MODULAR COMPILATION OF SYNCHRONOUS PROGRAMS

Klaus Schneider, Jens Brandt, and Eric Vecchié
University of Kaiserslautern
Department of Computer Science
Reactive Systems Group
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://rsg.informatik.uni-kl.de

Abstract: We present a new method for modular compilation of synchronous programs given in imperative languages like Quartz or Esterel. The main idea of our approach consists of computing sequential jobs that correspond with control flow locations of the program. Each job encodes that part of an instantaneous reaction that is triggered by the activation of the corresponding control flow location. The special consideration of the initial job that is executed at initial time yields a simple method for modular code generation.

Keywords: synchronous languages, modular compilation

1. INTRODUCTION

Synchronous languages [13], [2] like Esterel [3] and its variants [14], [19] are particularly interesting for system design: First, it is possible to generate both efficient software and hardware from the same synchronous program. Second, it is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis [15]. Third, the formal semantics of these languages allows one to formally prove (1) the correctness of the compilation and (2) the correctness of particular programs with respect to given formal specifications [18], [19], [21].

Although several success stories have been reported [12], there is still a need for further research on efficient and modular compilation of synchronous languages. In the past years, several different compilation techniques have been developed [9], [17], [11]:

Please use the following format when citing this chapter:

Schneider, K., Brandt, J., Vecchié, E., 2006, in IFIP International Federation for Information Processing, Volume 225, From Model-Driven Design to Resource Management for Distributed Embedded Systems, eds. B. Kleinjohann, Kleinjohann L., Machado R., Pereira C., Thiagarajan P.S., (Boston: Springer), pp. 75–84.

- The first compilers translated the program to an extended finite state machine whose transitions are endowed with corresponding code fragments [6]. The disadvantage is the potential state-explosion problem; the advantage is the very fast execution time of the generated code.
- Polynomial compilation was first achieved by a translation to equation systems [16], [18], [21] that symbolically encode the automata. The idea behind this approach is to consider control flow locations instead of entire control states[1]. This approach is successfully used for hardware synthesis and it is still the core of commercial tools [12], although the generated software is sometimes comparably slow.
- A third approach has been followed by the Saxo-RT compiler [8], [7] of France Telecom, which translates the program into an event graph. Hence, an event driven simulation scheme can be used to generate code, which is compiled into efficient C code.
- A fourth approach is based on the translation of programs into concurrent control data flow graphs [11], [17], [9], [10], [23], whose sizes depend 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 subtree.

All of the above approaches have been developed to optimize the compilation time, as well as the size and the execution time of the generated code. However, with the exception of [23], essentially none of the above compilation techniques considered a modular compilation, which is standard for all sequential programming languages.

Modular compilation of synchronous programs is not at all straightforward: A previously compiled module may start or end with an incomplete macro step whose micro steps can interact with the micro steps of later added modules. Hence, to achieve a modular compilation, the *surface* of each module must be known: The surface [4], [22] of a statement consists of those micro steps that are executed at initial time before the first control flow location is reached. Surfaces are the essential information for combining pre-compiled statements.

For this reason, we have developed a completely new compilation technique, which has different advantages [20]. Our compiler splits the given program into so-called jobs that correspond with the control flow locations of the program. Hence, we simply execute those jobs that correspond with the currently active control flow locations. To this end, we have to take care of mutual dependencies that have to be checked by causality analysis. An important sim-

---

[1] We distinguish between a control flow state that consists of a boolean vector of control flow locations. A control flow location is a statement of the program that can hold the control flow for an instant of time. In case of Esterel, control flow locations are essentially *pause* statements.

plification is obtained by our compilation technique since each job consists of purely sequential code.

For modular compilation, the job-based compilation technique has the advantage that the surface of the compiled module is explicitely given as the unique initial job. Thus, modular compilation is basically achieved by taking the union of the set of jobs and declaring the new initial job as the new surface.

