Proving Properties of Reactive Programs From C to Lustre

B. Blanc, Loïc Correnson, Zaynah Lea Dargaye, J. Gassino, B. Marre

Proving Properties of Reactive Programs
From C to Lustre

B. Blanc[1], L. Correnson[1], Z. Dargaye[1], J. Gassino[2], and B. Marre[1]

[1] CEA, LIST — *Saclay, France*
[2] IRSN — *Fontenay-aux-Roses, France*

Abstract. In critical embedded software, proving functional properties of programs is a major area where formal methods are applied with an increasing success. Anyway, the more a property is complex, the more a high-level formal model of the software and its environment is required. However, in an industrial setting, such a model is not always available, or cannot be used for independent verification. We propose here a new route, where a high-level Lustre model is extracted from a C source program. Thus, high-level functional properties can be specified in Lustre and proved on this extracted model, hence on the real code, without requiring any additional formal documentation.

Keywords: Formal Methods, Functional and Temporal Properties, Lustre, Scade, Embedded C, Reactive Programs.

1 Introduction

CEA List and IRSN have been working together for a long time to introduce formal methods in the assessment of properties for safety critical software in nuclear power plants ; such methods also apply to many other industrial domains.

However, our experience in using different methods and technologies at various abstraction levels reveals that each one requires its own methods. At low-level, for instance, we have deployed the Frama-C platform [6] for proving properties of embedded C programs. In particular, we can address many properties of critical software at the level of C language: absence of runtime errors during the execution by using abstract interpretation, conformance of elementary functions to their formal specifications using deductive verification, non-interference of system and applications memories, *etc.*

At a higher level of abstraction, we have used the GATeL platform [7, 8] to generate tests covering formally specified objectives from a Lustre or Scade description of the system. This platform can also be used to prove that a required behavior is always achieved. At such an abstraction level, one can address properties like "*if those sensors are activated 80 percents of the time during more than 2 seconds, then this specific alarm shall be raised*".

Such a high-level property can not be formalized *and* proven on low-level programs, for instance by using ACSL [1] language on the source C-code. More

precisely, it is not possible to encode the property in such a way that known and implemented static analysis techniques have any chance to handle the proof. This is due, for instance, to the over-complexity of low-level details of real-life C programs, or to the lack of expressiveness of the specification language.

As a matter of fact, industrial users usually face the following alternative: either prove low-level properties on source programs, or prove high-level ones on abstract models. But, typically at IRSN, there is a specific need for proving *high-level* properties on *low-level* C programs: actually, we want to increase confidence in the program, independently of its development process and artefacts. This verification activity complements standard development processes where high-level desired properties are step-by-step refined into the actual code.

Hence, we decided to design a way to automatically extract, from the low-level C code, an equivalent high-level Lustre model in order to formally prove its high-level properties. This is a completely *new route* that still re-uses all the mature technologies we have already industrially deployed. Section 2 is dedicated to briefly introducing our method and Section 3 presents our case study and the necessary features we need to implement; Section 4 describes the extraction mechanism and, finally, Section 5 reports on our experimental results.

2 Method Overview

We consider a synchronous reactive program, available as C routines that are cyclically invoked by the underlying operating system. Communication with the external environment is achieved through specific memory locations for inputs/outputs, that are written/read by the operating system before/after the cyclic routines. The initial state of the program is prepared by running some initializing routine that can be arbitrarily complex.

Extracting a Model from the Code. The verification method starts with the identification of the memory locations related to the *inputs* and *outputs*. They are designated by the user in terms of dedicated ACSL annotations added into the C code.

From such an annotated code, the Frama-C/Synchrone tool, which we have specially developed for this verification method, automatically analyses the cycling C-code and synthesizes a functionally equivalent Lustre node with as many inputs as observed inputs, and as many outputs as observed outputs. Such a synthesis, we call it *Lustre Extraction*, relies on a combination of static analysis techniques available from the Frama-C platform, as explained with more details in Section 4.

Expressing the Property. The desired properties are specified using the standard and well-known technique of *observers* [5], which has been successfully applied in industrial domains, at AIRBUS for instance [2]. In our context, an observer is a Lustre program that reads *both* the inputs and outputs of the observed program, and yields a single boolean value that becomes `false` whenever one of the desired properties is violated.

