

## Timing Analysis Enhancement for Synchronous Program

Pascal Raymond · Claire Maiza ·  
Catherine Parent-Vigouroux ·  
Fabienne Carrier · Mihail Asavoae

Received: date / Accepted: date

**Abstract** Real-time critical systems can be considered as correct if they compute both *right* and *fast enough*. Functionality aspects (computing right) can be addressed using high level design methods, such as the synchronous approach that provides languages, compilers and verification tools. Real-time aspects (computing fast enough) can be addressed with static timing analysis, that aims at discovering safe bounds on the Worst-Case Execution Time (WCET) of the binary code. In this paper, we aim at improving the estimated WCET in the case where the binary code comes from a high-level synchronous design. The key idea is that some high-level functional properties may imply that some execution paths of the binary code are actually infeasible, and thus, can be removed from the worst-case candidates. In order to automatize the method, we show (1) how to trace semantic information between the high-level design and the executable code, (2) how to use a model-checker to prove infeasibility of some execution paths, and (3) how to integrate such infeasibility information into an existing timing analysis framework. Based on a realistic example, we show that there is a large possible improvement for a reasonable computation time overhead.

**Keywords** WCET · Model-Based Design · Synchronous Languages · Model Checking · Traceability

### 1 Introduction

Hard real-time systems are generally built using model-based design. Particularly, control engineering systems are often generated from synchronous designs. In hard real-time systems, any execution must fulfill timing constraints. To guarantee that

---

This work is supported by the french research fundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001).

---

Grenoble-Alpes University  
Verimag, Centre Équation  
2 avenue de Vignate, 38610 Gières, France  
E-mail: [firstname.lastname@imag.fr](mailto:firstname.lastname@imag.fr)

these constraints are respected, a bound on the Worst-Case Execution Time is necessary.

Static timing analyses aim at estimating this upper-bound on the WCET. They are based on abstractions of the hardware and the software. They generally suffer the necessary over-estimation due to abstraction. The source of this overestimation is twofold: the hardware model may generate over-estimation when joining abstract states, the semantics of the program may generate over-estimation due to the fact that the execution path corresponding to the estimated WCET may be infeasible. In this paper, we aim at improving static WCET analysis when the program under analysis has been generated from a synchronous model. We improve the WCET estimation by reducing the set of feasible paths. We do not focus on the hardware analysis that we consider orthogonal.

WCET analyses are derived on the binary level. When programs are generated from high-level design, they are first translated into an intermediate language (usually C or ADA) then compiled into binary. Due to this two-step compilation, it may appear that most of semantic information is lost: the WCET estimation should be enhanced by considering this high-level semantic information. Two main issues must be solved to integrate these high-level semantic properties: (i) how to extract interesting semantic properties at the high-level, (ii) how to transfer information from one level to the lower next level (traceability).

This paper is structured as follows. First, we introduce the context: WCET analysis, synchronous approach, and why synchronous models are good candidates to enhance WCET estimation. Then, we show on a realistic example, written in Lustre (Caspi et al 2007), that the WCET estimation may be largely refined. This experiment is completed by a section where the example is re-designed using the industrial design tool Scade, in order to illustrate the influence of the high-level language and compiler on the proposed method.

## 2 Context

### 2.1 WCET/Timing Analysis

In this paper, we consider static timing analysis of binary code, which is widely accepted as the most accurate method (versus, e.g. analysis of C code).

*Timing analysis workflow.* Figure 1 shows the general timing analysis workflow used by most of the WCET tools (Wilhelm et al 2008). The first step is to reconstruct the control flow graph (CFG) from the binary code and to establish, in this way, a convenient representation for further semantic extraction phases. The second step works on the CFG and consists of a set of analyses for accessed memory addresses (value analysis), loop bounds (loop bound analysis) and simple infeasible paths (control-flow analysis). These analyses are usually performed at either the binary, or the C level. Note that, in case the C files are used, some traceability analysis is necessary to relate them to the binary code. The CFG, annotated with this semantic and address information, is considered for the micro-architectural analysis and the path analysis.

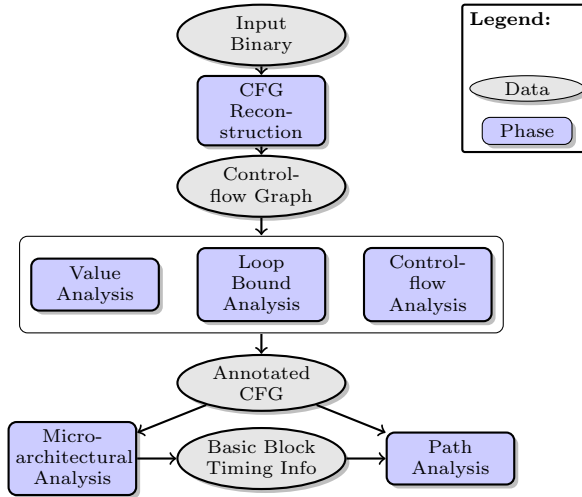


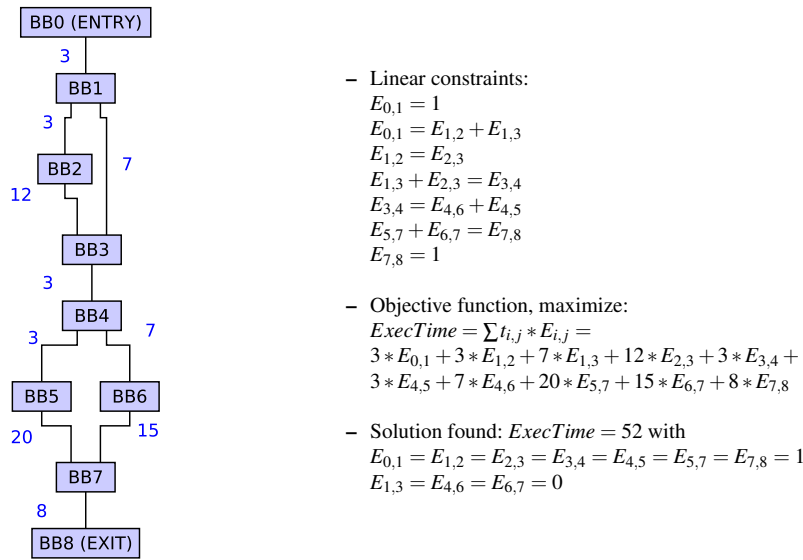
Fig. 1 WCET analysis workflow

The micro-architectural analysis mainly estimates the execution time of basic blocks taking into account the whole architecture of the platform (pipeline, caches, buses,...). Note that, in this paper, we consider mono-processor platforms. The last step, called path analysis, takes as input the CFG annotated with basic block execution time and outputs the WCET estimation. There are several alternative ways to perform path analysis, but the prevalent approach in the WCET analysis community is through solving an integer linear programming problem.

In this paper we are contributing in the path analysis and the value/control-flow analysis. We extract semantic information from design-level (semantic analysis) and add them to the path analysis as integer linear constraints. In the rest of this section, we detail the most popular path analysis: the implicit path enumeration technique. More information about previous steps or different path analyses may be found in state of the art survey papers (Wilhelm et al 2008; Asavoae et al 2013).

*Implicit Path Enumeration Technique.* The Implicit Path Enumeration Technique (IPET) is applied once the micro-architectural analysis has found (local) timing information for each block and/or edge of the binary control flow graph. A simple example of annotated graph is presented in the left-hand side of Figure 2. For the sake of simplicity, the timing information, expressed in processor cycles, is attached to the edges only (the costs of the basic blocks are shifted to their outgoing edges).

The key idea of IPET is to encode the WCET problem into a numerical optimization problem, more precisely an *integer linear programming problem* (referred as ILP in the sequel). For this purpose, an integer variable is introduced for each edge of the graph ( $E_{i,j}$  denotes the edge between block  $i$  and block  $j$ ). The value of a variable  $E_{i,j}$  represents the number of traversals of the node during a complete execution of the program, that is, along a path from the entry to the exit. Note that, for this simple



**Fig. 2** CFG + timing information of a small program, and the corresponding ILP problem.

example, the value of these variables can only be 1 or 0, but this is not the case for programs with loops.

