Separate Compilation and Execution of Imperative Synchronous Modules

Eric Vecchié
INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
Eric.Vecchie@irisa.fr

Jean-Pierre Talpin
INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu
35042 Rennes Cedex, France
Jean-Pierre.Talpin@irisa.fr

Klaus Schneider
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern, Germany
Klaus.Schneider@informatik.uni-kl.de

***Abstract*—The compilation of imperative synchronous languages like Esterel has been widely studied, the separate compilation of synchronous modules has not, and remains a challenge. We propose a new compilation method inspired by traditional sequential code generation techniques to produce coroutines whose hierarchical structure reflects the control flow of the original source code. A minimalistic runtime system executes separately compiled modules.**

I. Introduction

Synchronous programming languages are domain-specific languages dedicated to the design of real-time embedded systems. The imperative style of Esterel [2] has been specifically designed to ease programming control-dominated applications. Its efficient compilation has been a major issue for more than a decade [3], [1], [8]. More recently, the increasing complexity of systems and the need for integrating IPs (Intellectual Property) have motivated a renewed interest for the modular compilation of synchronous languages [10], [9], [7]. True modular compilation suits application area focusing on rapid prototyping, simulation and test or flexibility demands posed by the need for dynamic reconfigurability (e.g. for space applications). Unlike related approaches, we start from the very same notion of modular compilation as in general-purpose languages like C: the translation of an Esterel module into binary code with a header file describing its interface. The instantiation of a module is simply realized by passing its parameters and executing its code through simple calling conventions.

Early Work: Because it was thought to make causality analysis more difficult, modular compilation were not considered in early implementations of Esterel. Instead, early compilers generate code from a data-structure representing a flattened and inlined expansion of the source program. For instance, [3] translates a program into an extended finite state machine whose transitions are decorated with code fragments. The obvious disadvantage of this technique is the potential state-space explosion of the automaton. Its main advantage is the very short execution time of the generated code. Compilation techniques achieving a polynomial space complexity were first obtained by using systems of equations to symbolically represent the automaton of a program [1]. This approach was successfully used for hardware synthesis and it is still at the core of commercial tools, although the generated software is usually slower. Another approach is based on the translation of programs into concurrent control flow graphs [8] whose size grows linearly on the given program. At each instant, the control flow graph is traversed until active nodes are found to trigger the execution of the corresponding sub-tree.

Related Work: Some works [9], [10] address different notions of "modular compilation". In [10], the generated code is able to partially deal with undefined inputs by using a three-valued logic. The generated programs try to compute as many outputs as they can while ignoring inputs that are still unknown. In [9], the Esterel dialect Quartz is translated into a "job language". This translation can be conceptually regarded as a graph of simultaneously active jobs where each job contains an atomic task. In these approaches, the increased effort involved for the compilation of the modules separately does not spare some "tuning" of the generated code at instantiation time, depending on the calling context. Unlike our approach, this implies that the code of the instantiated module must be duplicated. Other works [7] use multi-function interfaces to generate modular code for synchronous block diagrams. Each function in the interface is responsible for the evaluation of some outputs through the evaluation of the relevant part of the module. The trade-off between modularity (decreasing with the number of functions in the interface) and reusability (increasing with the number of functions) is also addressed. In our approach modularity is maximized since the compilation of each module generates a single function. Reusability is nevertheless ensured by a flexible model of execution which allows causality issues to be resolved dynamically at run-time.

Our Approach: Our method tries to mimic the well-known code generation techniques described in classical compilation books. Expressions are flattened and translated into assembly code sequences. Conditional statements are replaced by conditional gotos and function calls use stack frames for passing the parameters and saving the register context of the caller. Meanwhile, this compilation scheme has to capture the particular model of computation of the synchronous languages.

