructed from base blocks or other diagrams. The inductive type definition `circuit` is defined by 2 constructors : `Block` corresponding to base blocks and `Diagram` which allows to recursively define subcircuits.

```
Inductive circuit  : Set :=
| Block   : blockBase       -> circuit
| Diagram : diagram circuit -> circuit.
```

We consider for the rest of the paper the block diagram depicted in figure 1.

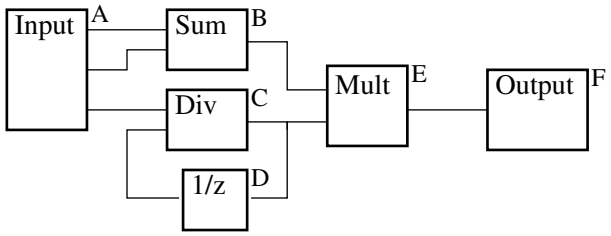

Fig. 1. A sample diagram

An example of how we create the Coq data structure corresponding to this diagram is given in the Annex section.

C. Scheduler Module

As said earlier, we focus on the specification and verification of the scheduler process of the code generator.

The algorithmic specification of the scheduler relies on Environment functions which compute, for each block of the considered diagram, its corresponding rank.

```
Environment : block -> rank
```

The rank of a block can be a numerical value for blocks whose scheduling constraints we already know, or be undefined otherwise. Initially, all blocks have an undefined rank which corresponds to Bottom in the `rank` inductive definition.

```
Inductive rank : Set :=
| Num    :   nat  -> rank
| Bottom : rank.
```

Furthermore, `rank` elements enjoy a (very simple) well-founded partial order structure, provided by the following `lt` ordering relation. Exhibiting such a relation is mandatory in order to prove that our scheduling algorithm terminates.

```
Definition lt (r r' : rank) : Prop :=
match r, r' with
| Bottom, Num _  => True
| _              => False
end.
```

From this strict order, we easily derive `le` as the corresponding non strict order. These orders between ranks are canonically extended to environment functions, with `s` an extra parameter which represents the number of blocks of a given diagram (numbered from 0 to $s-1$).

```
Definition le_Env s (e e' : Environment) :=
forall k, k < s -> le (e k) (e' k).

Definition lt_Env s (e e' : Environment) :=
  le_Env s e e'
/\ exists k, k < s /\ lt (e k) (e' k).
```

In the same example, the initial environment is :

$$Env = \begin{cases} blockA \mapsto \bot \\ blockB \mapsto \bot \\ blockC \mapsto \bot \\ blockD \mapsto \bot \\ blockE \mapsto \bot \\ blockF \mapsto \bot \end{cases}$$

The scheduler takes as input the considered diagram to schedule and the initial environment, and computes a final rank environment, taking into account each scheduling constraint of the circuit. For each recursive call of the scheduler, a new environment is computed; the new environment depends on the previous environments computed until now. This dependency is expressed by the `Forward` function. Roughly speaking, this function, for a diagram `d`, first tests whether the block index `k` is valid or not. Then, in the environment `E`, the ranks of fan-ins (or feeding blocks) of block `k` are considered and the maximum rank is computed. Finally, this rank is incremented before being returned as the new rank of block `k`. Hence, for each block we compute a new rank, possibly different from the one defined in `E`, yielding a new environment.

```
Definition Forward (d : diagram circuit) :
Environment -> Environment :=
fun E k =>
 if (valid_index k d)
 then succ_rank (max_rank E d (Fanins d k))
 else Bot.

Function Scheduler_rec (d : diagram circuit)
(E : Environment) {wf (gt_Env d) E} :=
if (le_Env_dec d (Forward E d) E)
then E else Scheduler_rec d (Forward E d).

Definition Schedule (d : diagram circuit) :=
Schedule_rec d (fun k => Bottom).
```

For the sake of room, we don't detail here the auxiliary functions involved in the definition of `Forward`. The only

important point lies in the way delay blocks are handled. As these blocks often occur in (well formed) feedback loops, their ranks as output devices must be distinguished from their ranks as input devices, whereas other memoryless base blocks don't need such a distinction. The output rank of a delay block is always 0 because the stored value is present at the beginning of every execution cycle. For that purpose, we indeed compute the input rank of a block as a function of the output ranks of its feeding blocks. In this respect, modelling of such a delay operator frequently makes use of two distinct blocks. The scheduler computes environments iteratively through recursive calls, until no new defined ranks are generated. When applying the scheduler algorithm to the previous diagram, we obtain the following environments :

$$Env_1 = \begin{cases} blockA \mapsto 0 \\ blockB \mapsto \bot \\ blockC \mapsto \bot \\ blockD \mapsto 0 \\ blockE \mapsto \bot \\ blockF \mapsto \bot \end{cases} \quad \cdots \quad Env_5 = \begin{cases} blockA \mapsto 0 \\ blockB \mapsto 1 \\ blockC \mapsto 2 \\ blockD \mapsto 3 \\ blockE \mapsto 4 \\ blockF \mapsto 5 \end{cases}$$