The core of the ILP problem (Figure 2, right) is a set of linear constraints that literally translate the structure of the graph according to the principle that “what enters is what exits”, and that the main entry and exit are traversed once. The linear objective function to maximize is the sum of the edge variables ( $E_{i,j}$ ) weighted by their local time (noted  $t_{i,j}$ ). For this example, an ILP solver will find the solution shown in Figure 2, which corresponds to an execution path passing by the blocks 0, 1, 2, 3, 4, 5, 7, 8, for a total execution time of 52 processor cycles.

Note that, in the case of programs with loops, extra linear constraints are necessary to bound the number of executions of the back-edges (otherwise, the worst case will trivially correspond to a non-terminating execution path). This information is provided by the *Loop Bound Analysis* (Figure 1).

In this paper we use the tool OTAWA<sup>1</sup>. Given a binary executable code, this tool provides the CFG, all the local timing information ( $t_{i,j}$ ), and the corresponding ILP problem (constraints + objective function). The initial set of constraints, also called *structural constraints*, is noted  $ilp_{cfg}$  in the sequel.

<sup>1</sup> <http://www.otawa.fr>

## 2.2 Synchronous programming

The synchronous paradigm was proposed in the early 80's with the aim of making easier the development of safety critical control systems. The main idea is to propose an idealized vision of time and concurrency that helps the programmer to focus on functional concerns (does the program compute right?), while abstracting away real-time concerns (does the system compute fast enough?). The goal of this section is to list the characteristics of synchronous languages that are of interest for timing analysis. For a more general presentation see Halbwachs (1993); Caspi et al (2007).

Semantically, a synchronous program is a reactive state machine: it reacts to its inputs by updating its internal state (internal variables) and producing the corresponding outputs. The sequence of reactions defines a global notion of (discrete) time, often called the basic clock of the program. Reactions are deterministic, which means that the semantics can be captured by a *transition function*  $(s_{k+1}, o_k) = F(s_k, i_k)$ , meaning that at the reaction number  $k$ , both the current outputs  $o_k$  and the next state  $s_{k+1}$  are completely determined by the current inputs  $i_k$  and state  $s_k$ . Moreover, the program must be well initialized: the initial state ( $s_0$ ) is also completely determined.

Practically, synchronous languages are providing features for designing complex programs in a concise and structured way. There are mainly two programming paradigms. In data-flow languages like Lustre, Scade<sup>2</sup> or Signal (Gauthier et al 1987), programs are designed as networks of operators communicating through explicit wires. In control-flow languages, programs are designed as sets of automata communicating via special variables, generally called signals. Examples of control-flow languages are Esterel (Berry and Gonthier 1992), SynchCharts<sup>3</sup>, Scade V6<sup>2</sup>. The style may vary, but the synchronous languages are all based on the same principles:

**Synchronous concurrency:** all the components of a program communicate and react simultaneously;

**Causality checking:** since all communications occur at the same “logical instant”, a causality analysis is necessary to check whether the whole behavior is deterministic or not, in which case the program is rejected. This analysis is based on data-dependency analysis and may be more or less sophisticated depending on the language/compiler;

**Compilation into sequential code:** the basic compilation scheme for all synchronous languages consists in producing purely sequential code from the parallel source design; this generation relies on the causality analysis, which, if it succeeds, guarantees that there exists a computation order compatible with the data dependencies (this principle is often referred as *static scheduling*).

To summarize, the compilation of synchronous programs implements the semantics of the program as defined before: it identifies and generates the *memory* and the *transition function* of the program. Figure 3 illustrates this principle for a data-flow design: the hierarchic concurrent design (a) is compiled into a simple state machine code (b); according to the data dependencies (arrows), a possible sequential code for the function  $F$  is  $H();G()$ , where  $H() = H_1();H_3();H_2()$ .

<sup>2</sup> <http://www.esterel-technologies.com/products/scade-suite/>

<sup>3</sup> <http://www.i3s.unice.fr/map/WEBSPOrts/SyncCharts/>

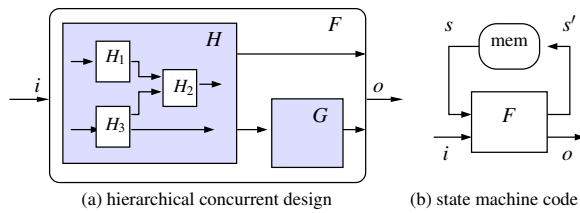


Fig. 3 Synchronous compilation.

The synchronous compiler only produces the transition function, that is, the function that performs a single step of the logical clock. The design of the main program, responsible of iterating the steps, is left to the user. This loop strongly depends on architectural choices and on the underlying operating system. A typical choice consists in embedding the transition function into a periodic task activated on a real-time system clock.

Note that the principles presented in this section are valid for a large class of Model Based Design methods. As soon as a modeling language is equipped with an automatic code generator following the principles of Figure 3, it becomes a “de facto synchronous language”. For instance, it is the case for the well-known Simulink/Stateflow<sup>4</sup> environment, which was originally designed for simulation purpose, and has been completed with code generators.

### 2.3 Timing analysis of Synchronous programs

The synchronous approach permits an orthogonal separation of concerns between functional design and timing analysis. The design and compilation method produces a code (transition function) which is *intrinsically real-time*, in the sense that it guarantees the existence of a WCET bound whatever be the actual execution platform. This property is achieved because the languages are *voluntary restricted* in order to forbid any source of unboundedness: no dynamic allocation, no recursion, no “while” loops. The characteristics of a transition function generated from a synchronous design are the following:

- the control flow graph is structurally simple, mainly made of basic statements and nested conditionals (if, switch);
- synchronous languages allow to declare and manipulate statically bounded arrays, that are naturally implemented using statically bounded *for* loops;
- the code generation may be modular, in which case it contains function calls: the main transition function calls the transition functions of its sub-programs and so on; however this calling tree is bounded and statically known.

Note that, besides these clean and appealing principles, synchronous languages also allow the user to import external code, in which case it becomes his/her responsibility to guarantee the validity of the real-time property.

<sup>4</sup> <http://www.mathworks.fr/products/stateflow/>

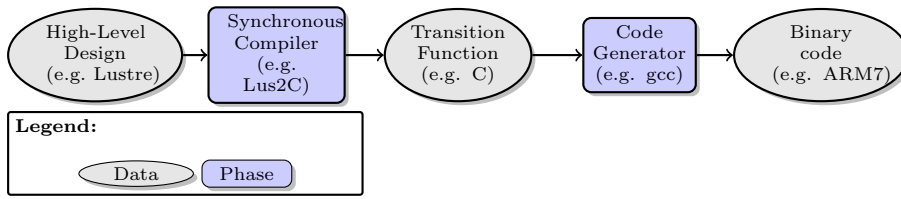


Fig. 4 Typical 2-stage code generation in Model Based Design

The role of timing analysis is then to estimate an actual WCET for a particular platform. Synchronous generated code is particularly favorable for WCET estimation, since it is free of complex features (no heap, complex aliasing, loops, nor recursion). The goal of this paper is to try to go further, by exploiting not only *generic properties* of synchronous programs, but also *functional properties*. We start from the statement that static analysis of high-level synchronous programs has made important progress during the last decades: there exist verification tools (Model-checking, Abstract interpretation) able to discover or verify non-trivial functional properties, in particular *invariants* on the state-space of the program. This statement raises the following question: can such properties be exploited to enhance the timing analysis? Another statement is that such properties are hard or even impossible to discover at the binary level, and thus cannot be handled by existing timing analysis tools.

More precisely, the binary code is obtained via a two-stage compilation. Figure 4 illustrates this typical scheme, which is valid for all synchronous languages, and more generally for most Model-Based Design methods: high-level design is compiled into an “agnostic” general purpose language (C, most of the time), and then compiled for a specific binary platform. We give as example the languages and compilers that are actually used in this work (Lustre and its compiler, C and gcc, and ARM7 as the target platform).

WCET estimation is performed at the binary level, and enhancing the estimation mainly consists in rejecting (pruning) execution paths on the binary. This process raises several problems:

*Control structure.* There must exist conditional branches in the binary, otherwise there is nothing to prune. It supposes that the synchronous compiler actually generates a control structure, that will later be compiled into conditional branches. This fact strongly depends on the high-level language and its compiler. Control flow languages (Esterel, Scade 6) provide explicit control structures that are likely to be mapped into C, and then binary control structures. In data-flow languages, the control structure is implicit: conditional computation is expressed in terms of clock-enable conditions, similarly to what happens in circuit design. The compiler must generate conditional statements, but the whole structure is likely to be less detailed than the one produced from a high level control-flow language. In this paper, we first focus on this less favorable case, by considering an example written in Lustre. We then complete the experiment with a program designed with the more control-flow oriented language Scade 6.

