Chapter 1

OPTIMIZATIONS FOR FASTER EXECUTION OF ESTEREL PROGRAMS

Dumitru Potop-Butucaru *

IRISA, Campus de Beaulieu, 35042 Rennes, France

Dumitru.Potop@irisa.fr

Robert de Simone

INRIA, 2004 rte. des Lucioles, 06902 Sophia Antipolis Cedex, France

Robert.De_Simone@inria.fr

Abstract The fine-grained parallelism and the need for determinism are traditional issues in the design of real-time embedded software. In addition, the increasing complexity of the specifications requires an increasing use of higher level formalisms. The Esterel language offers natural solutions to all these problems, but its compilation proved challenging, so that efficient compilation techniques have only recently been proposed. Consisting essentially in direct simulation of the reactive primitives of the language, these techniques now need to be accommodated with traditional issues of Esterel: the definition of formal semantics, the constructive causality, and the design of analysis and optimization methods that are both efficient and correct.

We address these problems by defining a new intermediate model, called GRC, for the representation of Esterel programs. The GRC representation preserves much of the initial program structure while making the control flow pattern and the hierarchical state structure explicit. It fully complies with the semantics of Esterel, and, most important, it is a good support for efficient analysis, optimization, and code generation algorithms based on static analysis.

The use of a GRC-based approach results in significant efficiency and generality gains. The approach could be extended to the hardware compilation of Esterel and to the compilation of other synchronous languages with fine-grained parallelism.

*Partly supported by the ARTIST European project

Keywords: Causality, code generation, compiler, embedded system, Esterel, intermediate representation, optimization, static analysis, synchronous.

1. Introduction

Sequential languages like C and assembly are still widely used in the development of reactive real-time embedded systems, even for applications that are most naturally described as concurrent systems. Real-time operating systems (RTOS) are used in these cases to schedule the resulting sequential processes, thus providing the needed concurrency. Two problems arise from here. The first is that a RTOS can be unpredictable, making functional and timing verification difficult. The second is that RTOS-level scheduling can be very inefficient for applications involving fine-grained parallelism, due to context switching overhead.

The *synchronous approach* [BB91, Hal93, BCE$^+$03] uses classical hardware concepts in order to provide deterministic concurrency and better verification capabilities. The execution of a synchronous system is cyclic, driven by a *global clock*. At each clock cycle the system reads the inputs, computes its *reaction*, and outputs the result. The code executed during a clock cycle is *loop-free* and the clock is operated by the system itself, allowing precise timing control. The underlying synchronous model also allows the use of the well-studied functional verification techniques developed for synchronous circuits, so that the approach is a good basis for the development of safety-critical embedded software.

On the other hand, the high semantic standards behind the *synchronous languages* (*e.g.* Esterel [BG92], Lustre [HCRP91], Signal [LGLL91]) made their correct and efficient implementation a challenging task. A particular difficulty is here that all the informations which might affect a given variable at a given clock cycle must be compiled *before* the variable is used. A typical example is the *signal absence*, which can only be decided after negative information about all potential emissions has been propagated. The fine-grained parallelism and the intricate instruction ordering make run-time scheduling inefficient, due to the context switching overhead. In consequence, implementation code is often generated by compiling away all internal concurrency[1], while proceeding in a single-loop periodic evaluation of all the instructions of the program. The synchronous specification is transformed into a single sequential (C) *reaction function* where all the operations have been *statically scheduled* using auxiliary variables and tests. A small run-time executive interfaces with the asynchronous environment to read the inputs and decide when to call the reaction function.

[1] In distributed, multi-processor implementations like the one proposed by Caspi *et al.* [CGP99], some concurrency is preserved between specification parts operating on different processors

A program written in a concurrent synchronous language like Esterel and its desired sequential implementation are very different objects. For this reason, all Esterel compilers use intermediate representations, and each intermediate representation determines in turn the capabilities of the associated compiling technique. The first implementations of Esterel [BG92, Est00] used mathematical models (Mealy machines and digital circuits) as intermediate representations. Consequently, the resulting C code was either needlessly large, or slow. More recent compilers [Edw02, WBC+00] use control-flow graphs that are easily translated into well-structured C code, but are too far from Esterel to represent its exact semantics or to support a number of efficient analysis and optimizations.

This paper presents a new approach to the compilation of the Esterel language which achieves significant performance and generality improvements by using the GRC (*GRaph Code*) intermediate representation.

The main contribution of the paper is the definition of the GRC code, which uses two graph-based structures – a *hierarchical state representation* (HSR) and a *concurrent control-flow graph* (CCFG) – to preserve most of the structural information of the Esterel program while making the control flow explicit with few graph-building primitive nodes. The hierarchical state can be seen as a structured data memory that preserves state information across reactions. In each instant, a set of activation conditions (triggers) is computed from this memory state, to drive the execution towards active instructions (for historical reasons such activation triggers are generally called *clocks* in reactive synchronous terminology). Because of the formal synchronous semantics these clocks enjoy nice properties (a computation is only triggered when its outcome will be consumed, and vice-versa). In the current GRC scheme, the computation of *activation clocks* from the current state has to be performed first thing in each reaction. But then the actual data instruction blocks will only be activated at proper pace.

On the new format, we introduce several transformations, mostly semantics-preserving optimization algorithms based on static analysis methods. Due to the mathematical nature of the GRC model these optimization can be formally proved sound, although we shall not insist on this aspect in the current paper. Benchmarks indicate the strong interest of the approach in terms of potential optimizing gains. The overall compilation scheme can be seen either as an attempt to produce efficient software code, or efficient simulation code for hardware representations (a goal equally followed by the simulation semantics of VHDL and Verilog hardware description languages, but without our strong synchronous semantics).

We focus here on the generation of software simulation code by defining a GRC-to-C translation scheme that handles all *GRC-acyclic* and *circuit-acyclic* Esterel programs. We shall disregard here the topics of simulating cyclic GRC

specifications, or of determining program correctness in the sense of constructive causality at GRC level. Advances in that direction can be found in [PB02]. The efficiency of our approach is supported by benchmarks comparing our GRC-based compiler `grc2c` to existing compilers.

The paper is organized as follows. Section 2 is devoted to an informal presentation of the Esterel language, including the description of the small example which will be used throughout the paper. A review of the existing Esterel compilers is given in section 3. The next three sections describe our new approach. In section 4 we define the GRC intermediate representation: its primitives, their semantics, and the Esterel translation to GRC. We also insist here on some causal correctness (acyclicity) issues relating GRC and the digital circuits. Section 5 presents the GRC-level optimization algorithms that are currently used in our compiler. The software code generation technique, based on efficient state encoding and static scheduling, is described in section 6. Section 7 presents experimental results comparing `grc2c` with existing compilers, and we conclude with some suggestions about how to extend this work.

2. The Esterel language

Esterel [BG92, Est00, BdS91] is a design language for the representation of reactive real-time embedded systems. It has a full-fledged formal operational semantics [Ber99, PB02] which solves tricky modeling issues such as the proper definition of global synchronous reactive behaviors and the constructive causality in control flow propagation. Esterel specifications can be translated into sequential software code, digital synchronous circuits, or a combination of both, but our work only concerns its software compilation.

Through its intuitive, highly-readable imperative style, Esterel tries to combine strong modeling features together with efficient compilation purposes. As illustrated by the small example of fig. 1.1, Esterel features the control flow operators of a language like C (sequence, loop), but also provides concurrency, preemption, and a synchronous, clock-driven execution model. In each clock cycle the inputs are read, and the program computes its *reaction* by resuming the control threads and running them until suspension. The communication is done through *broadcast signals* which are either *present* or *absent* in each clock cycle. The signal `S` is present if a statement “`emit S`” is executed during the cycle, and absent otherwise.