The paper is organized as follows: in the next section, we briefly describe the Esterel/Quartz language that we consider in this paper. We then define the syntax and semantics of an intermediate language that we use to represent the sequential jobs, and we explain the key idea of our job-based compiler. After this, we illustrate the job-based compilation by means of a small example. Then, we explain in detail how modular compilation can be achieved with the job-based code. Finally, we discuss the advantages of our new compilation technique for modular compilation and conclude with a short summary. Details of the compiler are given in [20].

2. THE SYNCHRONOUS LANGUAGE Quartz

Quartz [18], [19], [20] is a descendant of Esterel that shares its basic model with its ancestor Esterel. In this paper, we rely on the common statements and therefore only consider the following:

DEFINITION 1 **[Statements of Quartz]** *The set of statements of* Quartz *is the smallest set that contains the following statements, provided that S, S_1, and S_2 are also statements of* Quartz*, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and α is a type:*

- *nothing (empty statement)*
- *emit x and emit next(x) (boolean event emissions)*
- *$y = \tau$ and next(y) $= \tau$ (assignments)*
- *ℓ : pause (consumption of time)*
- *if(σ) S_1 else S_2 (conditional)*
- *$S_1; S_2$ (sequential composition)*
- *$S_1 \parallel S_2$ (synchronous concurrency)*
- *do S while(σ) (iteration)*
- *[weak] suspend S when [immediate](σ) (suspension)*
- *[weak] abort S when [immediate](σ) (abortion)*
- *$\{\alpha\ y;\ S\}$ (local variable y with type α)*

There are two kinds of (local and output) variables in Quartz, namely *event* and *state variables*: State variables y are persistent, i.e., they store their current value until an assignment changes it, while event variables take a default value if no assignment is made. Executing a delayed assignment *next*(y) $= \tau$ means to evaluate τ in the current macro step (environment) and to assign the obtained value to y in the following macro step. Immediate assignments update y in the current macro step and are therefore rather equations than assignments. As most events are of Boolean type, we use the statements *emit* x and *emit next*(x) as macros for $y =$ true and *next*$(y) =$ true, respectively.

There is only one basic statement that defines a control flow location, namely the *pause* statement[2]. For this reason, we endow *pause* statements with unique Boolean valued *location variables* ℓ that are true iff the control is currently at location ℓ : *pause*.

The semantics of the statements is the same as in Esterel. Due to lack of space, we do not describe their semantics in detail, and refer instead to [19], [18], and, in particular, to the Esterel primer [5], which is an excellent introduction to synchronous programming.

3. COMPUTING JOBS FOR PROGRAMS

In this section, we describe the computation of an equivalent set of jobs for a given Quartz program. As already outlined, the overall idea of the proposed code generator is as follows: For each control flow location ℓ of the program, a job S_ℓ is computed that has to be executed iff the control flow resumes the execution from location ℓ. Of course, several jobs may have to be executed in one macro step since several locations can be active at once.

3.1 THE JOB LANGUAGE

In principle, a job S_ℓ consists of a set of guarded actions and guarded schedule statements (see below) to implement the data flow and the control flow of the program, respectively. However, we do not compute simple sets of guarded statements. Instead, we additionally use conditional and sequential statements to allow sharing of common conditions. Moreover, we use statements for barrier synchronization to implement the concurrency of synchronous programs.

DEFINITION 2 **[Job Language]** *The set of* Job *statements is the smallest set that contains the following statements, provided that S, S_1, and S_2 are also* Job *statements, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and λ is a lock variable (having integer type):*

- `nothing` *(empty statement)*
- `emit` x *and* `emit next`(x) *(event emissions)*
- $y = \tau$ *and* `next`$(y) = \tau$ *(assignments)*
- `init`(x, τ_0) *(initialize local variable)*
- `schedule`(ℓ) *(resumption at next reaction)*

---

[2] To be precise, immediate forms of suspend also have this ability.