Model of Computation: Naive semantics of Esterel goes as follows: programs behaviors are discretely divided into instants. Control threads are executed until reaching a `pause` statement, which cuts behaviors into atomic instants. We call "reaction" the full behavior performed during a given instant. In a reaction cycle, input signals are sampled, and internal computation takes place until output signals are emitted in answer, and the program state is progressed. Instants are based on a common logical clock, which paces all parallel threads. This (the fact that all components proceed with the same atomic logical instants) is why we call the model "synchronous". Of course in a reaction various parallel threads do not run independently, as they may synchronize and affect one another causally. When control reaches a statement involving the value of a signal, it may have to postpone execution until a consistent definitive value is obtained for the signal inside the current reaction (either because it is assigned somewhere in parallel, or because other threads of execution progressed to a point where provably all potential emissions were discarded). The topic of constructive causality is a large body of Esterel semantic theory, but we shall not address it here; instead we shall assume that there is no cyclic dependency between signals.

978-3-9810801-5-5/DATE09 © 2009 EDAA

Model of Execution: Our execution scheme is rather inspired by the reactive kernel of Junior [6]. This tool is a relative of the Esterel language used to write reactive applications through a Java API. Unlike Esterel, the reaction to the absence of a signal is delayed to the next instant. In Junior, each reaction step executes and reduces a tree representing the instantaneous state of the reactive program and reflecting its original structure. The major complexity of our approach lies in the conciliation of this execution scheme with the traditional compilation techniques. In this purpose, we use a mechanism similar to co-routines [5] so that the control yields at some points of the generated program. These co-routines are hierarchically nested to reflect the control structure of the source program.

The paper is organized as follows: the section II describes the considered language. We present the translation of the language into "high-level" assembly code in section III and our solution for dealing with signals communications in section IV. We finally conclude this article by some future works.

II. The Synchronous Language

As source language, we use a simple but sufficient Esterel-like language, containing the essential constructions of imperative synchronous languages. It contains the following statements:

- `pause` (division between instants)
- `emit` V / V `=` E (pure/valued signal emission)
- `if` E `then` P_1 `else` P_2 (conditional)
- `do` P `while` E (iteration)
- { P_1`;` $\dots P_n$ } (sequential block/scope)
- P_1 `||` P_2 (synchronous parallel composition)
- [`weak`] `abort` P `when` E ([weak] abortion)
- `suspend` P `when` E (suspension)
- *name*`(`V_1`,` $\dots V_n$`)` (module instantiation)
- `module` *name*`(`V_1`,` $\dots V_n$`)`$\{P\}$ (module declaration)

Variables are boolean, integer or pure signals. Expressions can be any boolean (`present`, `and`, `or`, `not`), arithmetic (`+`, `-`, ...) or comparison expression (`==`, `!=`, `>`, ...) based on signals and constants. As we stick to the semantics of Esterel, further documentation can be found in [2].

III. Our Compiler

The synchronous modules are translated into sequential code intended to be executed in a Runtime System. The task of this system is rather simple: it maintains a collection of *cooperative* threads to be executed during the current logical instant (reaction) and a second one to be executed at the next logical instant. The "*current*" list of threads can be dynamically enlarged. This is typically the case when a thread reaches a parallel statement. Thus, the execution of a reaction step means the execution of all the threads of the *current* list until it is made completely empty. The same way, the execution of a synchronous program means the execution of reaction steps until the "*next*" list of threads is empty. The second task of the Runtime System is also to deal with the signal emissions and resets but we shall come back to this later.

A. Compilation of Sequential Statements

The compilation of classical sequential statements follows a classical compilation scheme using conditional gotos for *if-then-else* and *do-while* statements. We shall simply consider these statements as part of our "high-level assembler". In the same way, local variables are referred through their names rather than FP[i] where FP is the Frame Pointer register and i would be the relative address of the concerned variable in the current frame. New stack frames are allocated when scopes are entered. In our implementation, a particular care has been taken to reallocate unused frames through a simple garbage collection mechanism.