Our small example has four input signals and one output signal. Meant to model a cyclic computation like a communication protocol, the core of our example is the loop which awaits the input `I`, emits `O`, and then awaits `J` before instantly restarting. The local signal `END` signals the completion of loop cycles.

```
module Example: input I,J,KILL,SUSP; output O;
suspend
  trap T in %exception handler, performs the preemption
    signal END in
      loop %basic computation loop
        await I;emit O;await J;emit END
      end
    ||
      %preemption protocol, triggered by KILL
      await KILL;await END;exit T
    end
  end;
when SUSP %suspend signal
end module
```

Figure 1.1. A simple Esterel program modeling a cyclic computation (like a communication protocol) which can be interrupted between cycles and which can be suspended

When started, the `await` statement waits for the *next* clock cycle where its signal is present. The computation of all the other statements present in our example is performed during a single clock cycle, so that the `await` statements are the only places where control can be suspended between reactions (they preserve the *state* of the program between cycles). A direct consequence is that the signals `I` and `J` must come in different clock cycles in order not to be discarded.

clock	inputs	outputs	comments
0	any		all inputs discarded
1	I	O	
2	KILL		preemption protocol triggered
3			nothing happens
4	J,SUSP		suspend, J discarded
5	J		END emitted, T raised, program terminates

Figure 1.2. A possible execution trace for our example

The loop is preempted by the exception handling statement `trap` when "`exit T`" is executed. In this case, `trap` instantly terminates, control is given in sequence, and the program terminates. The preemption protocol is triggered by the input signal `KILL`, but the exception `T` is raised only when `END` is emitted. The program is suspended – no computation is performed and the state is kept unchanged – in clock cycles where the `SUSP` signal is received. A possible execution trace for our program is given in fig. 1.2.

The operations that compose a reaction are *causally ordered* by the control flow and by the signal communication. A signal `S` can be tested in a clock

cycle only after its status has been determined. The signal becomes present as soon as it has been emitted. It becomes absent when we can determine that no "`emit S`" statement can be executed during the current clock cycle. The signal causality may lead to deadlocks in the computation of a reaction despite the fact that the code of a reaction is loop-free. Programs that can deadlock (*e.g.* `present T then emit T end`) are incorrect and must be rejected at compilation.

Determining whether a program can deadlock or not is not easy. The semantics of Esterel is *constructive*, in the sense that the absence of signals must be determined without any kind of guess. To do so, one has to notice that according to decisions already made in the current reaction all the emissions of the signal are invalidated. The recursive fact propagation is exemplified with the following (correct) statement:

```
emit S;
present T then
  present S else emit T end
end
```

When the statement is started, the signal `S` is emitted and the control is blocked on the test on signal `T`, whose status is still unknown. The *present* status of `S` is then propagated into the not yet executed code. There, it invalidates the `else` branch of the test on `S`, so that `T` becomes *absent*, and execution can proceed. Note that our execution involved the evaluation of code (the test on `S`) that will never be executed.

Due to its constructive semantics, Esterel has a natural semantic model as Synchronous Digital Circuits, at schematic gate level, endowed with intuitionistic constructive semantics [Ber99]. In turn, such circuits have a natural operational interpretation in terms of Finite State Mealy Machines. Thus, all Esterel modeling primitives are provided meaning as hierarchical constructs allowing to build compound objects in each of these domains. Actually, this goes for reactive control flow structuring primitives, while the language also offers data handling of a classical imperative nature (assignments).

3. Related work

Four approaches have been proposed for the compilation of Esterel into sequential code. Historically the first, **the automata-based approach** [BG92, Bre02] basically follows the Structural Operational Semantics of the language to produce a *flat, global automaton* that is then encoded in C. Performed by means of exhaustive symbolic simulation, the automaton expansion also determines the causal correctness of the specification and solves all the interleaving problems that are due to constructivity. The automata-based C code is theoretically the

fastest possible, as only semantically active code is executed in each state. It can also be exponentially larger than the source, due to state-wise code replication.

Several approaches have been proposed to reduce the size of the generated code. The automata compiler of the Polis group [BCG+99] uses binary decision diagrams to identify code that can be shared between states. The HIPPCO automaton optimizer of Castelluccia *et al.* [CDO97] uses code sharing, inlining, and re-ordering of tests to improve not only the code size, but also the execution time (on average and for privileged paths). The Esterel compiler of Bres [Bre02, Est00] does not use the unoptimized automaton representation. It directly generates code where common sub-trees are shared inside a state. While these techniques improve the quality of the generated code, the number of states still explodes for most meaningful examples to the point where the generation of the automaton is intractable.

Surprisingly, benchmarks show that `grc2c`-generated code is often equally fast or faster than the theoretically-optimal automata-based code. The reason is probably twofold. First, the limited size of the processor caches penalizes the usually larger automata-based executables. Second, the GRC code optimizations leave only few redundancies, and the generated code is well-structured. The code generated by `grc2c` can also be exponentially smaller because the translation does not involve state-wise code duplication.

The circuit-based compilers [Est00] start by structurally translating the Esterel program into a sequential digital circuit, at netlist level. The compiler then generates C code that is a *levelized compiled logic simulator*. At each reaction, the code emulates circuit activity by evaluating in an ordered manner all the gates of the circuit. The circuit-based code is compact, quasi-linear in the size of the initial Esterel program. It is also slow because all the gates are evaluated in each clock cycle.

The circuit-based code is always slower than its `grc2c`-generated counterpart, sometimes by factors of 50. The encoding of all program operators using Boolean gates also leads to a larger code size, except for the small examples where aggressive *circuit optimizations* can be applied.

The Saxo compiler of Closse *et al.* [WBC+00, CPP+02] uses a discrete-event interpretation of Esterel to generate a *compiled event-driven simulator*. The compiler flow is similar to that of VeriSUIF [FLLO95], but Esterel's synchronous semantics are used to highly simplify the approach. An *event graph* intermediate representation is used here to split the program into a list of *guarded procedures*. The guards intuitively correspond to events that trigger computation. At each clock cycle, the simulation engine traverses the list once, from the beginning to the end, and executes the procedures with an active guard. The execution of a procedure may modify the guards for the current cycle and for the next cycle.

The resulting code is much faster than the circuit-based one, as the guards prevent the evaluation of semantically-inactive code. At the same time, two reasons make it slower than its `grc2c`-generated counterpart: First, it does not exploit the hierarchy of exclusion relations determined by switching statements like the tests. Second, optimization is less effective because the program hierarchy is lost when the state is (very redundantly) encoded using guards.

The EC compiler of Edwards [Edw02] has been a major source of inspiration in our work. It treats Esterel as having control-flow semantics (in the spirit of [LPM99, Muc97]) in order to take advantage of the initial program hierarchy and produce efficient, well-structured C code. The Esterel program is first translated here into a *concurrent control-flow graph* (CCFG) representing the computation of a reaction. The translation makes the control flow explicit and encodes the state access operations using tests and assignments of integer variables. The back-end accepts only acyclic CCFGs. Its *static scheduling* algorithm takes advantage of the mutual exclusions between parts of the program and generates code that uses *program counter* variables instead of simple Boolean guards. The result is therefore faster than its Saxo-generated counterpart.

EC and `grc2c` share many common aspects. Both use control-flow graphs as intermediate representations and both use static scheduling to generate well-structured code. Moreover, the GRC-level CCFG and the CCFG of the EC compiler are very similar in form because they are obtained through similar Esterel-specific demultiplexing and code duplication. Essential differences remain, however: EC takes a syntax-driven approach in determining the internal concurrency of the Esterel program. Parallel threads are identified on the Esterel source and hard-coded in the CCFG-level state encoding with tests and assignments over integer variables. The approach of `grc2c` is semantics-driven, aimed at achieving better optimization. The GRC code preserves the state structure of the initial Esterel program and uses static analysis techniques to determine redundancies in the activation pattern. Thus, it is able to simplify both the final state representation and the CCFG. For this reason, `grc2c` generates faster code than EC.

3.1 Program correctness issues. Acyclicity

The constructive semantics of Esterel, introduced in section 2, is based on fact propagation schemes traversing at each instant not only the active parts of the program, but also inactive ones. This comes in contradiction with the philosophy of efficient code generation. For this reason, the constructive causality is usually subsumed in this context by a more restrictive requirement, namely *acyclicity*.

Acyclicity cannot be formally defined at the syntactic level of Esterel programs, as it largely depends on connections drawn between signal emissions

and corresponding receptions. It was first defined at the level of circuits, and was simply rephrased on event graphs and CCFGs. Acyclicity is then dependent on both the intermediate representation and the translation scheme. Regardless of the intermediate representation, acyclicity represents the (strict) *restriction of the constructive model* where the evaluation process can be performed in the same order at each clock cycle. Thus, it supports the generation of efficient statically-scheduled C code and represents a cheap, syntactic correctness criterion.