- `reset`(λ) *(reset a barrier variable)*
- `join`(λ) *(apply for passing barrier)*
- `barrier`(λ, c) *(declare barrier λ)*
- `if`(σ) S_1 `else` S_2 *(conditional)*
- $S_1; S_2$ *(sequential composition)*

Note that there is no longer a parallel statement and also the abort/suspend statements are no longer required. Moreover, there are no loops, since we can implement them by the help of schedule statements (explained below). Furthermore, all job statements are instantaneous[3].

The atomic statements `nothing`, `emit` x, `emit next`(x), $y = \tau$, and `next`$(y) = \tau$ have the same meaning as in Quartz programs. The meaning of conditionals and sequences is also the same as in Quartz. The statement `init`(x, τ_0) replaces a local variable declaration as follows: when executed, it first removes x from the current context as well as pending (delayed) assignments to x, and then gives x the initial default value τ_0.

The `schedule`(ℓ) statement corresponds with a control flow location ℓ of the Quartz program. When executed, it simply puts the label ℓ in the schedule, so that the runtime environment will execute the corresponding job S_ℓ in the next reaction step. Note, however, that `schedule`(ℓ) is instantaneous, so that `schedule`(ℓ_1); `schedule`(ℓ_2) will at once put both ℓ_1 and ℓ_2 to the schedule for the next reaction step.

The statements `reset`(λ), `join`(λ), and `barrier`(λ, c) are used to implement concurrency based on *barrier synchronization*. `barrier`(λ, c) declares a barrier with an integer lock variable λ and an integer constant c as threshold. Executing this statement checks whether $\lambda \geq c$ holds, and if so, it immediately terminates, so that a further statement S can be executed in a sequence like `barrier`$(\lambda, c); S$. If $\lambda < c$ holds, the execution stops, so that the control thread terminates.

Executing `reset`(λ) simply resets $\lambda = 0$, and `join`(λ) first increments λ and then invokes a function f_λ that is associated with the barrier whose lock variable is λ. Usually (and in our compiled jobs always), it is the case that the code of function f_λ is a sequence `barrier`$(\lambda, c); S$ with some statement S.

Using the statements for barrier synchronization, it is straightforward to execute parallel code on a uniprocessor machine: We associate with each parallel statement a barrier with lock variable λ and threshold $c = 2$ that is reset when the parallel statement is started. When a thread of the parallel statement terminates, it executes a `join`(λ) statement. If both threads have executed their final `join`(λ) statements, the barrier will be passed, so that the code following

---

[3] The job language is therefore also a synchronous language on its own, which is however not meant to be offered to the programmer. Instead, it is used as an intermediate language that could, in principle, be the target for many synchronous languages.

the associated `barrier`(λ, c) statement in the function f_λ associated with the barrier can be executed.

The implementation of the barrier synchronization for other architectures may (and must) be different. Hence, it depends on the platform that is used to execute the program, while our jobs remain architecture-independent. Different implementations for barrier synchronization already exist [1] for hardware, software on multiprocessors, and software on uniprocessors, so that our jobs can be executed on all of these platforms.

3.2 COMPUTING JOBS

The computation of the jobs of a statement is done in a single pass using a recursive function $\mathsf{Jobs}(\cdot,\cdot,\cdot)$. To compile a statement S, we start the function call $\mathsf{Jobs}(S, \textit{nothing}, \{\})$, which computes a tuple $(S_\alpha, \mathcal{P}, \mathcal{F})$ with the following meaning:

- S_α is the surface statement of S, i.e., that code that is executed when S is initially started (which is often viewed as being started from an invisible 'boot' control flow location ℓ_α).
- $\mathcal{P}$ is a set of pairs (ℓ, S_ℓ) such that S_ℓ is the job that is associated with control flow location ℓ.
- $\mathcal{F}$ is a set of pairs (λ, S_λ), where λ is the lock variable of a barrier and S_λ is of the form `barrier`$(\lambda, c); S'$ with some threshold c (hence, S' is the job that is executed when the barrier is passed).