D. Correctness of the scheduler

The scheduler module was proved correct with the Coq proof assistant. To establish the termination of the algorithm, we had to define and prove two main theorems required by the Coq proof assistant:

- the first theorem states that each recursive call brings new information about the rank of some block, or equivalently that the `Forward` function is monotone:

```
Theorem Forward_mono : forall E E' d,
   lt_Env E E'
-> lt_Env (Forward d E) (Forward d E').
```

- the second theorem states that bringing new information, as proved by the previous theorem, is a process that cannot go on forever. This amounts to prove that the order relation `gt_Env` is well founded.

```
Theorem gt_Env_wf : forall d,
well_founded (gt_Env (size d)).
```

Beside these theorems, many auxiliary lemmas were defined and proved in order to facilitate the overall scheduler proving process.

The most important part is to prove the adequacy of our proposal with respect to what should provide a scheduling algorithm. We also need two properties:

- each block with a defined rank, has its rank strictly greater than the rank of its fan-ins.

```
Theorem num_rank_spec : forall d b,
let R := Schedule d b in
R b <> Bottom -> forall b',
In b (Fanins d b') -> gt (R b) (R b').
```

- each block with an undefined rank is involved in a cyclic memoryless (i.e. without delay blocks) path in the circuit. This kind of path raises causality issues, is considered harmful and thus forbidden.

```
Theorem bottom_rank_spec : forall d b,
let R := Schedule d b in
R b = Bottom -> In b (BadLoops d).
```

IV. CONCLUSION

Our work aims at developing a certified automatic code generator and we chose for this purpose to use the Coq proof assistant. We divide the code generation into several steps : scheduler, typer, clock calculus, code generation and finally code optimisation. Our approach is inspired by the works of Xavier Leroy in certifying a compiler using the same proof assistant.

This paper reports on a first step, the specification and verification of the scheduler process in Coq.

We have described the input language of the code generator in the Coq tool and we have developed the scheduler algorithm which determines in which order blocks must be executed. The main interest when using Coq is that an implementation, as a piece of Caml code, can be automatically generated without any development effort. This coding phase, when achieved with traditional means, is well known for introducing undesired discrepancies between the specification of a system and its implementation.

Indeed, proving a Coq specification is a tedious task, but far less tedious than directly proving a code generator written in a standard programming language. All the more than, as far as we know, existing such generators were not designed and developed with a forthcoming rigorous correctness proving phase in mind, thus burdening this very proof task.

The scheduler is only a first step towards a complete code generator. The next work is to develop a typing module for our block diagram language. It seems to be more complicated since for instance the Matlab/Simulink library contains base blocks whose instantiations may each belong to a different type. Not only input or output types, but also the number of ports for a given block, may vary.

However, we have developed our scheduler in a quite generic way that will help us in factoring out and reuse several parts of the proofs for the typing phase. This also applies to the clock calculus that we will consider later. Last, we have developed an inlining module that expands a diagram into a flat circuit of simple base blocks. This module was used to detect delay blocks hidden in hierarchical diagrams.

V. ANNEX

```
(*BEGIN SAMPLE CIRCUIT*)
Definition blocA := Block(Build_blockBase
(InputExt 3)(nil)(Int::Int::Int::nil)(0)).
Definition blocB :=Block(Build_blockBase
(Sum 2)(Int::Int::nil)(Int::nil)(1)).
Definition blocC := Block(Build_blockBase
(Div 2)(Int::Int::nil)(Int::nil)(2)).
Definition blocD := Block(Build_blockBase
(Delay)(Int::nil)(Int::nil)(3)).
Definition blocE := Block(Build_blockBase
(Mult 2)(Int::Int::nil)(Int::nil)(4)).
```

```
Definition blocF :=Block(Build_blockBase
(OutputExt 1)(Int::nil)(nil)(5)).

Definition CNX_Input_Sum1 := CNX
(ExternalPort 0 0)(InternalPort 0 1).
Definition CNX_Input_Sum2 := CNX
(ExternalPort 1 0)(InternalPort 1 1).
Definition CNX_Input_Div1 := CNX
(ExternalPort 2 0)(InternalPort 0 2).
Definition CNX_Input_Div2 := CNX
(ExternalPort 0 3)(InternalPort 1 2).
Definition CNX_Sum_Mult1 := CNX
(ExternalPort 0 1)(InternalPort 0 3).
Definition CNX_Div_Mult2 := CNX
(ExternalPort 0 2)(InternalPort EV 1 4).
Definition CNX_Div_Delay := CNX
(ExternalPort 0 2)(InternalPort 0 3).
Definition CNX_Mult_Out:= CNX
(ExternalPort 0 4)(InternalPort 0 5).

Definition diag :=(Build_diagram circuit
(blocA::blocB::blocC::blocD::blocE::nil)
(CNX_Input_Sum1::CNX_Input_Sum2::
CNX_Input_Div1::CNX_Input_Div2::
CNX_Sum_Mult1::CNX_Div_Mult2::
CNX_Div_Delay::CNX_Mult_Out::nil)
(Int::Int::Int::nil)
(Int::nil)
(0)).

Definition sample_circuit :=
Diagram diag.
(*END SAMPLE CIRCUIT*)
```