Among the compiling techniques that are able to handle large Esterel programs, the circuit-based approach is the only one currently able to analyze and compile cyclic programs (by transforming cyclic parts into equivalent acyclic ones). However, the procedure is expensive, requiring symbolic state space exploration and circuit resynthesis [SBT96, Tom97, Edw03]. In practice, acyclicity is the most used correctness criterion for Esterel programs, and the only working on large specifications.

Most meaningful programs are acyclic, regardless of the intermediate compiler representation. However, differences subsist, the circuit-based compilers accepting more correct programs due to the finer grain of their intermediate representation. The GRC-based approach supports a refinement technique making the GRC-level acyclicity equivalent to the circuit-level one. Thus, we pave the way towards defining acyclicity as a representation-independent correctness criterion.

4. The GRC intermediate representation

The translation of the Esterel source into GRC (for *GRaph Code*) makes the control flow explicit while preserving an important part of the structural information. The resulting GRC model consists of two objects: a *concurrent control-flow graph* (CCFG) representing the behavior of the Esterel program and a *hierarchical state representation* (HSR), historically called *selection tree*. The CCFG represents in an operational fashion the computation of an instant (the transition function). During each reaction, the dynamic CCFG operates on the static HSR by marking/unmarking component subtrees with “active” tags as they are activated or deactivated by the semantics.

4.1 The Hierarchical State Representation

The hierarchical state representation of a GRC model is an abstraction of the syntax tree of the initial Esterel program. It preserves the modular structure of the program while forgetting about behavioral statements. The HSR of our example is presented in fig. 1.3, along with intuitive tags showing the correspondence between HSR nodes and Esterel statements.

The square-shaped HSR leaves correspond there to simple program statements, like `await`, where control can be suspended between clock cycles. The oval-shaped intermediate nodes correspond to composed statements. Specific HSR tags mark the combination mode in nodes having more than one child. *Exclusive nodes*, tagged with `#`, correspond to sequences and to if-then-else Esterel statements. *Parallel nodes*, tagged with `||`, correspond to parallel Esterel constructs. Simple statements that cannot retain control between clock cycles (*e.g.* `emit`, `exit`) are discarded in the construction of the HSR.

The HSR nodes of indices 0 and 1 do not correspond to program statements. They are automatically generated at program level in order to represent the start state of the program, where no statement is active, but the program is.

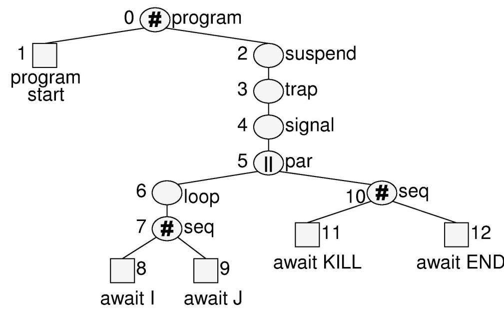

Figure 1.3. The hierarchical state representation (HSR) of our example

Each node of the state representation is endowed with a *selection status* flag (a Boolean data variable) which records at run-time the current activity status of the corresponding subprogram. Selection statuses are checked and modified by *state access operations* performed from the CCFG (and described in the next section). We say that a state representation node is *active* if its selection status is *true*. The definition naturally extends to the associated Esterel statement.

We shall see later that optimization methods will introduce more HSR tags, computed by static analysis of both the HSR and the CCFG. In essence these tags will record redundancy in the selection statuses. For instance it is often the case that a branch of parallel construct is always active, in which case the associated selection status and the CCFG code managing it can safely be removed. Thus, the HSR acts both as a state representation and as a high-level information repository.

4.2 The Concurrent Control-Flow Graph

We pictured in fig. 1.4 the CCFG associated with our small example. Intuitively, control enters the flowgraph at each execution instant through the

topmost node. Then, it follows the edges until complete traversal of the flow-graph.

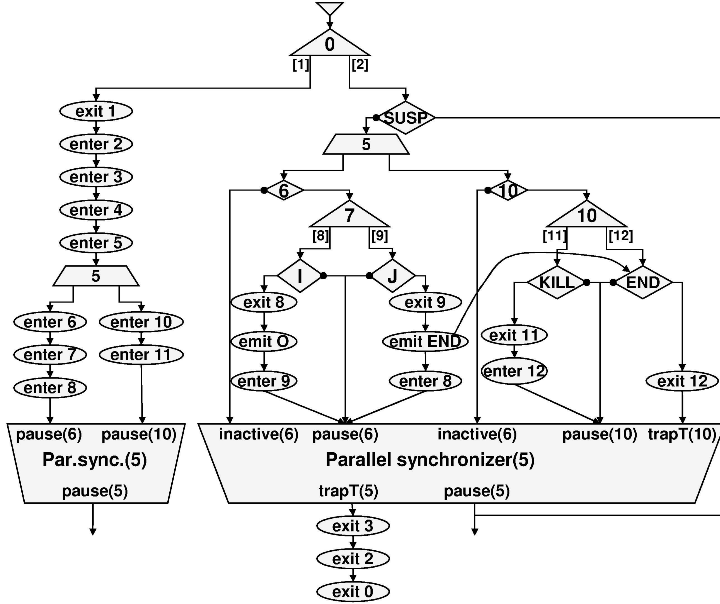

Figure 1.4. The concurrent control-flow graph (CCFG) of our example

All the edges of our example are oriented *control edges* representing control flow and signal `emit/present` dependencies. Control edges propagate the control and the signal statuses between *typed computation nodes*. Esterel programs involving data manipulations may also require the use of *data dependency edges*, representing scheduling constraints not implied by the control ones.

A computation node has a number of *named input and output ports* to which edges will be connected according to the translation rules. Each control edge connects exactly one output port to one input port. Several edges (or none) can be connected to every given port. Most nodes also contain a *data access operation*, allowing them to test/update user-defined variables or selection statuses. During a reaction, a node is executed when it receives enough control and signal information through its input ports. It performs then its computation, which may involve the activation of its data operation, and propagates the results through its output ports. We now describe the nodes and the specific data access operations by exemplifying on fig. 1.4. Note that we dropped (for space reasons) most input and output port names from our graphical representation. Explanations will make up for this inconvenience.

A unique **Tick** node (topmost in our example) serves in every CCFG as unique control entry point for the behavioral reactions. It bears a single output port, called *cont*.

At each clock cycle, the **Tick** node activates the **Switch** node decoding the status of the HSR node of index 0. **Switch** nodes represent the locations where the status of exclusive HSR nodes is decoded in order to resume the appropriate sub-statement. For instance, the **Switch** node decoding the HSR node 0 will decide whether the program is in the start state or in a subsequent execution instant. In the first case, HSR node 1 is active, and the **Switch** gives control to the output port labeled `[1]`. In the second case HSR node 2 is active, and control is given to output port `[2]` to resume the program body. **Switch** nodes are an essential part of our state representation scheme, allowing later an efficient encoding that uses multiway branching instructions.

If the program is in the start state, control is given to a sequence of **Call** nodes performing *state update operations*. The program exits the start state by setting to *false* the selection status of the node 1 (with a call to **exit 1**). Then, it recursively activates the statements all the way from `suspend` to `await I` and `await KILL` by performing **enter** operations on the HSR nodes 2, 3, 4, 5, 6, 7, 8, 10, and 11. The trapezoidal nodes are the **Fork** and the synchronizer **Sync** nodes corresponding to the parallel statement (HSR index 5).

If the program is not in the start state, then we have to perform the suspension test on signal `SUSP`. The associated **Test** node has one input port, named *go*, which represents the test trigger. It has no signal input port because the signal `SUSP`, which is not emitted within the program, is treated like a data variable. The node has two output ports, named *then* and *else* (the latter is identified with a dot). If the test succeeds, the hierarchical state decoding is interrupted and control (given to the *then* port) leaves the flowgraph thereby suspending the execution of the program until the next clock cycle. We also say in this case that the program *pauses*.