*Non trivial functional properties.* At the high level, we must be able to discover and/or check properties that are hard to find at the binary level. For this purpose, state invariants are clearly good candidates: they are related to the fact that the program state is initialized with a known value, and that it is only modified by successive calls to the transition function. All this information is indeed not available for the WCET analyser.

*Traceability.* Supposing that the compilation produces a suitable control structure, and that we can find interesting functional properties, there remains the technical problem of relating the properties with the binary code branches. This traceability problem is made more complex because of the two stage compilation: if we can expect a relatively simple traceability between Lustre and C, this is not the case between C and the binary code, in particular because of the code optimizations that may be rather intrusive.

All these problems and the proposed solutions are detailed in section 4.

### 3 Related work

Our proposed approach, which exploits high-level semantic information to improve the results of the WCET analysis, is novel with respect to several points. First, to the best of our knowledge, there is no previous work specifically targeting timing analysis enhancements for data-flow languages (SCADE/Lustre). Second, our approach is agnostic to the level of compiler optimizations, and could be applied without modifications, to both unoptimized and optimized code (with the potential loss of precision, in this latter case). Third, our approach is a high-level infeasible analysis: we discover potential sources of infeasibility in the high-level design. Next, we elaborate on how our work compares, on each of these points, with the current state-of-the-art WCET analyses.

As explained in Section 2.3 and in Ringler (2000), Esterel is a control-flow language and for such languages, the timing analysis is easier than for data-flow languages. The main reason lays in how the control structure is transferred between the high-level and the programming level (in general C code). For control-flow languages, the high-level control-flow graph (CFG) and the C level CFG are in an almost one-to-one relationship. Thus, the analysis (which exploits the semantics of the high-level program) may be based on the high-level CFG through some graph traversal analysis and pattern matching (Ringler 2000; Ju et al 2008; Boldt et al 2008).

Andalam et al (2011) and Wang et al (2013) are focusing on a synchronous language, Pret-c, very similar to Esterel. Their approach is specifically dedicated to multi-thread implementations, where concurrency is not solved at compile time. Therefore, the approach is suitable mainly for imperative languages with explicit "fork-like" statements, like Esterel, Reactive-C or Pret-c. In this paper we consider data-flow languages together with the more classical compilation scheme where concurrency is fully solved at compile time (static scheduling). Besides this difference, the kind of properties that are considered to prune infeasible paths is very similar to



the ones considered in this paper: exclusions or implications between the "clocks" that are controlling the executions of (sub) modules.

The existing analyses of Matlab/Simulink programs are also mainly based on the "control-flow" features of this language with some annotations about the execution modes (Tan et al 2009; Kirner et al 2002). All these compute WCET bounds for one tick (as it is the case with our method). However, the analysis of synchronous programs may gain some semantic information by considering in the context of a step the state of the memory at the end of the previous step (Ju et al 2009), the so-called multi-tick analysis. This work is orthogonal to our approach, but as future work, we could extend ours with this "memory" consideration.

Several approaches (Ferdinand et al 2008; Tan et al 2009; Kirner et al 2002) focus on embedding a WCET analyzer into a synchronous design tools. In Ferdinand et al (2008) the AiT WCET tool<sup>5</sup> is integrated in the SCADE suite for analyzing the binary programs generated from SCADE. This work focuses on traceability but does not check the feasibility of estimated worst-case paths according to the high-level semantics.

Our approach applies to both optimized and unoptimized code, as it relies on the debugging information of a standard C compiler - `gcc`. None of the cited works considers optimizations from the C level to binary level. They suppose that there is no optimization and limit their path analysis at the C level while we start from binary level. In general, preserving the predictability of a WCET analysis, through compiler optimizations, is done on a case-by-case basis (Engblom et al 1998; Kirner et al 2010). Recent research leads to WCET-aware compilation (Falk et al 2006) in a certified environment (Blazy et al 2013). Because of its systematic way of generating C code, the synchronous programming paradigm in general, and in our particular case - the Lustre compiler (Halbwachs et al 1991) - are good candidates for the WCET-aware compilation.

Finally, our approach combines a high-level infeasible path detection using model checking, and a path analysis using the usual IPET method. Ju et al (2008, 2009) also rely on the translation of high level properties into classical ILP constraints, but differ on the kind of properties that are handled: they consider particular properties pattern (conflicting pairs) strongly related to the control-flow nature of the considered language (Esterel). Other existing solutions for the infeasible path detection are based on symbolic execution (Knoop et al 2013), abstract interpretation (Gustafsson et al 2006), or model checking (Andalam et al 2011). In several works (Andalam et al 2011; Metzner 2004; Dalsgaard et al 2010; Andalam et al 2011; Béchenec and Cassez 2011) model checking is used both for infeasible path detection and path analysis: the WCET is computed by the model checker itself.

In our framework, we use a model checker dedicated to the Lustre language: Lesar (Raymond 2008). The infeasibility properties discovered by Lesar are encoded as additional ILP constraints, similarly to what is proposed in (Gustafsson et al 2006; Knoop et al 2013). In our combined path analysis, a main novelty concerns the refutation path algorithm: we show that refuting paths one by one, as in (Knoop et al 2013), is not efficient and we introduce a more efficient algorithm.

---

<sup>5</sup> <http://www.absint.com/>

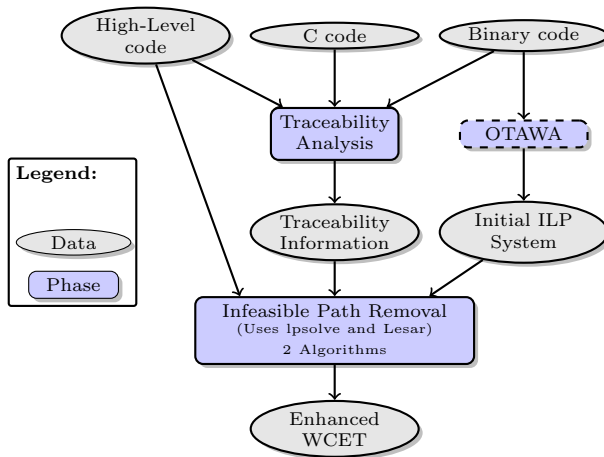


Fig. 5 Proof-of-concept Workflow

#### 4 Proof-of-concept on a realistic example

This section details the methods developed for this study, through the treatment of a typical example. We do not claim to propose a complete “turnkey” solution, but rather a proof of concept that illustrates how, and how much, WCET can be enhanced by exploiting high-level functional properties.

Figure 5 shows the general workflow of the experiment presented in this section. From the source high-level code, the generated C code, and the corresponding binary code, the *Traceability Analysis* tool extracts information relating high-level expressions to binary branches (§ 4.2). In the meanwhile, the external tool OTAWA analyses the binary code and builds the initial WCET problem (expressed as an Integer Linear Programming optimization System). Using the high-level code and the traceability information, the *Infeasible Path Removal* tool tries to refine the initial ILP system in order to obtain an enhanced WCET estimation. This tool provides several algorithms/heuristics and uses the external tools `lp_solve` to solve ILP systems and `Lesar` to model-check properties of the high-level code (§ 4.6).

##### 4.1 Example

The example studied in this section is a simplified version of a typical control engineering application. The overall organization is shown in Figure 6. The program is designed as a hierarchy of concurrent sub-programs dedicated to a particular “task”. The example is made of two concurrent tasks, *A* and *B*, performing treatments on data according to input commands (`onoff`, `toggle`). Tasks are in general not independent: in the example, *B* depends on the value `outA` produced by *A* in order to produce the global output `out`. One role of the high-level compiler is to correctly schedule the code according to these dependencies (§2.2).

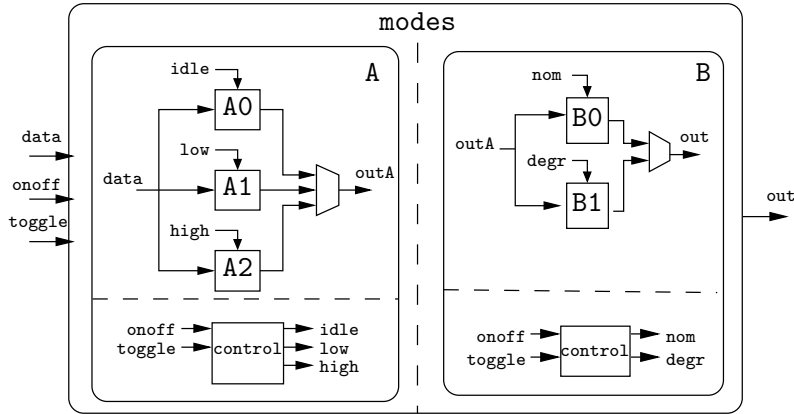


Fig. 6 Typical control system design.