Handling the Proof. The extracted Lustre node, which is equivalent to the low-level C program for the considered inputs and outputs, is finally connected to the observer. Although observers can be used for testing or in simulations, we never use those techniques in this experiment. Here, we want to *formally prove* that, for *any* possible inputs, the observer will *never* reach a state where its output is `false`. In our experiments, we used GATeL to perform this final step, but other provers can be used, such as the Kind-2 [3] model checker for instance, or any other Lustre formal verification technique.

The Crux. From the synchronous model point of view, the method is quite standard and well-known. Proving properties by using program models combined with property observers is classical. The difficulty here comes from the fact that we *do not* want to start from an existing Lustre model of the software, but only from its actual C source code. The central part of our contribution is an algorithm for automatically synthesizing, or *extracting* such a model from the source code.

Intuitively, extracting a Lustre model from a cyclic program is quite natural. For instance, consider the overly simple program given Figure 1(a). Consider now the values successively returned by function `read`, and passed to function `write` at each loop iteration, we can model both sequences by the Lustre node given Figure 1(b). Establishing the synchronous equations from the source code is straightforward.

```
int read();
void write(int);
int main(void)
{
  for(int s=0;;) {
    s+=read();
    write(s+1);
  }
}
```

```
node Loop( read : int )
returns ( write : int )
var s,s' : int ;
let
  s' = s + read ;
  write = s' + 1 ;
  s = 0 -> pre s' ;
tel
```

Fig. 1. A simple C program (a) and its Lustre model (b).

In real-life, synthesizing a high-level Lustre model from a low-level C-code is much more complex. Although the intuitive idea sketched above is still valid, in practice we are facing many challenging issues related to actual C programs, that we explore in Section 3 and 4.

3 Case Study

As introduced above, our main experiment is based on a real embedded C-code involved in the safety of nuclear plants, on which IRSN needs to verify high-level (temporal) properties. This huge piece of software (several thousand lines of code) consists of several parts with very different characteristics. Parts related to the operating system, hardware interruptions, real-time management

and process scheduling have been studied by other means, use different language features, and therefore are not in the scope of this experiment.

The functional part we focus on is made of synchronous C routines that are cyclically invoked by the operating system. Those routines are implemented by sequential C code and consists of two kinds of functions: a library of *elementary blocks* and the *cabling code* that flows data from one block to the other ones, each individual block being responsible for a usually simple operation. We illustrate now the main characteristics of these two parts of the cycling routines.

Pointers. Memory values are often obtained via pointers. The only way to handle them soundly is to use address-indexed arrays. This is achieved by re-using the efficient memory models implemented in the Frama-C/WP plug-in. Unfortunately, those arrays can not be translated in Lustre directly. Hence, we need a way to distinguish individual pointed-cells as single variables. This is performed by using the invariants collected by abstract-interpretation *via* the Frama-C/EVA plug-in.

Bitwise Encoding. Many real-life embedded programs use a single machine word to encode several boolean signals in parallel, using one bit of information for each and bitwise logic operators for computations. In high-level models, however, one would expect to find back boolean signals, because provers are generally much more efficient on booleans than on bit-vectors. This is achieved by re-using the simplifications on bitwise operators already provided by Qed and the Frama-C/WP plug-in.

Dead Code Elimination. It is often the case that some parts of the cycling routine are in fact only used during initialisation or during specific maintenance or field-testing modes. Hence, those parts of the code are dead during the cyclic behavior of the application, but not dead during its initial phase. These two specific contexts must be taken into account during the analysis. This is handled by the trace-partitioning facilities offered by the Frama-C/EVA plug-in.

Parameterized Code. A typical synchronous code is built from basic generic blocks that are combined together. However, each block instance is generally parameterized by specific values (delay, threshold, *etc.*), while still sharing the same generic code. Once again, we use trace-partitioning to distinguish those different instances and generate specialized models in turn.

Dynamically Optimized Code. Some cycling sub-routines compute, during their first execution cycle, operational parameters from their specified set of parameters. Then, during further cycles, those operational parameters can be used to simplify their computations. By including a first cycle into the initialisation phase, this optimizing code can be eliminated from the cycling phase. The resulting model is then much simpler and have fewer internal memory states.

Our case study includes all those features, that are present in many reactive programs used in the nuclear field. However, such characteristics are shared with many other domains, and the size and functional complexity of our case study is fully representative of a large panel of embedded programs.