If the test fails, the sub-statements of `suspend` are recursively resumed. In practice, this amounts to resuming the branches of the parallel statement *i.e.* checking that they are active (with the **Test** nodes reading the selection statuses of the HSR nodes 6 and 10), decoding their internal status (with the **Switch** nodes on the HSR nodes 7 and 10), and giving control to the appropriate `await` signal tests. Note how the internal signal communication on `END` is encoded using a control edge that connects its emission with its test.

When a parallel branch completes its execution for the current clock cycle, it reports to the parallel synchronizer **Sync** node its completion code (inactive, paused, terminated, or exited through some exception). When all the branches complete the execution of the cycle, the **Sync** node can compute the completion

code of the parallel statement[2] and activate the appropriate output port. In our case, the parallel can pause (suspend until the next clock cycle) or raise the exception `T`. In the first case, the control directly leaves the flowgraph. In the second case the program terminates, as **exit 0** sets the program status to inactive.

CCFG semantics We have presented here the intuitive, control-flow semantics of the GRC flowgraph. Its formal *constructive simulation semantics* [PB02] is based on the constructive circuit semantics of Berry [Ber99]. Under constructive semantics, each control edge of the CCFG has a status of $\perp$, `true`, or `false`. The value $\perp$ means "not yet resolved, not stabilized". In the beginning of each clock cycle, the status of all control edges (with the exception of the inputs) is set to $\perp$. The computation of a reaction consists in incrementally assigning a value of `true` (to represent execution or signal presence) or `false` (to represent inactivity or signal absence) to every control edge. There are technical intricacies such as the fact that a signal becomes *present* (`true` value) as soon as it has been emitted, while it can be turned to *absent* (`false` value) only when all emissions have been provably discarded by control-flow choices (the sequential implementation will avoid this problem by properly scheduling the corresponding assignments so that all emissions precede all signal tests).

The incremental computation of the statuses is performed by the CCFG nodes, which change their output as soon as their inputs change. The computation of a node may have side-effects that model state access and data access operations. Good properties of the computation nodes (monotony) guarantee that once the Boolean status of a control edge is established, it does not change for the current reaction. Thus, the computation converges in finite time. In so-called *constructively causal* programs, signal values are all defined (i.e., not $\perp$) in the end of each reaction from every reachable configuration of the system.

Well-formedness properties Not every GRC specification built with the previously-defined nodes is meaningful. Otherwise said, while the CCFG of a GRC specification is not explicitly structured, it has to satisfy a number of well-formedness properties insuring its correct operation. Here are some of the most important:

- apart from the internal signal and data dependencies, the CCFG must be acyclic

[2] If at least a parallel branch has reported an exception code, then the parallel itself completes with an exception (the highest-level among those generated by branches). If no branch reported an exception code, but at least one paused, then the parallel pauses. Otherwise, the parallel terminates its execution and gives control in sequence.

- the state decoding part of the CCFG has to be hierarchic, following the HSR structure. Moreover, all state update nodes must occur *after* the state decoding nodes.

These properties are insured by the translation scheme defined in the following section and must be preserved by all GRC optimization algorithms.

4.3 Esterel to GRC translation

Most Esterel statements have distinct start and resumption behaviors. For instance, the signal test is performed by "`await S`" only when the statement is resumed. This multiplexing of behaviors produces intuitive, high-level constructs, very useful in the development of large applications. At the same time, it prohibits most control flow optimizations, as the start and resume behaviors are usually subject to different redundancies. The main purpose of the Esterel translation to GRC is to make the control flow explicit and allow its optimization.

In an approach borrowed from Berry's circuit translation [Ber92, Ber99], the translation of Esterel into GRC is based on a structural unfolding using well-defined patterns associated to the primitive statements. The translation involves two phases. The *structural phase* builds the HSR, the graph nodes, and the control dependencies not corresponding to signals. The *link phase* properly connects with control edges the ports corresponding to signal emission and reception, and connects with data dependency edges the data operations that access the same shared variable.

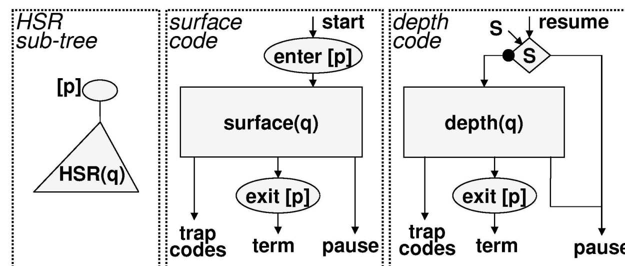

Figure 1.5. Translation of p `= suspend` q `when S`

For each statement three objects are generated: the *HSR sub-tree*, the *surface code*, representing the start behavior, and the *depth code*, representing the resumption behavior of the statement. For instance, the `suspend` statement is translated using the pattern pictured in fig. 1.5. To obtain the GRC representation, the three components are then put together at global level using the pattern of fig. 1.6.

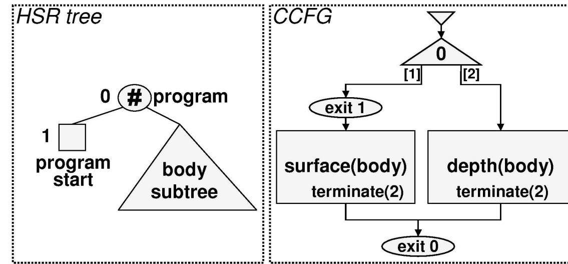

Figure 1.6. The global context

Well-studied semantic reasons [Ber99] often *require* a certain separation between the start and the resume code associated with a statement. Performing it, as we do, in a systematic and thorough way has the disadvantage of increasing the GRC code size. However, the duplication may also reduce the number and scope of signal and data dependencies, thus helping the optimization and software code generation algorithms. More parsimonious code duplication schemes like that of Edwards [Edw02] are supported by GRC, but we did not evaluate them in practice.

4.4 Acyclic GRC specifications

In section 3.1 we introduced the notion of cyclic program, showing that it depends on the chosen intermediate representation and compiling scheme. We also explained why it is important in practice to unify the circuit-level and GRC-level notions of acyclicity. We now derive a partial solution to this problem. More exactly, we use the similarity between the Esterel translations to GRC and digital circuits in order to determine the origin of the differences between the two notions of acyclicity. Based on this analysis, we define a minimal transformation of the GRC code that unifies the two notions, but still allows the generation of efficient C code.

The result is weak in formulation, as it relies specific, unoptimized translation schemes. It is nevertheless important, because it offers a road-map for developing flowgraph- and circuit-based compilers that accept the same classes of programs.

The translation of Esterel into digital circuits is usually performed by encoding the control flow with wires (and gates), while the state is represented using Boolean registers. Without losing generality, we shall use throughout this section the circuit translation of Potop-Butucaru [PB02]. This one, a close variant of the “reference” circuit translation of Berry, is exemplified in fig. 1.7.

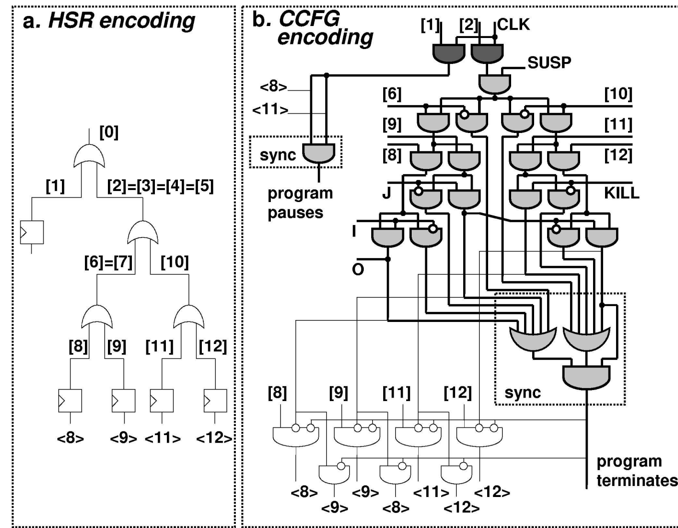

Figure 1.7. The circuit translation of our example. Unfilled gates perform the circuit-level state encoding. To emphasize the similarity with the GRC representation of fig. 1.3 and 1.4, we divided the circuit in two parts. To obtain the global circuit, connect the wires with identical $< num >$ or $[num]$ labels.