B. Parallel Statements

In cooperative multithreading, each thread is responsible for relinquishing control. This is ensured by the assembly instruction "`stop`". Starting a new thread is done with the instruction "`start` pc`,` fp" where pc is the address of the first instruction of the started thread and fp is its frame pointer. The translation of the parallel statement P_1 `||` P_2 is then the following:

```
    sync = 2
    start FORK_lbl, FP
    P1
    goto SYNC_lbl
FORK_lbl:
    P2
SYNC_lbl:
    sync = sync-1
    if sync > 0 then stop endif
```

Here "`sync`" is a local variable used to synchronize P_1 and P_2 so that the first thread reaching `SYNC_lbl` is stopped and the second one goes on. This translation can be easily adapted for n-ary parallel statements.

C. Hierarchical Coroutines

The execution of a synchronous program is a succession of atomic reactions during which threads run in parallel until reaching a `pause` statement. In the beginning of each reaction step, the threads are resumed at the very locations where the program eventually paused at the end of the previous reaction. However, **not all** threads are resumed: because of `abort` and `suspend` statements, the resumption of threads should actually be performed hierarchically, according to the structure of the source program and the dynamically fulfilled conditions.

1) Guarded "`pause`": Each `pause` statement is guarded by the closest `abort` or `suspend` parent statement or, at the top level, by the Runtime System. The assembly code for `pause` statements is then the following:

```
    RPC = PAUSE_lbl ; RFP = FP
    FP = FP[0]^n ; goto parent_guard_lbl
PAUSE_lbl:
```

The context of the thread is saved in two data registers: the resume point is saved in RPC (Resume Program Counter) and the current frame pointer in RFP (Resume Frame Pointer). The frame pointer is then popped of as many levels (given in n) as to retrieve the frame pointer of the guarding `abort`/`suspend` statement. At the address $parent\_guard\_lbl$ is the code responsible for managing the paused thread. The values $parent\_guard\_lbl$ and n are defined at compile time. In the following, we shall use the simpler assembly instruction "`resume` res" where res is the address where the `pause` has to resume at the next reaction step. The "`resume` res" instruction is comparable to that of the `yield()` instruction of languages implementing co-routines [5] (Java, Python, Lua...). Thus, the translation of the `pause` statement becomes:

```
    resume PAUSE_lbl
PAUSE_lbl:
```

2) Guard Statements: The task of `abort` and `suspend` statements comprises of guarding the execution of their bodies. Their compilation has to provide the assembly code for managing the nested `pause` statements through the callback mechanism described before and the data registers RPC and RFP. The compilation of `abort` statements produces the following code:

```
    P
    goto END_lbl
  GUARD_lbl:
    if child == [] then
      child = (RPC, RFP)::child ; resume RES_lbl
    else
      child = (RPC, RFP)::child ; stop
    endif
  RES_lbl:
    if not(C) then
      start_all(child) ; child = [] ; stop
    endif
  END_lbl:
```

The assembler block starting at the addresses `RES_lbl` is responsible for resuming the execution of the paused threads inside P, under the condition C. Each time a thread of P is paused, it is added to a list "`child`" (a local variable stored in the current frame). These threads shall be passed through the registers RPC and RFP and managed from the address `GUARD_lbl`. This label has to be provided when compiling `pause` statements inside P, so that the control jumps to this very address each time a thread is paused. When the first thread is registered (`child == []`), the `abort` statement has also to register itself to the parent guard (`resume RES_lbl`). The translation of `suspend` statements is hardly different so that when the condition C holds, the paused threads are not started. Instead, the list is kept and the `suspend` statement registers itself again to the parent guard:

```
  RES_lbl:
    if C then
      resume RES_lbl
    else ...
```

In the case of the `abort` statement, P is simply not restarted and the control flows to the rest of the program.