4 Lustre Extraction

We now focus on the synthesis algorithm: provided a C source code and an instrumentation provided by the user, our extraction problem can be reformulated into the following terms: an initialization routine `Init` is executed first, then a routine `Cycle` is repeatedly executed; before each iteration, inputs (l-values) are injected by the operating system into memory; then, after each iteration, outputs (l-values) are available for the operating system.

```
int main(void)
{
    Init();
    while(!error)
    {
      Input();
      Compute();
      Probe();
    }
    return error;
}
```

```
node Model ( A )
returns    ( B );
var M, M' ;
let
  M' = φ(M, A) ;
  B  = ψ(M') ;
  M  = I →pre M' ;
tel
```

Fig. 2. C-Model (a) and Lustre-Model (b)

More generally, from the property validation point of view, the user is interested into injecting arbitrary values as *inputs* and observe any expression or predicate as outputs, that we call *probes*. The *instrumentation* of the code only consists in specifying inputs and probes, and designating the C functions called during initialization and cyclic phase.

Initialization and cyclic routines, inputs and probes constitute the *instrumentation* of the code to be verified. Hence, given such an instrumentation, the actual software to be analyzed has a behavior which is equivalent to the C program described in Figure 2(a).

In actual programs, the main loop is more complex; it may, *e.g.* perform some hardware tests, use interruptions to timely trigger each iteration, *etc.* However, the resulting sequence of global memory states is equivalent regarding the provided instrumentation.

In more formal terms, let us denote by M_t the memory state of the program just before the t^{th} iteration. Thus, M_{t+1} is the memory state after this iteration and is also the memory state just before the $(t+1)^{\text{th}}$ iteration. In particular, M_0 is the memory state just after the initializing routine. At each iteration, we denote by A_t the values read from I/O. We can model the effect of the each loop iteration by the following equations:

$$M_{t+1} = \varphi\left(M_t, A_t\right) \tag{1}$$

where φ is the effect of one single loop iteration on the memory state. The output values B_t can be formalized as a projection of the memory state after each loop iteration, as follows:

$$B_t = \psi\left(M_{t+1}\right) \tag{2}$$

Both Equations 1 and 2 constitute a synchronous system. Our objective is to synthesize a Lustre representation of this system, *ie* a node having stream A as input, stream B as output, and stream M as internal state. Hence, the general structure of such a Lustre model is depicted in Figure 2(b), where I denotes the initial state M_0, and M'_t denotes the memory state M_{t+1}.

More generally, A, B, M and M' are vectors of Lustre variables, to handle multiple inputs, multiple outputs and multiple internal states. The problem at this point is finding a representation of the C memory state which can be encoded with Lustre expressions. For scalar values, one can use Lustre scalar variables and operators, but pointer-addressed values (arrays, structures, and so on) have no counterparts in Lustre. Moreover, there are several ways to formalize C memory states with pointers.

To compute such an abstract model while handling all the low-level C features required by our case study and listed in Section 3, we need to proceed in several steps and re-use complex and mature functionalities from the Frama-C platform. The overall architecture of the extractor is illustrated in Figure 3. It consists in several modules we briefly introduce below.

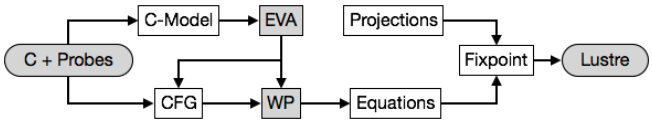

Fig. 3. Architecture of the Extractor

C-Model. From the supplied C code and instrumentation, a C-model of the software is generated. This is a standard C code very similar to the one depicted in Figure 2(a), with additional ACSL annotations to model the injected input values. We can now invoke any Frama-C analyzers on this C code to compute properties and invariants.

EVA. The abstract interpretation plug-in of Frama-C is invoked on the generated C-model. It is an automatic step that verifies the absence of runtime errors, which is a critical feature for the soundness of the extracted model. As a side effect, such a verification is also of great added value for the user. Moreover, the analyzer provides us with invariants on memory locations at each program point, depending on call-contexts. This makes us capable of distinguishing invariants that hold during initialization or during the cycling phase, and to distinguish different instances of identical elementary blocks.

CFG. Aside from the C-model, the cyclic routine is structurally decomposed into a *Control Flow Graph* of elementary operations, including single assignments, `if-then-else` branching, and function calls. During this step, internal loops

(*i.e.* not the main one) are statically unrolled[3] to produce a direct acyclic graph. Dead-code detected during the EVA analysis is also pruned out at this stage.