More precisely, the sub-program *A* performs different treatment on its input data depending on different *operating modes*. The data part is kept abstracted, but we gave meaningful names to the control part in order to make their role clearer. The small arrow on the top of a block behaves as a clock-enable: the block is activated (i.e., executed) if and only if the clock is true. In the example, *idle* (resp. *low* and *high*), enables the computation of *A0* (resp. *A1* and *A2*). The clocks are themselves computed by a *control* logic, according to the input commands *onoff* and *toggle*. We don't detail the code corresponding to *control*, but roughly, the command *onoff* switches between *idle* or not, and, when not *idle*, *toggle* switches between *low* and *high* mode. What is really important for the sequel, is that the controller satisfies the following high-level requirement:

**Property 1 (intra-exclusion A)** *If the inputs *onoff* and *toggle* are assumed exclusive, then the program guarantees that *idle*, *low* and *high* are also (pairwise) exclusive.*

Sub-program *B* is similar to *A* except that it has only two operating modes: when *nom* (nominal mode) it computes the function *B0*, and when *degr* (degraded mode) it computes *B1*. Just like for the module *A*, the control logic guarantees the intra-exclusion between the modes of *B*:

**Property 2 (intra-exclusion B)** *If the inputs *onoff* and *toggle* are assumed exclusive, then the program guarantees that *nom* and *degr* are also exclusive.*

Moreover, the control guarantees another important high-level property that relates the modes of *A* and the modes of *B*:

**Property 3 (inter-exclusion)** **B* is necessarily in mode *degr* whenever *A* is not in mode *idle*.*

Finally, this example, while relatively simple, is a good candidate for experimenting since it satisfies high-level properties that are likely to enhance the WCET estimation:

- whatever be the details of the compilation (from Lustre to C, and C to binary), high-level control variables (clock-enables) will certainly be implemented by means of conditional statements,
- high-level relations between clock-enables (exclusivity) are likely to make some execution paths infeasible, and then, the WCET estimation should be enhanced.

For this experiment, the example has been developed in Lustre, in order to use the associated tool chain, mainly the Lustre to C compiler `lus2c`<sup>6</sup> and the Lustre model-checker `Lesar` (Raymond 2008). For generating the executable code, we use a state of the art C compiler, `arm-e1f-gcc 4.4.2`, a widely used cross compiler for the ARM7 platform.

Before considering the whole program, we analyse the WCET of the different modes, in order to foresee which combination of modes is likely to be the most costly. This analysis is made for a non optimized version of the code (option `-O0` of `gcc`) and a strongly optimized one (option `-O2` of `gcc`). The worst cases obtained are:

optim.	A0	A1	A2	B0	B1
-O0	408	730	1226	1130	380
-O2	64	83	128	123	48

From properties 1 and 2 we know that exactly one mode of A and one mode of B must be executed at each step. Moreover, from property 3 we know that certain combinations of modes are forbidden. The table below summarizes the possible combinations:

	A0	A1	A2
B0	yes	no	no
B1	yes	yes	yes

For both optimization levels, it appears that the costs of the combinations  $A2 + B1$  (1606 or 176) and  $A0 + B0$  (1538 or 187) are very close and much higher than any other possible combination. Thus, the worst case is expected for one of these cases.

The rest of this section details the different steps necessary to treat the example. The first problem is to relate high-level variables to the branches in the binary code, referred as the *traceability* problem in the following. Once this traceability problem is solved, the next step consists in developing an automated method for enhancing the WCET estimation (§ 4.6).

## 4.2 Traceability

First of all, we have to define precisely what the traceability problem is: for a given edge in the binary control flow graph (e.g., basic block  $i$  to basic block  $j$ , noted  $E_{i,j}$ ), find, if possible, a Boolean Lustre expression (e.g. `toggle` and `not onoff`), such that the binary branch is taken iff the Lustre expression is true. We have developed a prototype tool to solve this problem, for which we only present here the main principles. Because of the two-stage compilation, the problem requires to trace information at two levels: from Lustre to C, and from C to binary.

<sup>6</sup> <http://www-verimag.imag.fr/The-Lustre-Toolbox.html>

*From Lustre to C.* This problem is easily solved since we control the development of the `lus2c` compiler. The compiler has been adapted in such a way that all `if` statements in the generated C code get the form `if (Lk)` where `Lk` is a local C variable; moreover, a pragma is generated in order to associate this variable to the source expression it comes from. We denote by  $\mathcal{E}(Lk)$  this Lustre expression. The semantics of such a pragma is: the value of the C variable at this particular point of the C program is exactly the value of the Lustre expression. For the sake of simplicity, we no longer make a difference between the C variable and the corresponding expression: we talk about the Lustre condition controlling the C statement.

*From C to binary.* The problem, while mainly technical, can hardly be addressed here in a completely satisfactory way: patching the `gcc` compiler to exactly add the information we need would require an amount of work which is out of the scope of this academic study. We have therefore chosen to rely on the existing debugging features to relate binary code to the C code. Indeed, code optimization may dramatically obfuscate the C control structure within the binary. The retained solution is then not complete but safe: whenever a binary choice can be related, via the debugging information, to a corresponding `if` statement in the C code, we keep this information. Otherwise the choice is simply ignored. While naive, this solution gives good results, despite the relatively intrusive optimizations performed by `gcc` on the control structure.

#### 4.2.1 Traceability without optimization

Without optimization (`-O0` option of `gcc`), as expected, the binary control flow graph strictly maps the C control flow graph.

Figure 7 shows the C CFG (left) and the binary CFG (right) of the example. Both graphs have been simplified in order to outline parts of interest (the actual graphs have more than 70 nodes). Only the branches concerning the calls of interest (`A0`, `A1`, etc) are depicted. Unsurprisingly, a one-to-one correspondence is automatically found by our tool, based on the `gcc` debugging information.

#### 4.2.2 Traceability with optimization

Using optimizations raises problems in critical domains submitted to certification process. For instance in avionics, the highest safety levels of the DO-178B document<sup>7</sup> specify that traceability is mandatory from requirements to all source or executable code. Discussing whether some optimization is reasonable in critical domains is out of scope of this paper: we only aim at experimenting how optimizations may affect traceability, and thus, be an obstacle for enhancing WCET estimation.

Figure 8 shows the C CFG and the binary CFG obtained with `gcc -O2`. The most remarkable effect of the optimization concerns the first part (computation of `A`): the source sequence of 3 conditionals is implemented by a structure of interleaved jumps. In this graph, some branches of binary code get no debugging information at

<sup>7</sup> [http://www.rtca.org/store\\_list.asp](http://www.rtca.org/store_list.asp)

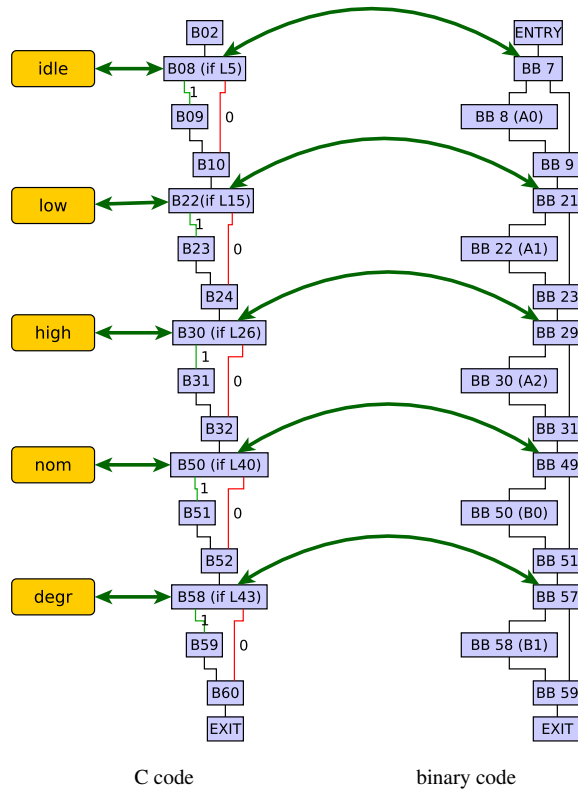


Fig. 7 CFG traceability with -O0

all and thus, cannot be related to the source code (BB1 and BB10). Conversely some source tests have been duplicated in the target code (e.g. BB4 and BB5 are attached to the same source test). Even relatively intrusive, this transformation does not affect traceability: the binary choice is related to its source C condition, but, indeed, several branches may refer to the same source.