The circuit translation is very similar in form to our GRC translation: It is based on patterns associated to the primitive statements of Esterel, and the final circuit is obtained by structurally combining and linking these patterns. In fact, one can consider the GRC code as an intermediate step in the translation of Esterel into digital circuits, a step where the basic Esterel structures (state, tests, synchronizations) have not yet been encoded with logic gates. For instance, most of the gates and wires of fig. 1.7 can be obtained from the corresponding GRC representation (fig. 1.3 and 1.4) through simple expansion operations. We emphasized with darker gates the expansion of the topmost **Switch** node, and with dotted boxes the expansion of the parallel synchronizers.

In both circuits and GRC specifications generated from Esterel programs we can identify a distinct class of components whose role is exclusively to encode the state of the program for the next instant. In GRC, these components are the **enter** and **exit** state update operations. In the circuit of fig. 1.7 they are the unfilled gates and thin wires that drive the registers. These state encoding components never determine cycles, so that we can remove them in order to perform our analysis using simplified versions of the CCFG flowgraph and of the circuit. Fig. 1.8 gives the simplified CCFG and circuit associated with a small example that has the particularity of being GRC-cyclic, but circuit-acyclic.

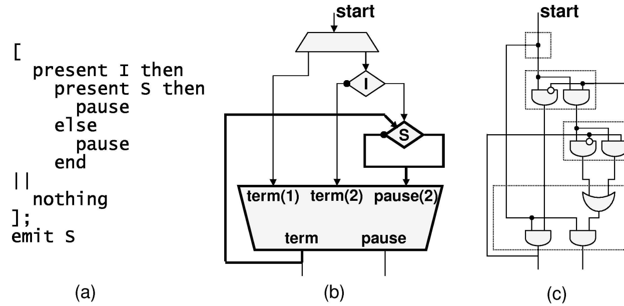

Figure 1.8. Esterel example (a) that is GRC-cyclic (b), but circuit-acyclic (c).

After the simplification of the unneeded state encoding components, a very simple correspondence exists between the GRC nodes and edges and the circuit gates and wires. The simplified GRC code is in fact an abstracted version of the simplified circuit code, which means that a GRC-acyclic Esterel program is always circuit-acyclic.

At this point, we are able to see that expanding into logic gates a node of type **Switch** or **Test** cannot lead to the removal of a cycle, as the resulting logic gates bear the same static dependencies[3] as the initial node. Thus, only the circuit expansion of a parallel synchronizer **Sync** node can remove causal dependencies.

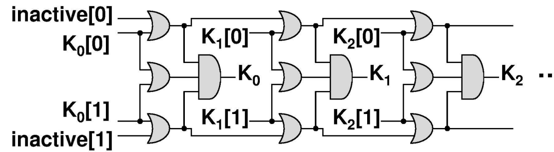

Figure 1.9. The circuit-level parallel synchronizer for two parallel branches (0 and 1). The completion codes are encoded with integers: termination = K_0, pause = K_1, the nth surrounding trap = K_{n+1}. With this encoding, computing the completion code of the parallel statement consists in taking the greatest code produced by a branch.

Meanwhile, the circuit expansion of a synchronizer, given in fig. 1.9, is quite complex and does not offer support for the generation of efficient sequential code. To allow a better encoding, we shall try to keep the gates grouped into

[3]The static dependency relation between ports (input and output, without distinction) of nodes of a CCFG graph, denoted with $\rightarrow$, is the smallest transitive relation such that (1) if i is an input port and o an output port of the same node, then $i \rightarrow o$ and (2) if a control arc connects in the CCFG the output port o with the input port i, then $o \rightarrow i$. The CCFG is cyclic if $p \rightarrow p$ for some port p.

larger blocks. We define here a technique that allows us, in cases where the circuit is acyclic, but the GRC code not, to split the synchronizer into a minimal number of components of a form that allows a simple software encoding.

The observation that allows our minimal refinement concerns the internal structure of the parallel synchronizers: The static dependencies between synchronizer ports that disappear through circuit expansion are $\mathtt{K_i}[\mathtt{br}] \rightarrow \mathtt{K_j}$, with $j < i$. All the other dependencies ($\mathtt{K_i}[\mathtt{br}] \rightarrow \mathtt{K_j}, j \geq i$ and $\mathtt{Inactive}[\mathtt{br}] \rightarrow \mathtt{K_j}$) are preserved. The indices i and j range here over possible completion codes (following the encoding defined in fig. 1.9), while `br` ranges over HSR indices of branches of the corresponding parallel statement.

When a GRC synchronizer is part of a cycle, static dependencies link synchronizer outputs to synchronizer inputs: $\mathtt{K_j} \rightarrow \mathtt{K_i}[\mathtt{br}]$ for some i and j. If the circuit expansion breaks the cycle, then $j < i$. Otherwise, the cycle cannot be broken by circuit expansion. We are then looking for the minimal expansion of a synchronizer, under given constraints of the form $\mathtt{K_j} \rightarrow \mathtt{K_i}[\mathtt{br}]$, $j < i$.

We shall be splitting the synchronizers into parts that correspond to sets of consecutive completion codes. A parallel synchronizer `SYNC` handling completion codes from 0 to n can be split into $\mathtt{SYNC}[0, \mathrm{k_1}]$,...,$\mathtt{SYNC}[\mathrm{k_r}, \mathrm{n}]$ with $0 < k_1 < \ldots < k_r \leq n$. Here, $\mathtt{SYNC}[\mathrm{k_i}, \mathrm{k_{i+1}}]$ receives the completion code wires $\mathtt{K_i}[\mathtt{br}], k_i \leq i < k_{i+1}$, produces the outputs $\mathtt{K_i}, k_i \leq i < k_{i+1}$, and:

- if $k_i = 0$, then $\mathtt{SYNC}[\mathrm{k_i}, \mathrm{k_{i+1}}]$ receives the inputs $\mathtt{Inactive}[\mathtt{br}]$.
- if $k_i \neq 0$, then $\mathtt{SYNC}[\mathrm{k_i}, \mathrm{k_{i+1}}]$ receives from $\mathtt{SYNC}[\mathrm{k_{i-1}}, \mathrm{k_i}]$ one carry wire per parallel branch.

Thus, our problem is to determine a sequence $0 < k_1 < \ldots < k_r \leq n$ of minimal length r such that for each static dependency $\mathtt{K_j} \rightarrow \mathtt{K_i}[\mathtt{br}]$ ($j < i$) there exists $1 \leq t \leq r$ such that $i < k_t < j$ (so that the cycle determined by $\mathtt{K_j} \rightarrow \mathtt{K_i}[\mathtt{br}]$ is broken through refinement). The algorithm that determines such a sequence is given next. It takes as parameter the set `DEPEND` of pairs (j, i) such that a static dependency $\mathtt{K_j} \rightarrow \mathtt{K_i}[\mathtt{br}]$ ($j < i$) exists in the GRC code, and returns the list `LIST` of splitting points (numeric completion codes):

```
LIST:=NewVoidList ;
for code:=0 to n do
  if (i,code) in DEPEND then
    Append(LIST,code);
    RemoveDependenciesBrokenByNewSplitPoint(DEPEND,code)
  end
end
```

We do not formally prove here the minimality of the resulting split. Intuitively, it is determined by the fact that split points are only appended to the list when

this is required. For instance, if only one dependency exists (like in fig. 1.8), the synchronizer shall be split in two parts.

5. Analysis and optimizations

The translation of Esterel into both circuits and flowgraph-based formats like GRC usually results in code that is far from optimal. There are good reasons for this: Translations have to be structurally defined, so that the code resulting from a subprogram includes elements allowing it to be handled in *any* surrounding reactive context. For instance, provision is made in the translated code to let *any* subcomponent be started, resumed, or preempted. In addition, the translated form maintains the activity information everywhere, consuming memory elements for the state representation. Very often, large parts of the program do not need such a heavy complete encapsulation (because they are always active, to mention a simple case), and the description can be drastically simplified.