The execution of the initial call $\mathsf{Jobs}(S, \textit{nothing}, \{\})$ will produce subsequent calls $\mathsf{Jobs}(S, S_\eta, J)$ with statements S, S_η with the following meaning: During the function calls, the statement that has to be compiled has been transformed to an equivalent one that is now of the form $S; S_\eta$. Moreover, the set J is either $\{\}$ or a singleton set $\{\lambda\}$. In the latter case, we have to immediately execute `join`(λ) to apply for passing the barrier λ as soon as $S; S_\eta$ terminates. If λ is large enough, the barrier can be passed and the job S_λ associated with the barrier will be immediately executed.

In principle, our compilation procedure performs a symbolic execution of the statement, and each recursive call corresponds with a SOS rule that defines the semantics of Quartz, which allows us to easily verify its correctness. The recursion is made primarily on S, and secondarily on S_η. Details of the compilation are given in a forthcoming publication and also in [20].

4. AN ILLUSTRATING EXAMPLE

A difficult example program (with event input `i` and event outputs `a`, `b`, and `c`) is given in Figure 1. This program suffers from a schizophrenia problem, since the scope of the declaration of the local variable x can be left and re-entered in

```
module Schizo(event i,&a,&b,&c) {
  loop
    {bool x;
      if(i) {
        next(x) = true;
        q1:pause;
      }
      abort {
       emit a; || if(not(x)) emit b;
                  else q2:pause;
       emit c;
       q3:pause;
     } when(not(i));
    }
}
```

Figure 1. A Challenging Example with a Schizophrenic Local Declaration.

```
void f__start() {
 init(x,false);
 if(i) {
    next(x) = true;
    schedule(q1);
 } else {
    reset(_lmb4);
    emit a;
    join(_lmb4);
    if(~x) {
       emit b;
       join(_lmb4);
    } else
       schedule(q2);
 }
}
```

```
void f_q1() {
 reset(_lmb4);
 emit a;
 join(_lmb4);
 if(~x) {
    emit b;
    join(_lmb4);
 } else
    schedule(q2);
}


void g__lmb4() {
 barrier(_lmb4,2);
 emit c;
 schedule(q3);
}
```

```
void f_q2() {
 if(~i) {
   init(x,false);
   reset(_lmb4);
   emit a;
   join(_lmb4);
   if(~x) {
      emit b;
      join(_lmb4);
   } else
      schedule(q2);
 } else
   join(_lmb4);
}
```

```
void f_q3() {
  if(~i) {
    init(x,false);
    reset(_lmb4);
    emit a;
    join(_lmb4);
    if(~x) {
       emit b;
       join(_lmb4);
    } else
       schedule(q2);
  } else {
    init(x,false);
    next(x) = true;
    schedule(q1);
  }
}
```

Figure 2. Sequential Jobs for Module Schizophrenia (Figure 1).

the same macro step. It is well-known that a statement may be entered more than once in a single macro step if the module is called in a surrounding context where the module is nested in several loops.

Figure 2 shows the resulting jobs that are obtained by compilation of this module. As can be seen, our code generator has constructed functions `f_q1`, `f_q2`, and `f_q3` for each control flow location as well as for the boot location (function `f__start`). Moreover, there is a continuation function `g__lmb4` to implement the termination of the parallel statement.

Note that the schizophrenic local declaration is correctly implemented due to the initialization statements that are called in the correct order.

5. MODULAR COMPILATION

Since a previously compiled module may start or end with incomplete macro steps, it is possible that these micro steps can interact with the micro steps of of a surrounding calling module. Hence, we have to consider the potentially incomplete initial and final macro steps of the modules in order to compile them in a modular way. In particular, we have to combine the incomplete macro steps to a complete macro step of the entire module.