The translation of `weak abort` statements is slightly more complex since the abortion of P has to take place one logical instant later than it would be in the case of an `abort` statement. Nevertheless, this translation is similar enough to that of the `abort` statement for doing without such extra details here.

D. Instantiate Module as Function Call

The instantiation of synchronous modules can be compiled as an *almost* classical function call: caller context and parameters are passed through a new stack frame and control jumps to the first instruction of the module. The only difference with respect to the classical function call of sequential programs is that the guard context has to be stored (since the module can be reentered several times along the successive reactions):

As to cope with our calling conventions, the declaration of a module `module` $mod\_name(S_1, \ldots S_n)$ $\{P\}$ generates the following code:

```
  mod_name:
    P
    ret = FP[0] ; FP = FP[1] ; goto ret
  GUARD_lbl:
    ret = FP[2] ; FP = FP[3] ; goto ret
```

Where `GUARD_lbl` is the parent guard label of the module body P. It is necessarily provided for the compilation of any `pause`, `abort` or `suspend` statement of P.

E. About Scopes

Schizophrenia is a usual issue in the compilation of imperative synchronous programs [2]. It comes from the fact that, because of loops, several instances of a same variable can coexist simultaneously (in a same reaction step). The problem does not arise here thanks to the use of stack frames: each time a scope is re-entered a new data frame is allocated. The separation between control and data is at the heart of our modular compilation technique since it allows to simultaneously run several threads sharing the same code but working on distinct data.

IV. Dealing with Signals

Over the rigid framework provided by the structural translation of synchronous modules into cooperative sequential threads, we shall now provide a solution to deal with runtime causality induced by signals. Let us consider the following example where `I` is an input signal, `O` is an output signal and `S1` and `S2` are local signals:

```
1    { if present(I)  then emit S1;
2      if present(S2) then emit O }
3 || if present(S1) then emit S2
```

The correct execution of this program starts with the beginning of the first thread (at line 1), then carry on to the second one (at line 3) and finally resume the first one (at line 2). This program is a friendly illustration of the problem. In real life we might face some nastier configurations where such statements are nested in different locations and different modules in the program. Considering modular compilation, there is no solution for statically scheduling our compiled threads in the general case. In our approach, causality issues are then resolved at execution time.

A. Implementation of Signals

At the end of a reaction step, the status of any signal has to be either *present* or *absent*. Inside a reaction, every signals but inputs remain *undefined* until reaching an `emit` statement or reaching a state of the program where all potential emissions are discarded. We propose the following approach: before a signal is read or tested, we check its status. If it is *undefined*, then the execution of the thread is suspended. Each signal maintains a list of pending threads, so that they are immediately restarted as soon as a definitive value or status of the signal is determined (observer pattern). For this purpose, we use an assembly instruction "`wait` S", defined as a macro for the following code:

```
    if S.status == UNDEFINED then
      S.pending = (RES_lbl, FP)::S.pending
      stop
    endif
  RES_lbl:
```

The emission or absence of a signal is realized through the assembly instruction "`emit` S, $status$, $value$", defined as a macro for the following code:

```
S.status = status
if status == PRESENT then
  S.value = value
endif
start_all(S.pending)
```

B. Immediate Reaction to Absence

Reacting to the absence of a signal depends on the global behavior of the program. Our approach involves the Runtime System at the top level which maintains a list "*pending*" of undefined signals. At some point some signals have to be declared absent so as to achieve the current reaction step. However, **not all** the pending signals can be declared as absent. Let us consider the following example:

```
   if present(S1) then emit S3 else emit S2
|| if present(S2) then emit S3
|| if present(S3) then emit O
```

where S1, S2 and S3 are local signals. The execution of this program leads to a point where the threads are blocked on these signals. At this very point, only S1 is safe to be set absent since S2 and S3 still have potential emitters.