For FSMs and digital circuits well-studied optimization techniques exist, based on mathematical properties of the underlying formal models: bisimulation minimization in the first case, combinational and sequential logic optimization in the second. These techniques come together with very efficient algorithms and software tools. However, they are mostly of a global nature, so that they are generally not able to handle industrial-size specifications. At GRC level one can hope to define optimizations based on static analysis techniques, less complete but more efficient in complexity of analysis (low-exponent polynomial). One main prospect in the introduction of the GRC format was indeed to be able to describe and formally justify such optimizations on a well-defined representation model. Some work has already been done in this direction in the existing flowgraph-based compilers. Still, results lack generality and formal support, due essentially to limitations of the intermediate compiler representations.

The optimization of a GRC specification is performed in two phases. First, potential redundancies between state encoding elements are detected through static analysis of the GRC code, and represented with tags on the HSR. The tags are then used in the second phase, where actual optimizations (by removal of edges and nodes) are performed on the CCFG. Actually, the tags are also used to improve software encoding, but this aspect shall be presented in section 6.

The CCFG simplification phase may unveil more redundancies between state elements, so that the optimization can be iterated to further savings. Current practice shows that three optimization cycles (tag computation followed by flowgraph simplification) are enough on our largest examples to reach full optimization.

5.1 HSR tag computation

The HSR serves two purposes in a GRC representation. It completes the semantics of the flowgraph by keeping track of the state hierarchy and it also is a repository for tags representing potential redundancies in the state representation. The choice of tags is given by their utility (experimentally determined) in the optimization and encoding process. In addition to `#` and `||`, that were already defined, we currently use the tags: `nt:` and `void:`. Here is their meaning:

- `void:` indicates that the tagged HSR node never remains active at the end of a reaction. The associated statement is either never started, or instantly preempted upon start.

- `nt:` (for *non-terminating*) indicates that the selection status of the tagged HSR node is always equal to that of its parent. The tag is of specific interest on direct children of parallel nodes. In general, a parallel branch can terminate while other branches (and thus the parallel itself) remain active. This, however, is not the case when the branch never terminates, or when it only does so by raising an exception, thereby aborting the whole parallel construct.

Note that other tags can be easily defined for use with different analysis and optimization techniques. Promising analysis algorithms need, for instance, information about loop scopes, or the refinement of the exclusion relation into sequence and conditional exclusion. Our set of tags is only a first version, with already good practical results.

The *computation of tags* currently relies on algorithms of linear complexity in the size of the GRC specification. We set the `void:` tag on the HSR node of index n if it falls in one of the following cases: (1) The CCFG contains no **enter**(n) operation. (2) All of n's children are tagged with `void:`. (3) The parallel node n has at least one child tagged with both `void:` and `nt:`. (4) The node n is child of a parallel node itself tagged with `void:`.

The `nt:` flag is set on all the children of unary HSR nodes. In addition, it is set onto direct children of parallel nodes based on the following analysis of the CCFG parallel synchronizers: *Let* p *be a parallel HSR node, let* c *be one of its children, and let* $s_1, \ldots, s_n$ *be the parallel synchronizers associated with* p*. Then, we set the* `nt:` *tag on* c *if none of* $s_1, \ldots, s_n$ *receives a control edge representing the normal termination of the parallel branch* c *(*i.e. *if the translation did not generate such an edge or if it has been optimized out).* For our example, the result of the tagging phase is given in fig. 1.10(a).

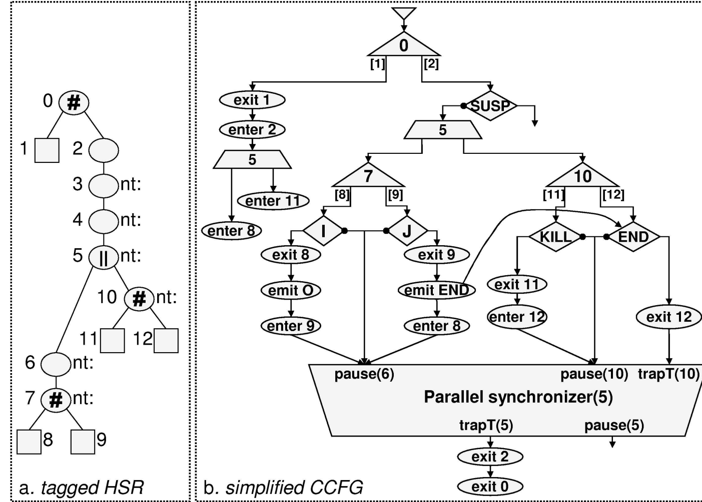

Figure 1.10. Tagged HSR and optimized CCFG for our example

5.2 CCFG optimizations

We apply on GRC flowgraphs two types of simplifications: (1) state access/update protocol simplifications that take into account the new HSR tags and (2) "internal" graph simplifications that do not use state representation information.

Simplifications based on HSR tags. These simplifications take into account the state representation redundancies that are represented with tags in order to simplify the state access/update protocol by removing unneeded operations. In many cases, maintaining the coherency of the flowgraph also requires the removal of the driver flowgraph nodes. The following simplifications are currently performed:

- the **Switch** nodes are simplified by erasing the outputs corresponding to `void:` branches. If only one output remains, the node is deleted, and the output is directly connected to the *go* input.
- **Test** nodes whose target is tagged with `void:` or `nt:` are deleted. If only one of the tags is set on the target, then the *go* input is directly connected to either *then*, if the tag is `nt:`, or *else*, for a `void:` tag.
- state update operations whose target is tagged with `void:` or `nt:` are deleted. The associated **Call** node is also deleted, and its output is directly connected to its input.

Internal flowgraph simplifications. Derived for the most part from classical control-flow optimization techniques, the "internal" graph simplifications delete dead code, reduce the number of dependencies, and simplify the nodes. We mention three techniques that proved very effective in practice:

- the *constructive sweep* intuitively corresponds to constant propagation and to dead code removal. Dead code is identified by performing a symbolic execution of the GRC code where all accessible control paths are explored in parallel (linear complexity). Then, we can delete control edges that have not been traversed. We also delete some of the adjacent nodes:
  - **Call**, **Test**, and **Switch** nodes whose *go* input port receives no more edges, as well as **Sync** nodes whose input edges have all been deleted.
  - **Test** nodes whose expression has been uniformly set to *true* or *false* due to the removal of signal edges.
- the *dependency simplification* consists of removing certain signal and data dependencies between exclusive control branches (*e.g.* branches of a test). We currently perform the simplification on tests (**Test** or **Switch** nodes) whose expression only involves input signals or state tests.
- the *useless code simplification* consists of deleting code that does not drive actions on data (state changes, output signal emissions, or user data update).

Figure 1.10(b) shows the result of the simplifications on the control-flow graph of our example.

6. Code generation

Once tagged and optimized, a GRC model is translated into sequential C code. There are two important steps in this translation, to further improve efficiency. First, a state encoding is choosed so that various **enter** and **exit** operations of the CCFG can be collapsed into fewer C-level data operations. Second, a proper static scheduling of the parallel threads in the CCFG is synthesized. We recall that our approach currently handles only GRC-acyclic and circuit-acyclic specifications, due essentially to the static scheduling algorithm which allows an efficient, low-overhead encoding of the context switches.

6.1 State encoding

The state encoding must allow the hierarchic (top-down) identification of the active program statements, through successive tests corresponding to the

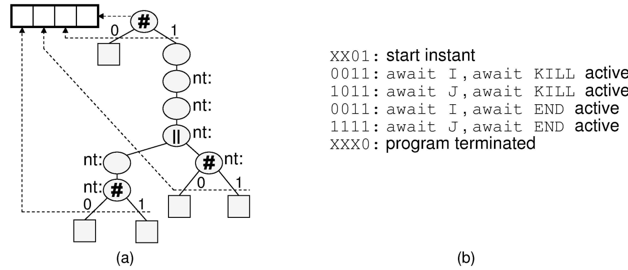

Figure 1.11. State encoding for our example: (a) the bit allocation, optimized according to the HSR tags and (b) the encoding of the reachable states of the program.

state decoding nodes of the CCFG. Our encoding technique basically tries to minimize the number and complexity of the state access operations performed in a clock cycle, but without performing code duplication (like the automaton expansion does).