The job-based compilation technique presented above lends itself well for this purpose: Assume we have to compile a module M with body statement S for later use. To this end, we first replace the usually used initial function call $\mathsf{Jobs}(S, \mathit{nothing}, \{\})$ for the module's body statement S with the extended function call $\mathsf{Jobs}(S, \mathit{nothing}, \{\lambda_S\})$ with a new lock variable λ_S. Hence, when M terminates, it immediately calls a corresponding continuation function g_{λ_S} with job statement S_{λ_S}. Since g_{λ_S} is not available in the compiled code of M, the runtime environment has to add such a function (with $S_{\lambda_S} :=$ `nothing`) when M is executed without a further context module.

Now assume M is instantiated in a surrounding module M'. Then, the job-based compilation function works as follows: The function call $\mathsf{Jobs}(S, S_\eta, J)$ is replaced by (1) $\mathsf{Jobs}(S, \mathit{nothing}, \{\lambda_S\})$ and (2) $\mathsf{Jobs}(S_\eta, \mathit{nothing}, J)$. Since (1) is what we already compiled in the previous compilation run for module M, we can simply read[4] the compilation result $(S_\alpha, \mathcal{P}, \mathcal{F})$ from the file that contains the results of the previous compilation run. Call (2) is obtained by normal compilation and will thereby generate a triple $(S_\alpha^\eta, \mathcal{P}^\eta, \mathcal{F}^\eta)$. The final result is then $(S_\alpha, \mathcal{P} \cup \mathcal{P}^\eta, \mathcal{F} \cup \mathcal{F}^\eta \cup \{(\lambda_S, \mathtt{barrier}(\lambda_S, 1); S_\alpha^\eta)\})$, i.e., we use the initial job S_α^η of S_η as the continuation function for the barrier λ_S.

Hence, modular compilation can be simply integrated with the job-based compilation technique. The only fact we have to verify is that $\mathsf{Jobs}(S, S_\eta, J)$ is equivalent to $(S_\alpha, \mathcal{P} \cup \mathcal{P}^\eta, \mathcal{F} \cup \mathcal{F}^\eta \cup \{(\lambda_S, \mathtt{barrier}(\lambda_S, 1); S_\alpha^\eta)\})$, where (1) λ_S is a new barrier variable, (2) $(S_\alpha, \mathcal{P}, \mathcal{F}) = \mathsf{Jobs}(S, \mathit{nothing}, \{\lambda_S\})$, and (3) $(S_\alpha^\eta, \mathcal{P}^\eta, \mathcal{F}^\eta) = \mathsf{Jobs}(S_\eta, \mathit{nothing}, J)$ holds.

Note that during the compilation of the context module M', the jobs $\mathcal{P}$ that have been generated by the previous compilation run of M may be modified due to surrounding abortion or suspension statements. These statements have to abort or suspend the job's execution whenever the corresponding abortion or suspension condition holds. Since this is done in the usual job-based compilation as well, we need not discuss this issue further, but we want to note that it may be necessary to modify the jobs $\mathcal{P}$. Moreover, several module calls to M may even require copies of the jobs $\mathcal{P}$.

Another problem is posed by causality analysis: Although all modules can be checked independently of each other, a complete causality analysis can be only performed after all modules have been linked together. Hence, causality analysis has to be done after the complete compilation. Nevertheless, it may

---

[4] Clearly, substitutions may be necessary due to the given arguments of the module instantiation.

be additionally done as well on single modules after their local compilation in order to speed up the final causality analysis.

Equation-based code can be integrated in the modular compilation scheme as well: The compiler just wraps the equations into two jobs: All initial equations define the initial job, and the transition equations define the main job j_{main}. Both jobs conclude with a check whether at least one control flow location is active: If such a location exists, j_{main} is scheduled again, otherwise, the exit continuation function is joined.

6. SUMMARY

In this paper, a very simple code generation scheme has been presented that is based on splitting the given program into sequential jobs that correspond with the control flow locations of the program. Additionally, continuation functions are required in order to avoid an exponential blow-up of the code, and to efficiently execute parallel statements on uniprocessor systems.