WP. The generated *Control Flow Graph* is traversed in backward direction using *Weakest Precondition Calculus*. This step intensively uses the Frama-C/WP plug-in. In particular, a dedicated memory model is configured and supplied to the WP in order to compile the transitions of the CFG into oriented equations of the form $m' = \varphi(m)$, where m and m' are memory states at both ends of an edge. Also, invariants synthesized during EVA analysis are injected at this stage, with additional equations. In particular, the initial state I is also computed from the EVA invariants that holds after the initializing phase.

Equations. The set of elementary equations computed by weakest precondition calculus is then flattened into a global system of equations $M' = \varphi(M)$. We use a variant of congruence closure [9] for this algorithm, extended with a dedicated merge operation to take into account the branching nodes of the CFG. This part also takes benefit from the internal simplifications performed by our Qed library [4], used to dramatically simplify expressions and predicates in first-order logic.

Projections. To get rid of theories that occurs in the equations but are difficult to represent in Lustre, we use a general mechanism of *projection*, that allows us to lift a general equation $x' = \varphi(x)$ into a refined equation $y' = \psi(y)$, where $y = \pi(x)$ with π a *projection* and $\pi \circ \varphi = \psi \circ \pi$. We have defined projections for bitwise operators and arrays, but the mechanism is general and can be extended to other theories if needed. The projections are also applied to the initial values of memory states. In our case-study, this features allows us to eliminate all aliased pointer accesses, and to translate all the bitwise operations into purely boolean signals.

Fixpoint. The overall system of equations $M' = \varphi(M)$ is actually a multi-variable set of equations of the form $x'_i = \varphi_i(x_1, \ldots, x_n)$. Probes are computed as equations of the form $p = f(x'_i, \ldots)$. Since x'_i depends on other x_j variables though φ_i, we have to compute the transitive closure of a dependency relation to find the smallest set of variables needed to model all probes. Projections are injected into this set of equations, introducing new variables in the system, but reducing the dependencies after simplification and propagation. This is an iterative process that eventually converges to a fixpoint.

Transcription (Lustre). If the computed fixpoint only contains expressions that can be translated into Lustre, the extraction succeeds. The logical equations are modeled with the Qed library to compute maximal sharing of sub-terms in the

[3] Unrolling is generally feasible in the context of embedded software, since the cyclic routines shall terminate in a predictable amount of time. There are actually several ways in Frama-C to perform unrolling.

system. This allows for a linear translation of the set of equations that would likely be exponential otherwise. Finally, the extracted Lustre node exactly follows the general structure given in Figure 2(b).

Hence, the entire chain follows the intuitive idea sketched out in the overview, thanks to efficient algorithms supported by the available features of the Frama-C platform. The resulting extractor finally scales fairly well on a full-featured case-study of representative size, as discussed in the next section.

5 Experimental Results

The software from which we derived our case-study has been introduced in Section 3. Its code base is huge — more than ten thousands lines of code — although, as we only focus on functional properties, the operating system parts have not been considered.

The entire library of elementary blocks is large, and the software includes many applicative functionalities, spread out over several cycling routines. For our experiments, we have selected a representative subset of these functionalities. Thus, it is not a small experiment, and our case study includes a few thousands lines of C-code comprising all features of Section 3. Details are provided in the table below:

Definitions & Macros	1 759	**Total 4 630** (lines of C)
Elementary Blocks (×14)	2 389	
Functional Diagram	482	

Instrumenting the code was straightforward: this includes the writing of the test harness for the initializing routines as depicted in Section 2, and the specification of the probes of interest.

Regarding the properties, formalized as Lustre observers, we have conducted two complementary experiments: unitary proofs on one side, and functional proofs on the other side. In the unitary proof experiment, we wanted to validate the extractor itself; it consists in small functional diagrams implemented in C code, composed of few blocks. These functional diagrams have been precisely modeled by hand, down to the elementary blocks level, to build a Lustre reference node. Independently, the C code has been processed by our extractor. Finally, we tried, and succeed, to prove that nodes extracted from the C code and the reference ones are in exact bi-simulation for all inputs at every cycle.