The encoding algorithm starts by associating an integer test variable to HSR nodes where a state decoding decision may be necessary:

- exclusive selection nodes having more than one child not tagged with `void:`
- children of a parallel node that are not tagged with `nt:`

In the first case, the n children that are not tagged with `void:` are assigned indices from 0 to $n-1$, and the variable is set to i to mark the activation of the branch of index i (and the de-activation of all the other). In the second case, the variable is set to 1 when the parallel branch is active. Otherwise, it is 0. The variable assignment can in fact be further optimized, as the activation bits are in many cases redundant.

The second encoding phase uses exclusiveness properties to minimize the *size* of the representation. The basic remark is here that variables associated to nodes on different sub-trees of an exclusive selection node will never be decoded in the same reaction. Then, they can share the same implementation variable to minimize the executable size and the cache-miss probability. The multiplexing algorithm is actually performed at bit level, in a bottom up fashion. This phase defines the final mapping of the state operations onto the state variables of the implementation.

Figure 1.11 pictures the bit allocation pattern of our example and gives the encoding of the reachable states. For instance, 0011 represents an active program (bit0=1) that is not in the start reaction (bit1=1) and that awaits the signals `I` and `KILL`.

In order to match the HSR encoding, the state access primitives are transformed into assignments and tests of the integer state variable. This step is essential in the generation of efficient code, as each assignment operation usually encodes multiple state update primitives.

6.2 Scheduling and C code generation

To be executed on a sequential machine, the operations of the concurrent CCFG must be *scheduled*, *i.e.* totally ordered in a way that complies with the causality relation. A scheduling algorithm works by *interleaving* the concurrent branches of the CCFG using *context switches* where the execution of a branch is suspended and another branch is resumed. Context switches take time, and efficient schedules should have as few of them as possible. Determining a schedule with a minimal number of context switches is NP-complete, but our approach produces acceptable code using a simple depth-first scheduling algorithm.

The CCFG represents with edges all control and data causality, so that any topological order is a valid schedule. To avoid complex, data-dependent scheduling issues, only acyclic GRC specifications are accepted for code generation. In these cases, a topological order always exists and any topological order gives a *static schedule*, *i.e.* one that works at each clock cycle.

```
if(state&binary(0001)){   /*program is active*/
  if(state&binary(0010)){ /*not boot instant*/
    if(!SUSP) {
      int par = 1 ;       /*parallel completion code, init. with the minimal code*/
      bool cs = false ;   /*context switch variable*/
      bool END = 0 ;      /*the internal signal, initially absent*/
      /*parallel branch 2 starts the execution*/
      if(state&binary(0100)) { cs=true ; /*suspend execution on branch 2*/}
      else if(KILL) { state&=binary(1011) ; }
      /*parallel branch 1 starts the execution*/
      if(state&binary(1000)){ if(J){ END=1 ; state&=binary(0111) ; } }
      else{ if(I){ O=1 ; state|=binary(1000) ; } }
      /*parallel branch 2 is resumed*/
      if(cs){ if(END){ par=2 ; } }
      /*parallel synchronization code*/
      if(par==2){ emit_O() ; state=binary(0000) ; }
    }
  }else{ /*boot instant*/ state=binary(0010) ; }
}else{ error(``program already terminated'') ; }
```

Figure 1.12. Resulting C code. Note the Boolean *context switch variable* `cs` that is used to resume the second parallel branch after the first one is executed. The global variable `state` is initialized with `0001`.

To determine an acceptable static schedule, the code generator uses a simple "greedy" approach. Throughout the scheduling process, the list of concurrent branches is cyclically traversed from the beginning to the end. The code generator then uses a depth-first traversal technique to schedule as much as possible of the current concurrent branch before, if necessary, suspending it

and going to the next after performing a context switch. Fig. 1.12 shows the reaction function generated for our example. The variable `state` holds the state representation. The context switches are encoded using Boolean guards. The example needs only one (cs) in order to resume, if necessary, control on the second parallel branch. In general, the resulting code is well-structured, as guards corresponding to lower-level concurrent threads are subsumed to higher-level ones.

7. Results

To measure the efficiency of our approach we developed a prototype GRC-based optimizing compiler, called `grc2c`, that plugs into the Esterel compiler system [Est00] developed at INRIA/CMA. The prototype compiles all GRC-acyclic Esterel programs and implements the optimizations described in this paper. In-depth technical information concerning the implementation is provided in [PB02].

example	description	Esterel(LC)	CCFG nodes		
		statements	initial	optimized	
1-tcint	turbochannel bus	516	798	375	46.9%
2-wristwatch	berry's example[Est00]	533	834	366	43.8%
3-atds100	video generator	1122	2119	1059	49.9%
4-mca200	shock absorber[CEG+96]	3769	4562	3596	78.8%
5-Chorus	OS model[WBC+00]	4751	6299	3539	56.1%
6-Fuel	avionics fuel controller	4986	8544	4516	52.8%
7-cabine	avionics display[BBF+00]	10991	18872	8037	42.5%
8-global	avionics controller	15831	18852	13253	70.3%

Table 1.1. Testbench examples: description and GRC optimization results

We compared the output of `grc2c` with the output of four other compilers: The automata compiler of Bres[Bre02], the circuit-based compiler of Berry [Est00], the EC compiler of Edwards [Edw02], and the Saxo compiler of Closse *et al.* [WBC+00]. A state-of-the-art sequential circuit optimizer (SIS/blifopt [SSL+92]) has been used whenever possible on the intermediate netlist representation of the circuit-based compiler.

The examples of the testbench are, `wristwatch` excepted, industrial examples of controllers, either directly programmed in Esterel or obtained through automatic translation from SyncCharts [And96] specifications. Table 1.1 gives the initial and optimized specification sizes, and a short description for each of our 8 examples. The LC statement count is good estimation of the Esterel specification size. The unoptimized number of nodes in the GRC flowgraph is always larger than the LC count due to the demultiplexing and to the structural

code duplication. However, despite variations in the optimization ratios, the optimized figures always fall under the LC count, meaning that our systematic code duplication pays off. Recall that the optimization algorithms are all linear,

example	automata	circuit	EC	Saxo	grc2c
1-tcint	0.30	0.66[5]	**0.25**	0.36	**0.25**
2-wristwatch	**0.95**	1.40[5]	1.26	1.29	**0.95**
3-atds100	**0.05**	3.69[5]	0.14	0.12	0.08
4-mca200	–[4]	48.19[5]	2.61	3.68	**1.88**
5-chorus	–[4]	54.29	2.76	5.30	**1.10**
6-fuel	–[4]	37.66	–[7]	16.81	**15.65**
7-cabine	–[4]	–[6]	–[7]	31.97	**29.26**
8-global	–[4]	–[6]	–[7]	57.45	**43.27**

Table 1.2. Generated code speed (sec/1Mcycle)

so that the optimization phase does not represent a bottleneck in the compilation process. On the test machine (a Linux/PIII/1GHz/128M) it takes less than 1 minute for the largest example, depending quasi-linearly on the specification size.

example	automata	circuit	EC	Saxo	grc2c
1-tcint	72.8	**11.4**[5]	11.5	15.3	17.5
2-wristwatch	14.8	**11.2**[5]	13.3	16.1	14.7
3-atds100	28.8	31.3[5]	**21.7**	33.9	33.7
4-mca200	–[4]	171.8[5]	70.2	78.9	**67.7**
5-chorus	–[4]	230.3	99.5	104.1	**98.6**
6-fuel	–[4]	201.4	–[7]	**147.5**	168.7
7-cabine	–[4]	–[6]	–[7]	256.1	**196.8**
8-global	–[4]	–[6]	–[7]	309.1	**273.9**

Table 1.3. Object code size (Kbytes)

For each example, the C routines generated by the 5 compilers have been compiled using "gcc2.96 -O" and then run for 1 million cycles to determine the average reaction time. The input sequences were either provided by the programmer of the application or (when none provided) generated pseudorandom

[4]Out of memory or timeout (1 hour) during automaton expansion
[5]The intermediate circuit representation has been aggressively optimized using SIS/blifopt prior to C code generation (only possible on small- to medium-sized specifications).
[6]gcc -O exhausted system memory during compilation.
[7]The code was not available for tests.

inputs satisfying the given environmental constraints. The results are presented in the tables 1.2 and 1.3.