In particular, we have shown in this paper that our compilation technique is suited for modular compilation, since the jobs explicitly contain the surface of the program given as the initial job `f_start`. Modular compilation is not as simple as known from sequential programming languages, since a reprocessing of the compiled module cannot be avoided. However, the main benefits of modularization remain: Compiled and potentially highly-optimized components can be distributed and reused. Moreover, they can be shared without revealing their source codes, since the generated jobs are a rather low-level (but still adaptable) description from which the original code cannot be reconstructed.

REFERENCES

[1] ANDREWS, G. *Concurrent Programming – Principles and Practice.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[2] BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. The synchronous languages twelve years later. *Proceedings of the IEEE 91*, 1 (2003), 64–83.

[3] BERRY, G. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT, 1998.

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

[5] BERRY, G. The Esterel v5_91 language primer, June 2000.

[6] BERRY, G., AND GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[7] CLOSSE, E., POIZE, M., PULOU, J., SIFAKIS, J., VENTER, P., WEIL, D., AND YOVINE, S. TAXYS: A tool for the development and verification of real-time embed-

ded systems. In *Conference on Computer Aided Verification (CAV)* (Paris, France, 2001), vol. 2102 of *LNCS*, Springer, pp. 391–395.

[8] CLOSSE, E., POIZE, M., PULOU, J., VENIER, P., AND WEIL, D. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. *Electronic Notes in Theoretical Computer Science (ENTCS) 65*, 5 (2002). Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[9] EDWARDS, S. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 21*, 2 (February 2002), 169–183.

[10] EDWARDS, S. ESUIF: An open Esterel compiler. *Electronic Notes in Theoretical Computer Science (ENTCS) 65*, 5 (2002). Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[11] EDWARDS, S., KAPADIA, V., AND HALAS, M. Compiling Esterel into static discrete-event code. In *Synchronous Languages, Applications, and Programming (SLAP)* (Barcelona, Spain, 2004).

[12] ESTEREL TECHNOLOGY. Website. http://www.esterel-technologies.com.

[13] HALBWACHS, N. *Synchronous programming of reactive systems.* Kluwer, 1993.

[14] LAVAGNO, L., AND SENTOVICH, E. ECL: A specification environment for system-level design. In *International Design Automation Conference (DAC)* (New Orleans, Louisiana, USA, 1999), ACM, pp. 511–516.

[15] LOGOTHETIS, G., AND SCHNEIDER, K. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)* (Munich, Germany, March 2003), IEEE Computer Society, pp. 196–203.

[16] POIGNÉ, A., AND HOLENDERSKI, L. Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin, 1995.

[17] POTOP-BUTUCARU, D., AND DE SIMONE, R. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Codesign (MEMOCODE)* (Mont Saint-Michel, France, 2003), IEEE Computer Society, pp. 227–236.

[18] SCHNEIDER, K. A verified hardware synthesis for Esterel. In *Workshop on Distributed and Parallel Embedded Systems (DIPES)* (Schloß Ehringerfeld, Germany, 2000), F. Rammig, Ed., Kluwer, pp. 205–214.

[19] SCHNEIDER, K. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)* (Newcastle upon Tyne, UK, June 2001), IEEE Computer Society, pp. 143–156.

[20] SCHNEIDER, K. The synchronous programming language Quartz. Internal Report to appear, Department of Computer Science, University of Kaiserslautern, 2006.

[21] SCHNEIDER, K., BRANDT, J., AND SCHUELE, T. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)* (2006).

[22] SCHNEIDER, K., AND WENZ, M. A new method for compiling schizophrenic synchronous programs. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (Atlanta, USA, November 2001), ACM, pp. 49–58.

[23] ZENG, J., AND EDWARDS, S. Separate compilation of synchronous modules. In *International Conference on Embedded Software and Systems (ICESS)* (Xian, China, 2005).