On the other experiment, we used the C-code of full-functional diagrams, which typically read switches and sensors, combine them in binary, numerical and temporal functions (memories, delays, thresholds with hysteresis, *etc.*) and finally compute the state of safety outputs. Different computing modes are used, *e.g.* when some inputs have a non-valid status. Properties range from simple ones such as verifying that an alarm is raised when a sensor goes over a threshold, to more complex ones involving sequences of events, memorized states, temporal windows, counters, *etc.* The following figures illustrate the size of our experiments, in lines of code:

Experiments	Source Code	Init Harness	Extracted Model	Property Observers
Unitary Proofs (×11)	2 631	242	185	708
Temporal Properties	4 630	113	91	348
	(lines of C-code)		(lines of Lustre-code)	

The time for extracting models and proving properties is not reported here, but the entire tool chain generally terminates in few seconds, when not immediately.

Interpretations. Several observations can be made from these experiments. First, the size of the generated Lustre models is rather small compared to the initial C code. This is explained by the fact that large parts of the code have been tailored to their sole impact on the observed probes. Two modules of the extractor are responsible for such a slicing effect: first, the EVA analyzer propagates all constants that parameterize each elementary block instance. Moreover, initializing code and dynamic optimizations (computing of operational parameters) are dead-code during the normal behavior of the cyclic routines and they were pruned out during WP. Finally, the projection mechanisms and the final fixpoint of the extractor eliminates all the non-necessary memory states, *i.e.* those that are not involved by the requested probes.

The second main observation concerns the process of writing and proving properties. It turns out that writing functional properties in Lustre is quite efficient for safety systems (indeed, Lustre has been designed for this domain). However, it is sometimes difficult to express properties of large code that are correct at each cycle and for all configurations. Typically, even when the main functional requirement is simple, details about *e.g.* the validity status of the inputs, may complexify it: for instance, one may require to take a default decision, or to maintain the previously computed state, depending on the precise functional requirements. All these details have to be taken into account when formally stating properties. Another approach we have followed, is the specification of partial properties, *e.g.* by partitioning the input state into different typical behaviors, with dedicated properties that are proved separately. During this process, the counter-examples generated by GATeL when a proof can not be established are a key-feature to understand the origin of the mismatch between the C code and the stated property (be it incorrect code or imprecise property).

Finally, when proving properties, we have used all the proof techniques for Lustre programs available in GATeL. Many properties are instantaneously proven by pure simplifications and propagations. Temporal properties usually require reasoning on a large number of cycles. Typically, delays of few seconds correspond to hundreds iterations of the main routine; however, this was smoothly handled by the refutation techniques used by GATeL. Few properties were proved by using k-induction techniques, seamlessly for the user. We found one complex property that required to be cut with an intermediate invariant observing an additional probe on an internal memory location of the code (a timeout counter). We believe that such a modular proof method will be common when increasing the complexity of the software and/or properties.

Plugin Implementation. The extraction plug-in itself is small, thanks to the large re-use of the mature library of standard Frama-C analyzers. Compared to the Frama-C kernel, the Frama-C/EVA and Frama-C/WP plug-ins, each representing about 100 000 lines of OCaml code, the Frama-C/Synchrone plugin only have 3 000 lines of code.

It is interesting to note that, even if the extractor is small, by reusing many powerful building blocks of the Frama-C platform we obtain a very efficient prototype implementing many complex features, which scales on non-trivial actual programs.

6 Conclusion

We propose a new independent formal verification method, mostly automatized, that allows for proving high-level properties on low-level C programs, without relying on any document or model elaborated during the development process. This method may be used in addition to correct-by-design techniques, or when development artifacts are not available, not formalized or when the verification must be performed independently. It also fits reverse engineering purpose, by providing a high-level, yet formally correct, point of view on low-level codes.

References

[1] P. Baudin et al. *ACSL: ANSI/ISO C Specification Language, v1.6*. 2013.

[2] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. "Model checking flight control systems: The Airbus experience". In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009.

[3] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. "The Kind 2 Model Checker". In: 2016.

[4] L. Correnson. "Qed. Computing What Remains to Be Proved". In: *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 – May 1, 2014. Proceedings*. 2014.

[5] N. Halbwachs, F. Lagnier, and P. Raymond. "Synchronous Observers and the Verification of Reactive Systems". In: *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*. 1994.

[6] F. Kirchner et al. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27.3 (2015).

[7] B. Marre and B. Blanc. "Test Selection Strategies for Lustre Descriptions in GATeL". In: *Electr. Notes Theor. Comput. Sci.* 111 (2005).

[8] B. Marre, B. Blanc, P. Mouy, and C. Junke. "GATeL: A V&V Platform for SCADE Models". In: *Formal Methods*. 2013.

[9] R. Nieuwenhuis and A. Oliveras. *Fast Congruence Closure and Extensions*. 2006.