The three flowgraph-based compilers (EC, Saxo, and `grc2c`) produce similar figures in both speed and size. Among them, `grc2c` generates faster code, while the differences in code size are not relevant. This seems to indicate that (1) removing redundant state tests (like we do) greatly improves the code speed and (2) the C language encoding of `grc2c` could be improved to match on small examples the EC-generated code size.

As expected, the code generated by the circuit-based compiler is slow, even when the intermediate circuit representation has been aggressively optimized. The circuit-based code is also very large and difficult to compile.

Surprisingly, only one example results into `grc2c`-generated code that is slower than its automata-based counterpart. This is probably due to the fact that the (small) processor caches penalize larger programs, and to the fact that GRC-level optimizations leave only few redundancies in the state representation. The automata-based compilation is restricted to small programs, even more than circuit optimization. For all but 3 examples in our testbench, we interrupted the automaton expansion after 1 hour.

The global `grc2c` compilation time, including GRC code generation, optimization, and C encoding, is negligible. It increases from several seconds to approximately 2 minutes, following the size of the GRC specification. The `grc2c`-generated C code is easily compiled by `gcc`.

8. Conclusion and Future work

We introduced in this paper a new intermediate model, called GRC, for the representation of Esterel programs. The GRC representation makes the control flow of the Esterel program explicit, but also preserves information about its initial structure. Thus, its supports a variety of efficient analysis, optimization, and sequential code generation algorithms based on low-cost static analysis. Benchmarks show that the GRC-based `grc2c` compiler generates faster code than the other high-capacity compilers, for a similar code size.

The GRC representation has well-defined formal semantics, allowing us to (1) prove the correctness of the software code with respect to the constructive semantics of Esterel and (2) maintain a close relation between GRC and circuit translations of an Esterel program. We were therefore able to define a notion of acyclic GRC description and match it with the corresponding one on circuits. The result is of special practical interest, since acyclicity is the main correctness criterion used for Esterel programs.

The `grc2c` compiler has been implemented as a module of the INRIA/CMA Esterel version 5 compiler system. At the same time, some of the optimizations defined in this paper have been incorporated into Esterel Studio, an industrial

Esterel-based environment for the design and verification of complex systems-on-a-chip (SoC).

Other applications are possible, in addition to generating software code for Esterel. The GRC-based approach could be easily adapted to compile other synchronous languages and formalisms with fine-grained parallelism, such as ECL [LS99] or the SyncCharts [And96]. However, the most obvious application of Esterel is the generation of optimized digital circuit code for Esterel: The analysis and optimizations presented in this paper are not software-specific and some experiments [PB01] suggest that applying them to Esterel-generated circuits would result in important simplifications. Such Esterel-specific simplifications should be performed *before* the main circuit optimization phase in order to improve its output and (usually long) execution time. However, applying the simplifications on flat netlists would be inefficient. Instead, the GRC code should be used as a *target-independent representation* for (Esterel-specific) optimizations based on high-level structural information. The optimized GRC code should then be translated into a netlist and passed to a state-of-the-art circuit optimizer. Such a GRC-based optimized circuit generator is under construction at Columbia University under the supervision of Stephen Edwards.

Many directions remain to be explored. First, it is essential to pursue the development of new GRC-level analysis and optimization techniques. The domain seems promising due to the large quantity of Esterel structural information not yet exploited. An interesting problem is here to determine the amount of code duplication that results in the best size/speed ratio, and to reduce the unneeded duplication by sharing identical branches.

A second direction that we would like to investigate is the generation of efficient simulation code for *correct cyclic* GRC specifications, thereby raising the applicability to the same class of programs as the automata- and circuit-based compilers. Once cycles have been precisely located and proved correct, an idea would be to run them locally under the 3-valued constructive semantics of circuits. Some advances in this direction can be found in Potop-Butucaru [PB02]. Other schemes would require substituting the cyclic GRC subgraph by another, equivalent acyclic one. Of particular interest is here the resynthesis technique of Edwards [Edw03], which performs a controlled code duplication that preserves many of the structures of the initial circuit. The technique could be extended to perform a similar GRC-level resynthesis while still using the current circuit-level analysis. Another question, related but nevertheless distinct, would be to find GRC-level static analysis approximation methods that give sufficient conditions for correctness in cyclic cases, and then try to derive a corresponding, more dynamic scheduling scheme which could cope with different ordering according to different state configurations.

Finally, we are investigating the possibility of a union, at intermediate representation level, between Esterel and the synchronous data-flow language Signal,

for analysis and efficient simulation purposes. The main goal would be here to use Esterel for specifying behaviors and Signal for specifying architectural aspects. The high-level intermediate structures of the Signal compiler – the *clock tree* and the *conditional dependency graph* – are respectively similar in function and form to the HSR and the CCFG of the GRC code. The first represents the hierarchy of the activation conditions, while the second represents the actual control and data flow. However, important differences exits: The HSR of a GRC specification represents the internal structure of the Esterel program, closely followed by the state encoding/decoding part of the CCFG. In the Signal compiler, the activation conditions of the clock tree are synthesized from (clock) constraints on the input and output signals. Thus, the union would require the ability to assign/derive complex interface constraints to/from an Esterel specification.

References

[And96] Charles André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proceedings CESA'96*, Lille, France, July 1996. Also available as I3S technical report RR 95-52.

[BB91] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[BBF+00] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. Esterel: A formal method applied to avionic software development. *Science of Computer Programming*, 36:5–25, 2000.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul LeGuernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCG+99] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design*, 18(6):834–849, 1999.

[BdS91] Frédéric Boussinot and Robert de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[Ber92] Gérard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London, Series A*, 339:87–104, 1992.

[Ber99] Gérard Berry. The constructive semantics of Pure Esterel. Draft book available at http://www.esterel-technologies.com/, 1999.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Bre02] Yannis Bres. *Exploration implicite et explicite de l'espace d'états atteignables de circuits logiques Esterel (Implicit and explicit exploration of the reachable state space of Esterel logic circuits)*. PhD thesis, Ecole des Mines de Paris, december 2002.

[CDO97] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[CEG+96] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided co-design of embedded systems. *Design Automation for Embedded Systems*, 1:51–67, 1996.

[CGP99] Paul Caspi, Alain Girault, and Daniel Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999.

[CPP+02] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Vernier, and Daniel Weil. Saxo-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Eric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[Edw02] Stephen Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), February 2002.

[Edw03] Stephen Edwards. Making cyclic circuits acyclic. In *Proceedings of the 40th conference on Design automation*, Anaheim, CA, USA, 2003.

[Est00] The Esterel manual, part of the Esterel version 5.92 distribution. Available for download at www.esterel-technologies.com, 2000.

[FLLO95] Robert French, Monica Lam, Jeremy Levitt, and Kunle Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32nd Design Automation Conference*, San Francisco, CA, USA, 1995.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[LGLL91] Paul LeGuernic, Thierry Gauthier, Michel LeBorgne, and Claude LeMaire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[LPM99] Jaejin Lee, David Padua, and Samuel Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, USA, 1999.

[LS99] Luciano Lavagno and Ellen Sentovich. ECL: A specification environment for system-level design. In *Proceedings of the 36th Design Automation Conference*, New Orleans, LA, 1999.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[PB01] Dumitru Potop-Butucaru. Fast redundancy elimination using high-level structural information from Esterel. RR 4330, INRIA, Sophia Antipolis, France, 2001.

[PB02] Dumitru Potop-Butucaru. *Optimizations for faster simulation of Esterel programs*. PhD thesis, Ecole des Mines de Paris, November 2002.

[SBT96] Tom Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the International Design and Testing Conference*, Paris, France, 1996.

[SSL+92] Ellen Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul Stephan, Robert Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Memorandum M92/41, UCB, ERL, 1992.

[Tom97] Horia Toma. *Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif Esterel (Constructive analysis and sequential optimizations of circuits generated from the synchronous reactive language Esterel)*. PhD thesis, Ecole des Mines de Paris, September 1997.

[WBC+00] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Vernier, and Jacques Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proceedings CASES2000*, San Jose, CA, USA, 2000.