#### 4.3 From paths to predicates

At this point, we suppose that traceability analysis has been achieved. The result is a partial function from CFG edges to C literals (either  $L_k$  or  $\neg L_k$ ), and then, to the corresponding Lustre conditions (either  $\mathcal{E}(L_k)$  or  $\neg\mathcal{E}(L_k)$ ).

We denote by  $cond_{i,j}$  the Lustre condition corresponding to  $edge_{i,j}$  (if it exists). Note that, if  $edge_{i,j}$  and  $edge_{i,k}$  are the two edges of a binary choice, if  $cond_{i,j}$  exists, then  $cond_{i,k}$  exists too, and  $cond_{i,j} = \neg cond_{i,k}$ . Here are, for instance, some of the 16 traceability “facts” found on the optimized code of the example (see Figure 8):

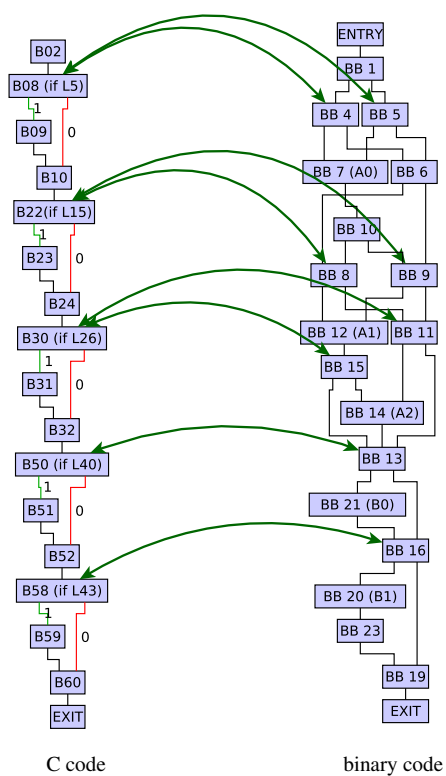


Fig. 8 CFG traceability with -O2

$$\begin{aligned}
 cond_{4,7} &= cond_{5,7} = \mathcal{E}(L5) \\
 cond_{4,6} &= cond_{5,6} = \neg \mathcal{E}(L5) \\
 &\dots \\
 cond_{16,20} &= \mathcal{E}(L43) \\
 cond_{16,19} &= \neg \mathcal{E}(L43)
 \end{aligned}$$

In the sequel, for the sake of simplicity, we extend the traceability function by assigning the condition “true” to any edge that is not related to a Lustre condition. For instance, in the example:  $cond_{entry,1} = cond_{1,4} = cond_{1,5} = \dots = true$ .

#### 4.4 Checking paths feasibility

A path (or more generally a set of paths) in the CFG can be described by a set of edges. Consider the binary CFG in the case -O2 (Fig. 8); the set of edges  $\{edge_{1,4}, edge_{4,7}, edge_{8,12}\}$  corresponds to the set of the complete paths that pass by those 3 edges. In particular, for this set of paths, basic blocks 7 (A0) and 12 (A1) are

both executed. For such a set  $\{edge_{i_k,j_k}\}$ , one can build a Lustre logical expression expressing the feasibility of the paths:  $\bigwedge_k cond_{i_k,j_k}$ .

The idea is then to check whether this Boolean expression is satisfiable according to the knowledge we have on the program. In some cases the answer is trivial: when the expression syntactically contains both a condition  $L_k$  and its negation  $\neg L_k$ , the corresponding edges are *trivially exclusive*. Some of these trivial exclusions are already taken into account as structural constraints: this is obviously the case for conditions guarding the two outgoing branches of a same node (e.g.  $edge_{i,j}$  and  $edge_{i,k}$ ). However, this is not always the case: in the C code, the same condition may be tested several times along the control graph, in which case the logical exclusion is not “hard-coded” in the structure.

When infeasibility is not trivial, it is necessary to use a decision tool. More precisely, we build the predicate  $infeasible = \neg(\bigwedge_k cond_{i_k,j_k})$ , and ask the model-checker (Lesar) to prove that *infeasible* is an invariant of the program. The model-checker (just like any another automatic decision tool) is partial; it may answer:

“yes”: in which case we know that *infeasible* remains true for any execution of the program, and, thus the corresponding paths can be pruned out.

“inconclusive”: in which case *infeasible* may or may not be an invariant.

In the example, the model-checker answers “yes” for the infeasibility predicate  $\neg(cond_{4,7} \wedge cond_{8,12})$ . As expected from the program specification, it means that blocks 7 (call of  $A_0$ ) and 12 (call of  $A_1$ ) are never both executed (Property 1, page 11).

One of the main arguments for the proposed method is that such a property is almost impossible to discover at the binary or C level: this property (intra-exclusion of A modes) is a relatively complex consequence of both assumptions on inputs and dynamics of the underlying state machine:

- inputs `onoff` and `toggle` are assumed exclusive,
- `idle` and `low` are initially exclusive (resp. true and false),
- whatever be an execution of the program satisfying the assumption, the computation of `idle` and `low` ensures that they remain exclusive in any reachable state.

Note that any other decision tool can be used instead of Lesar. As mentioned before, Lesar belongs to the family of model-checkers, which means that it is based on the exploration of the program’s state space. This kind of method is rather powerful, but indeed also very expensive or even intractable in practice if the state space “explodes”. Less expensive methods can be used, that typically ignore the infinite behavior of the program by only considering the execution of a single step. But in our case such methods are unlikely to discover the expected properties (clocks exclusions) because they are actually a consequence of the infinite behavior.

#### 4.5 From infeasibility to linear constraints

In this case study, we limit the use of the discovered properties to the pruning of infeasible paths in the last stage of the WCET estimation, that is, at the ILP level. For ILP solving, we use the tool `lp_solve`<sup>8</sup>.

<sup>8</sup> <http://lpsolve.sourceforge.net/>



Suppose that we have proved  $\neg(\bigwedge_k \text{cond}_{i_k, j_k})$ ; it means that there exists no path passing by all the corresponding edges  $\{\text{edge}_{i_k, j_k}\}$ .

In the ILP system, each edge is associated to a numerical variable representing the number of times the edge is traversed during an execution; the same notation  $\text{edge}_{i_k, j_k}$  is used for these numerical variables (context avoids misleading). This section presents how to translate the logical expression into a numerical constraint.

In the general case (programs with loops), translating the exclusivity of a set of edges  $\{\text{edge}_{i_k, j_k}\}$  in terms of ILP constraints is not trivial (see Raymond (2014) for a general presentation). For instance, when the edges belong to the sequential core of a loop executed at most  $\mu$  time, the exclusivity can be expressed with the following linear constraint:

$$\sum_{k=1}^n \text{edge}_{i_k, j_k} < n \times \mu$$

In the case of loop-free programs (which is the case for our example and, more generally, for any transition function not using arrays), each edge is traversed at most once, and therefore, the translation into ILP constraint is trivial:

$$\sum_{k=1}^n \text{edge}_{i_k, j_k} < n$$

For instance, the proven invariant  $\neg(\text{cond}_{4,7} \wedge \text{cond}_{8,12})$  is translated into:

$$\text{edge}_{4,7} + \text{edge}_{8,12} < 2$$

#### 4.6 Algorithms and strategies

We have seen so far: how to use a model-checker to check infeasibility properties, and how to translate such properties into numerical constraints for the ILP solver. The problem is now to define a complete algorithm for the WCET estimation, that decides which properties to check and when to check them.