1) Potential Emissions: Our solution relies on a local knowledge of the potentially emitted signals at each point of the program. We use reference counters on signal to indicate which signal can be safely set absent. We use the instructions "`can` S" to mark a signal as possibly emitted in the current thread (S`.refcount = `S`.refcount+1`). We use the instruction "`cannot` S" to mark that the control just reached a point where a signal cannot be emitted any more in the current instant and by the current thread (S`.refcount = `S`.refcount-1`). This strategy is closely related to the "*Can*" sets used in the Constructive Behavioral Semantics of Esterel [2].

Different semantics for causality issues in Esterel were discussed in [4]. This paper presents several variations of the constructive semantics of the Esterel language. These semantics are more or less restrictive depending on the way the *Can* sets are computed by so-called "*Potential Functions*". In our approach, we compute our *Can* sets with the help of the Potential Function corresponding to the v3 version of the Esterel semantics.

2) Reference Counter Policy: At compile time, we **locally** compute the *Can* set of potentially emitted signals for each statement of the program. We then generate the `can` statements at each resumption point of the program (i.e for each `pause` statement), so that a thread immediately declares its potentially emitted signals as soon as it is started or resumed. Finally, we generate the `cannot` statements at each point of the target code where the *Can* set decreases. For example, the insertion of `can` and `cannot` statements in the code of the first column will produce the code of the second column:

```
pause;                      pause;
                            can A; can B; can C;
emit A;                     emit A; cannot A;
if present(S) then          if present(S) then
                              cannot C;
  emit B                      emit B; cannot B
else                        else
                              cannot B;
  emit C;                     emit C; cannot C
pause                       pause
```

A signal can thus be safely declared absent when its reference counter is equal to zero and when the list of active threads is empty (which means that any active threads had the opportunity to increase the reference counter of the concerned signals before suspending their execution). The global information about the potentially emitted signals is thus obtained by the execution of all the active threads providing local partial information. This strategy for the generation of `can` and `cannot` instructions is not unique and leaves room for optimizations.

In the context of a modular execution, the correct compilation of programs requires some minimal knowledge about the modules. So as to identify the potentially emitted signals, we then need to know 1. which parameters can be emitted at the first reaction of the module and 2. if the module can be instantaneous. This information can be carried by a type system and stored in a header file. When a module is about to return, it also requires the list of signals that are potentially emitted by the caller after the termination of the callee. This information is actually passed on stack frames.

V. Conclusion

We presented a new strategy for compiling Esterel-like modules into sequential cooperative threads. The static scheduling induced by the syntax is ensured by a structural compilation into sequential code. The dynamic behavior induced by the inter-process communications is solved by the definition of a protocol for the emission of signals. This technique has been implemented as a lightweight compiler where consumption and recycling of memory in the generated code has been a particular focus.

Our work can further be extended in various direction. We shall investigate some optimizations around the reference counting policy or the minimization of context switches. We could also work on some architecture-dependent implementations like multi-core processors. Questions also arise from the theoretical point of view: synchronous programs are traditionally used for modeling finite and static systems. However, this modular compilation scheme lets us to make the language much more flexible. It can thus be slightly adapted so as to cope with recursive, potentially infinite synchronous systems as well as higher-order programming and dynamically modifiable systems. To provide the same degree of correctness-by-construction as modern Esterel-like compiler, our technique could additionally be paired with a modular analysis associating each separately compiled module with a profile declaring an abstraction of its behavior.

References

[1] G. Berry. A hardware implementation of pure Esterel. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.

[2] G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org, 1999.

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

[4] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, INRIA, 1998.

[5] M. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7), 1963.

[6] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior reactive kernel. Research Report 3732, INRIA, 1999.

[7] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE)*, 2008.

[8] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Co-Design (MEMOCODE)*, Mont Saint-Michel, France, 2003.

[9] K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In *IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*, Braga, Portugal, 2006.

[10] J. Zeng and S. Edwards. Separate compilation for synchronous modules. In *International Conference on Embedded Software and Systems (ICESS)*, Xian, China, 2005.