REFERENCES

[1] X.Leroy, "Formal Certification of a Compiler Back-end or Programming a compiler with a Proof Assistant," *In POPL'06, 33rd Symposium on Principles Of Programming Languages*, Jan. 2006.

[2] A. Tooms, T. Naks, M. Pantel, M. Gandriau, and I. Wati, "Formal Certification of a Compiler Back-end or Programming a compiler with a Proof Assistant," *In ERTS'08, 4th European symposium on Real Time Systems*, Jan. 2008.

[3] G. C. Necula and P. Lee, "Research on proof-carrying code for untrusted-code security," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997, p. 204.

[4] "The Calculus of constructions ," 1999.

[5] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," vol. 1384, pp. 151–166, 1998.

[6] G.-C. Neculae, "Translation Validation for an Optimizing Compiler," pp. 83–95, 2000.

[7] X. Rival, "Symbolic transfer function-based approaches to certified compilation." pp. 1–13, 2004.

[8] Y. F. L. D. Zuck, A. Pnueli and B. Goldberg, "VOC: A translation validator for optimizing compilers," vol. 65, 2002.

[9] J-B.Tristan and X.Leroy, "Formal Verification of translation Validators-A Case Study on Instruction Scheduling Optimizations," *In POPL'08, 35th Symposium on Principles Of Programming Languages*, Jan. 2008.

[10] "Extracting a data flow analyser in constructive logic," vol. 2986, 2004.

[11] "Automated soundness proofs for dataflow analyses and transformations via local rules," 2005.

[12] X. Rival, "Invariant translation-based certification of assembly code," *International Journal on Software and Tools for Technology Trasnfer*, vol. 6, no. 1, pp. 15–37, july 2004.

[13] ——, "Symbolic transfer functions-based approaches to certified compilation," in *31st Symposium on Principles of Programming Languages*, X. Leroy, Ed. ACM, janvier 2004, pp. 1–13.

[14] "Interactive Theorem Proving and Program Development : Coq-Art: The Calculus of Inductive Constructions," 2004.

[15] R. Boyer, M. Kaufmann, and J. Moore, "The boyer-moore theorem prover and its interactive enhancement," 1995. [Online]. Available: citeseer.ist.psu.edu/boyer95boyermoore.html

[16] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Development System*. NJ: Prentice-Hall, 1986. [Online]. Available: citeseer.ist.psu.edu/constable86implementing.html

[17] L.-C. Paulson, "The Isabelle reference manual, Tech. Rep. 283, 1993. [Online]. Available: citeseer.ist.psu.edu/paulson95isabelle.html

[18] R. Pollack, "The theory of LEGO: A proof checker for the extended calculus of constructions," Ph.D. dissertation, 1994. [Online]. Available: citeseer.ist.psu.edu/pollack94theory.html

[19] M-J-C. Gordon, "Mechanizing programming logics in higher-order logic," in *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, G.M. Birtwistle and P.A. Subrahmanyam, Eds. Banff, Canada: Springer-Verlag, Berlin, 1988, pp. 387–439. [Online]. Available: citeseer.ist.psu.edu/gordon88mechanizing.html

[20] S. Owre, N. Shankar, and J. M. Rushby, *User Guide for the PVS Specification and Verification System*. CSL, 1995. [Online]. Available: citeseer.ist.psu.edu/owre93user.html

[21] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Trans. Software Eng.*, vol. 23, no. 4, pp. 203–213, 1997.

[22] "The mathematical language automath, its usage and some of its extentions." vol. 125, 1970.

[23] "The Calculus of constructions," vol. 76, 1988.