In the sequel, we use the following notations:

- “*BinaryAnalysis*(*bprg*)  $\rightarrow$  *ilp<sub>cfg</sub>*” is the abstraction for the main OTAWA procedure that analyses a binary program and returns a whole initial integer linear program (i.e., linear constraints + objective function), a linear constraint is noted *ilc*,
- “*CheckInv*(*lprg*, *lexp*)  $\rightarrow$  *yes/no*” represents the call of the Lesar model-checker; it takes a Lustre program and a Lustre predicate, and returns yes if the predicate is a proven invariant of the program, no otherwise,
- “*LPSolve*(*ilp*)  $\rightarrow$  *wcep*” represents the call of the ILP solver; it returns the worst-case execution path,
- “*ConditionOfPath*(*wcep*)  $\rightarrow$  *lexp*” is the procedure that traces back a binary path to the corresponding conjunction of Lustre literals, as explained in Section 4.4,
- “*ConstraintOfPath*(*wcep*)  $\rightarrow$  *ilc*” is the procedure that takes a binary path and produces the integer linear constraint that will be used to express the infeasibility of the path, as explained in 4.5.

#### 4.6.1 Removing trivial exclusions

A first step, whose computation cost is almost null once the traceability step has been done, consists in taking into account trivial exclusions of the form  $L_n$  and  $\neg L_n$ . As explained in Section 4.4, some of these trivial exclusions are also structural, and thus the constraint is already taken into account in the  $ilp_{cfg}$  system. However other trivial exclusions are not structural: they are due to the Lustre compilation that may open and close the same test several times along the execution paths.

For all pairs of edges with opposite conditions, that are not structurally exclusive, we generate the ILP constraint  $edge_{i,j} + edge_{k,l} < 2$ . The effect of this first enhancement strongly depends on the optimization level:

- In  $-00$  case, traceability identifies 9 conditions: the 5 depicted on the simplified CFG (Fig. 7) plus 4 more. Some conditions are tested several times along the execution path. For instance, one of the extra condition (named M7), and that intuitively controls the initialization, is tested not less than 7 times. Other conditions are tested 4 times (L5, L15 etc). Finally, 102 trivial (but not structural) exclusions are automatically discovered and translated into ILP constraints. The WCET estimation is slightly enhanced: 4718 to 4693 cycles. The enhancement is not impressive, but the number of possible paths is once and for all dramatically reduced: by studying precisely the code, a simple reasoning shows that the number of feasible paths is divided by more than 2 million<sup>9</sup>.
- In  $-02$ , traceability only identifies 6 conditions, the ones outlined on Figure 8, plus the M7 presented above. The fact that 3 conditions from the C code have disappeared in the binary code is due to an optimization related to the target processor capabilities: ARM7 provides a conditional version for most of its basic instructions, and the gcc compiler can then replace simple conditional statements with (even simpler) conditional instructions. Since the resulting binary CFG is much simpler than the one obtained with  $-00$ , the result is less impressive: no trivial exclusion is discovered which is not already a structural exclusion. This first step gives a non-enhanced WCET estimation (758 cycles, which is OTAWA initial estimation).

#### 4.6.2 Iterative refutation algorithm

First, we experiment a *refinement algorithm* which iterates *LPSolve*, to find a candidate worst-case, and *CheckInv* to try to refute it. The pseudo-code of the algorithm is the following:

<sup>9</sup> In the binary code, 27 “if then else” patterns appearing in sequence are controlled by only 6 high-level conditions ; the impact in the number of paths is  $2^6$  when taking into account the trivial exclusions, while it was  $2^{27}$  without this information: the gain is a factor of  $2^{21} = 2.097.152$ .

**Algorithm 4.1:** *RefineWCET(lprg, bprg)*

```

ilp ← BinaryAnalysis(bprg)
while true
do {
  wcep ← LPSolve(ilp)
  lexp ← ConditionOfPath(wcep)
  infeasible ← CheckInv(lprg, ¬lexp)
  if infeasible
  then ilp ← ilp ∪ ConstraintOfPath(wcep)
  else return (ilp)
}

```

The result of this algorithm is optimal modulo the decision procedure: it converges to a worst-case path which is actually *feasible* according to the decision procedure.

The obvious drawback of the method is the number of necessary iterations, which can grow in a combinatorial way. Our example clearly illustrates this problem. Let us consider the  $-00$ , and focus only on the interesting branches (binary CFG in Figure 7). Unsurprisingly, the initial worst-case path found by *LPSolve* corresponds to a case where all modes are executed:

$$\{edge_{7,8}, edge_{21,22}, edge_{29,30}, edge_{49,50}, edge_{57,58}\}$$

This path is refuted by the model checker, and a new numerical constraint is added to the ILP problem:

$$edge_{7,8} + edge_{21,22} + edge_{29,30} + edge_{49,50} + edge_{57,58} < 5$$

In return, *LPSolve* finds another worst case where  $edge_{7,8}$  is replaced by  $edge_{7,9}$ . In other terms, all but the least costly mode are executed (A0). This path is refuted and a new constraint is added:

$$edge_{7,9} + edge_{21,22} + edge_{29,30} + edge_{49,50} + edge_{57,58} < 5$$

The new ILP system leads to another false worst case where all modes but A1 are executed (the second in the cost order), and so on. To summarize: the algorithm first enumerates all the cases where all modes but one are executed, then where all modes but two are executed, and finally all but three. It finally reaches the real worst case where exactly 2 modes (actually A2 and B1) are executed.

In the example, the algorithm behaves slightly differently depending on the code optimization level:

- With  $-00$ , the algorithm converges in 298 steps, from a first estimation of 4693 (after the trivial exclusion removal §4.6.1) to a final optimal estimation of 2371. The worst case path corresponds to one of the expected cases, where modes A2 and B1 are both executed.
- With  $-02$ , the algorithm converges after 116 steps, from an estimation of 758 to an estimation of 457 cycles. Here again, the worst case path corresponds to an expected one, but interestingly, not the same than with  $-00$ : here, the modes A0 and B0 are both executed.

This experiment outlines the combinational cost of the method. This problem arises because the iterative constraints are not precise enough: for instance, a constraint states that at most 4 of 5 edges can be taken, where the “right” information is that they are all *pairwise* exclusive.

### 4.6.3 Searching properties before WCET estimation

In this section, rather than refuting an already obtained worst-case path, we study the possibility of inferring, *a priori*, a set of high-level properties that are likely to help the forthcoming search of the worst-case path.

More precisely, we consider that a set of relevant high-level conditions has been selected. This selection may be greedy (all the identified conditions), or more sophisticated (conditions controlling “big” pieces of code). In our simple example, both heuristics lead to select the same set of interesting conditions: those that control the modes  $\{L5, L15, L26, L40, L43\}$  (Fig. 7 and 8), plus some others depending on the code optimization (4 more in case -O0, 1 more in case -O2).

Useful information on these variables are *disjunctive relations* (or clauses): if one can prove that a disjunction is an invariant (e.g.  $\neg L15 \vee \neg L26 \vee \neg L43$ ), its negation (conjunction) is proved impossible and the corresponding paths can be pruned. We detail now two strategies for discovering properties about the selected conditions.

*A virtual complete method.* Using a model checker, it is theoretically possible to find an “optimal set of invariant clauses” over a set of conditions  $L_k$  (optimality is relative to the decision procedure capabilities). The sketch of the algorithm is the following:

- let  $lprg$  be the considered Lustre program, and  $s$  be its state variables; a model-checker can compute a formula  $Reach(s)$  denoting a superset of the reachable states of the program; in some cases (e.g. finite-state programs) the formula is exact, but in general (e.g. numerical values) the formula is a strict over-approximation;
- let  $L_k$  be the conditions of interest, and  $e_k(s) = \mathcal{E}(L_k)$  the corresponding Lustre expressions; these expressions are functions of the program state variables;
- consider the following formula over the state variables and the conditions:

$$Reach(s) \wedge_{k \in K} (L_k = e_k(s))$$

Quantifying existentially the state variables projects the formula on the  $L_k$  variables only, giving the set of all possible configurations for the  $L_k$  variables:

$$\Phi(L_k) = \exists s \cdot Reach(s) \wedge_{k \in K} (L_k = e_k(s))$$

- formula  $\Phi(L_k)$  can be written in conjunctive normal form (i.e conjunction of clauses):

$$\phi(L_k) = \bigwedge_{i \in I} D_i(L_k)$$

involving a *minimal* number of *minimal clauses* (this is the dual problem of finding a minimal prime implicants cover for disjunctive normal forms);

- each minimal clause can be translated into a minimal infeasibility constraint as explained in Section 4.5.

This algorithm combines several well-known intractable problems (reachability analysis, minimal clause cover) and remains largely virtual.

*A pragmatic pairwise approach.* We now focus on non-complete methods that may/may not find relevant relations.

From a pragmatic point of view, two-variable clauses (i.e., pairwise disjunctive relations) are good candidates: the number of pairs remains relatively reasonable (quadratic), and their analysis is likely to remain reasonable (since they only involve a small part of the whole program).

We detail the algorithm on the example, in the case -02 (cf. Fig. 8). All the conditions identified by the traceability process are selected: the 5 emphasized in Fig. 8 (L5, L15, L26, L40, L43), plus an extra one (M7). Intuitively, this variable controls some initializations; it has a strong influence on functionality, but its influence on the WCET is not expected to be important.

We build and check all the possible pairwise disjunctive relations between these 6 conditions, that is,  $4 \times (6 \times 5) / 2 = 60$  predicates. The proof succeeds for 14 relations. Most of them are reflecting the properties we know from the program specification (see Properties page 11): exclusion between  $B0$  and  $B1$ , pairwise exclusion for  $A0$ ,  $A1$  and  $A2$ , exclusion between  $B0$  and either  $A1$  and  $A2$ . Some others are unexpected, but may be useful for the WCET analysis: intuitively, they relate the initialization and the computing modes.

Finally, the 14 properties are translated into 47 ILP constraints (remember that the same condition may control several branches). With this additional constraints, *LPSolve* directly gives the same worst-case path and time obtained with the iterative algorithm.

To summarize, this pragmatic method gives the same result with only 60 relatively cheap calls to Lesar, and a single call to *LPSolve*, where the *a priori* smarter refutation method requires 116 relatively costly calls to the model-checker and as many calls to *LPSolve*.

In case -00, 9 atomic conditions are identified, leading to 144 binary disjunctions to check; 21 are proved invariant by Lesar, and translated into 396 ILP constraints. The final estimation is 2372, and misses the optimal estimation for 1 cycle. This is due to the fact that the path obtained is still not feasible: this is the drawback of the method which only consider pairwise properties and may miss more complex relations. However, if and when optimality is the goal, the algorithm can be completed with one (or more) steps of the iterative method.

## 4.7 Results

This section summarizes some relevant quantitative information on the case study. The results presented here, for a single program, have obviously no "statistical" meaning, and they are not intended to illustrate the efficiency of the method. They mainly give information on how the method behaves, in particular in terms of overhead computation cost.

*Size and complexity of the example.* Table 1 give some quantitative results on the experiment that do not depend on the method (iterative/pairwise). They are given for the two versions of the binary code (non optimized and optimized). The complexity

Bin Code		WCET estimation		ILP constraints (#/arity)		OTAWA cost (cpu seconds)
opt.	# BB/edg.	initial	first	structural	trivial	
-00	83/119	4718	4693	178 / 2	102 / 2	64s
-02	24/42	758	758	60 / 3	6 / 2	1s

**Table 1** Results that do not depend on the pruning method

of the program can be figured out from its CFG size (in number of blocks and edges). The initial WCET estimation comes from the (unique) call to OTAWA, and corresponds to the initial (structural) ILP system. For ILP constraints, the complexity can be figured out with both their number and their arity (maximal size of one constraint, given in number of variables).

The first WCET estimation is the one obtained by removing the trivial exclusions: it corresponds to an ILP system where the constraints referred as *trivial* are added to the structural ones. The overhead due to the trivial exclusion removal is neglectable and not represented in the Table (less than 0.1 cpu seconds).

The cost of the (unique) call to OTAWA is given. One can note that this cost seems to grows exponentially with the size of the CFG: 1s for -02 code, 64s for the -00 code which is only 3 times bigger.

optim.	Wcet estimation		# iter.	ILP constraints	cost (cpu seconds)		
	initial	final			LPSolve	Lesar	Over. (Total)
-00	4718	2371	298	298/30	90s	5.2s	99s (163s)
-02	758	457	116	116/6	1.5s	0.7s	3.5s (4.5s)

**Table 2** Iterative method: gain is about 50% with -00 code, 40% with -02 code

*Iterative method.* Table 2 give quantitative results for the iterative method. The initial WCET estimation is recalled, and the final estimation shows an improvement of about 50 % or 40 % depending on the optimization level. The key information is the number of iterations to reach the optimal WCET: it corresponds to the number of calls of both LPSolve and Lesar. The computation time overhead is mainly due to the numerous calls of LPSolve, with ILP systems that are relatively complex: the maximum arity (e.g. 30 variables in -00 case) is a good information of the complexity.

optim.	Wcet estimation		Lesar (# pairs)		ILP constr. (#/arity)	cost (cpu seconds)		
	first	final	checked	valid		LPSolve	Lesar	Over. (Total)
-00	4718	2372	144	21	396/2	0.1s	2.2s	3s (67s)
-02	758	457	60	14	47/2	0.1s	1s	1.3s (2.3s)

**Table 3** Pairwise method: gain is almost the same, but with a much reasonable computation cost

*Pairwise method.* Table 3 summarizes the results for the pairwise exclusion method. The improvement is similar to the one obtained with the iterative method. Consider the -00 line: the main information is the number of pairwise exclusion checked by Lesar (144), the number of those that are proved valid (21), and the number of constraints they imply for the ILP system (396). In this case, a single extra call to LPSolve is necessary, and then the corresponding overhead is neglectable. The cost of Lesar is more important, but remains reasonable, comparing to the (unavoidable) cost of OTAWA.

*Conclusion.* We can reasonably conclude, by comparing the cases -02 and -00, that the iterative algorithm is unlikely to scale up for bigger codes. From the -02 code to the -00 code, the size grows roughly 3 times; in the same time, both the number of iterations and the total cost of Lesar seem to grow linearly (factor between 2 and 3), but the total cost of LPSolve dramatically explode (multiplied by 60). More than the increasing number of constraints, the increasing of their complexity explains this explosion (298 constraints, each involving 30 variables).

On the contrary, the pairwise exclusion method behaves in a very promising way: when considering only the overhead due to our method, and not the cost of OTAWA, the cost seems to grow linearly.

## 5 Note on the influence of the high-level design and compiler

As mentioned earlier, there are many high-level design tools, with different programming styles and compilation methods. In order to complete the experiment, we illustrate here how the high-level design language influences the method. For this purpose, we switch from the purely data-flow language Lustre to Scade 6, which proposes a mix of data-flow and control-flow styles.

*Control-flow oriented design.* Figure 9 shows a Scade 6 design semantically equivalent to our example program of Figure 6. Instead of implementing the “modes” in terms of clock enable (which is the only way in data-flow style), we use here the explicit state machine feature provided by Scade. For instance, module A is designed as a 3-states machine, each state corresponding to a particular computation mode ; the `control` part of A is not necessary: the mode selection is directly encoded in the transitions. The design of module B is similar, except that we keep an auxiliary module for computing the local conditions `sleep` and `wake`.

*Control-flow compilation.* The interest of such a design is that the “intra-exclusion” of modes is made explicit at the high level (exactly one state of each machine is executed at each step), and we can expect that the compilation will propagate these exclusions at the C and binary levels. This is actually what happens when using the Scade compiler (KCG), as depicted in Figure 10. This figure shows a small part of the control flow graphs of the C code (left) and the corresponding binary code (right), the whole size is 100 blocks for the C CFG and 127 for the binary CFG. As expected, the intra-exclusion properties (Properties 1 and 2, page 11) are “built-in” in the structure of binary CFG:

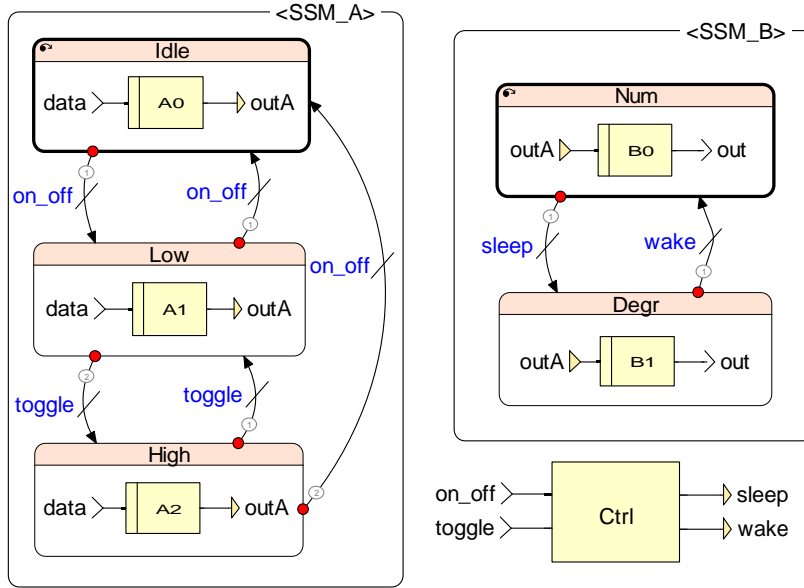


Fig. 9 Scade 6 design using explicit state machines

- there is no path between the basic blocks 50 (A0), 45 (A1) and 53 (A2) (top of the CFG),
- similarly for basic blocks 58 (B0) and 60 (B1) (bottom of the CFG).

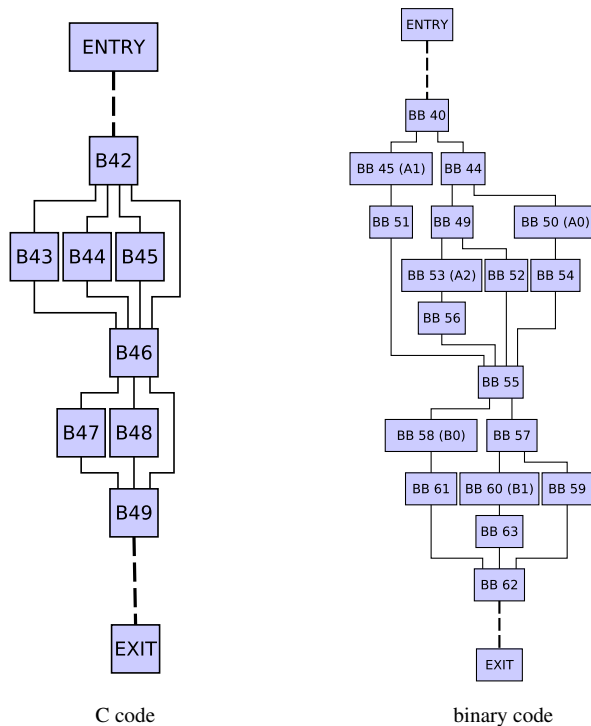
However, non structural properties (e.g. inter-exclusion, Property 3, page 11) are still likely to enhance the WCET.

optim	#BB	WCET estimation		Improvement
		initial	final	
-00	149	2971	2235	24.8 %
-02	67	432	383	11.4 %

Table 4 Experiment with the Scade design: quantitative results

*Quantitative results.* Table 4 summarizes some quantitative results. Conversely to the experiment made with Lustre, this experiment is not fully automatic: because KCG is a proprietary tool, and because we do not have a model-checker suitable for Scade, both the traceability and the introduction of semantic properties have been made by hand. As a consequence, we cannot present here relevant results for the computation cost, but only for the quality of the result. As expected, the improvement





**Fig. 10** Simplified C and binary CFG's obtained from the Scade 6 design of Figure 9

is less important than with Lustre. However, the gain, which is mainly due to the inter-exclusion property, is still interesting: about 25% or 11% depending on the binary code optimization.

## 6 Conclusion

In this paper, we propose a method to improve timing analysis of programs generated from high-level synchronous designs. The main idea is to take benefit from semantic information that are known at the design level and may be lost by the compilation steps (high-level to C, C to binary). For that purpose, we use an existing model-checker for verifying the feasibility of execution paths, according to high-level design semantics. Furthermore, our approach may work on optimized code (from C to binary). We introduced the approach on a realistic example and showed that there is a huge possible improvement, with a reasonable overhead compared to the (unavoidable) cost of the timing analysis of the binary code.

The possible extensions of this work are numerous: taking into account the features provided by a particular high-level language; studying clever heuristics for discovering relevant high-level properties; taking into account discovered infeasible

paths in hardware analysis. A long term aim is to study the possibility of a complete WCET-aware compilation chain, where the tradeoff to find not only concerns the code size and efficiency, but also the traceability of high-level semantic properties.

## References

- Andalam S, Roop P, Girault A (2011) Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs. In: International Conference on Design, Automation and Test in Europe (DATE 2011)
- Asavaoae M, Maiza C, Raymond P (2013) Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective. In: 13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013), pp 31–40
- Béchenec JL, Cassez F (2011) Computation of WCET using Program Slicing and Real-Time Model-Checking. The Computing Research Repository (CoRR) abs/1105.1633
- Berry G, Gonthier G (1992) The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci Comput Program (SCP)* 19(2):87–152
- Blazy S, Maroneze A, Pichardie D (2013) Formal Verification of Loop Bound Estimation for WCET Analysis. In: VSTTE - Verified Software: Theories, Tools and Experiments, Springer, vol 8164, pp 281–303
- Boldt M, Traulsen C, von Hanxleden R (2008) Worst Case Reaction Time Analysis of Concurrent Reactive Programs. *Electronic Notes in Theoretical Computer Science (ENTCS)* 203(4):65–79
- Caspi P, Raymond P, Tripakis S (2007) Synchronous Programming. In: Lee I, Leung JYT, Son SH (eds) *Handbook of Real-Time and Embedded Systems*, Chapman and Hall/CRC, chap 14
- Dalsgaard A, Olesen M, Toft M, Hansen R, Larsen K (2010) METAMOC: Modular Execution Time Analysis using Model Checking. In: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pp 113–123
- Engblom J, Ermedahl A, Altenbernd P (1998) Facilitating worst-case execution time analysis for optimized code. In: *Euromicro Conference on Real-Time Systems (ECRTS)*
- Falk H, Lokuciejewski P, Theiling H (2006) Design of a WCET-Aware C Compiler. In: 6th International Workshop on Worst-Case Execution Time Analysis (WCET 2006)
- Ferdinand C, Heckmann R, Sergent TL, Lopes D, Martin B, Fornari X, Martin F (2008) Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In: *International Conference on Embedded Real-Time Software and Systems (ERTS2)*
- Gauthier T, Guernic PL, Besnard L (1987) Signal, a declarative language for synchronous programming of real-time systems. In: *Proc. 3rd. Conference on Functional Programming Languages and Computer Architecture, LNCS 274*, Springer Verlag
- Gustafsson J, Ermedahl A, Sandberg C, Lisper B (2006) Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In: *IEEE Real-Time Systems Symposium (RTSS)*
- Halbwachs N (1993) *Synchronous programming of reactive systems*. Kluwer Academic Pub.
- Halbwachs N, Raymond P, Ratel C (1991) Generating Efficient Code From Data-Flow Programs. In: *3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, pp 207–218
- Ju L, Huynh BK, Roychoudhury A, Chakraborty S (2008) Performance debugging of Esterel specifications. In: *International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*
- Ju L, Huynh BK, Chakraborty S, Roychoudhury A (2009) Context-sensitive timing analysis of Esterel programs. In: *Proceedings of the 46th Annual Design Automation Conference (DAC 09)*, pp 870–873
- Kirner R, Lang R, Freiberger G, Puschner P (2002) Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In: *Euromicro Conference on Real-Time Systems (ECRTS)*
- Kirner R, Puschner P, Prantl A (2010) Transforming flow information during code optimization for timing analysis. *Journal on Real-Time Systems* 45(1-2)
- Knoop J, Kovács L, Zwirchmayr J (2013) WCET squeezing: on-demand feasibility refinement for proven precise WCET-bounds. In: *21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pp 161–170
- Metzner A (2004) Why Model Checking Can Improve WCET Analysis. In: *16th International Conference on Computer Aided Verification (CAV 2004)*, pp 334–347

- 
- Raymond P (2008) Synchronous Program Verification with Lustre/Lesar. In: Mertz S, Navet N (eds) Modeling and Verification of Real-Time Systems, ISTE/Wiley, chap 6
- Raymond P (2014) A General Approach for Expressing Infeasibility in Implicit Path Enumeration Technique. In: International Conference on Embedded Software (EMSOFT)
- Ringler T (2000) Static Worst-Case Execution Time Analysis of Synchronous Programs. In: Procs. of the 5th Ada-Europe International Conference on Reliable Software Technologies, pp 56–68
- Tan L, Wachter B, Lucas P, Wilhelm R (2009) Improving Timing Analysis for Matlab Simulink/Stateflow. In: 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)
- Wang JJ, Roop PS, Andalam S (2013) ILPc: A novel approach for scalable timing analysis of synchronous programs. In: International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2013), pp 1–10
- Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7(3)