Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 15 pages

ESUIF: An Open Esterel Compiler

Stephen A. Edwards [1]

Computer Science Department
Columbia University
New York, USA

Abstract

I describe a new compiler infrastructure for imperative synchronous languages such as Esterel and ECL. Built on the SUIF 2 system, it includes a new intermediate representation for this class of languages that has simple semantics designed for easy implementation in hardware or software. I describe the structure of this new compiler, the intermediate representation, and how Esterel source is translated into this intermediate representation.

1 Introduction

ESUIF is a new compiler designed for research on synchronous imperative languages such as Esterel [6] and ECL [10]. Its design is modular and sufficiently flexible to be the basis for work on hardware synthesis, software synthesis, optimization, and verification.

The difficulty of compiling imperative synchronous languages such as Esterel and ECL motivated this work. Their semantics are subtle and complex, integrating concurrency, preemption, and instantaneous broadcast. To date, at least four substantially different approaches, automata [6], logic networks [1], control-flow graphs [7], and events [11], have been proposed, each with different advantages and drawbacks. None is clearly superior, and more work is needed. Unfortunately, none of these compilers is available in source form, and only two are available in binary form, limiting new research in the field. ESUIF, by contrast, is freely available [2] and designed for flexibility.

ESUIF compiles programs in Esterel and related languages using a series of refinement steps that can easily be used in different ways and extended. To support this, ESUIF is built on the SUIF 2 system from Stanford University [3],

[1] Email: sedwards@cs.columbia.edu
[2] www.cs.columbia.edu/~sedwards
[3] suif.stanford.edu

©2002 Published by Elsevier Science B. V.

which consists of a persistent, customizable object-oriented database implemented in C++ along with a variety of compiler-specific utilities such as mechanisms for symbol tables and a type system.

Esuif, like all compilers built on suif 2, consists of a collection of passes, such as dead code elimination, implemented as shared objects that can be dynamically linked and invoked under control of a simple scripting language. This makes it easy to make small additions and modifications to the compilation process without substantially rewriting the code.

To decouple passes and allow them to be reordered, esuif adopts the suif 2 approach of using the same database throughout the compilation process. Esuif uses at least three representations of an Esterel program during compilation. The front end described in Section 3 builds a very high-level abstract-syntax-tree-like representation of the source Esterel program containing, e.g., the *LoopStatement* and *SuspendStatement* objects of Fig. 2. Dismantlers described in Section 6 expand these high-level constructs into the low-level ones listed in Fig. 3 and described in Section 5. Additional passes transform these into statements that can be optimized by existing suif 2 passes.

The modular nature of esuif allows it to easily support other compilation paths. A proposed path translates the output of the ecl compiler into the initial database, the dismantling passes transform this into primitives, and another set of passes build a logic-level netlist from this. The unified nature of esuif's database enables optimizations unavailable to the existing ecl compiler, which forcibly separates Esterel-like constructs from the C-like ones.

The overall compilation process is built modularly for flexibility, but is less efficient than a more direct implementation. Passes can be added and removed from the compilation chain by simply changing the top-level script; no recompilation is necessary. However, the dynamic nature of the executable along with the more dynamic nature of the database tends to increase compilation times and memory consumption. These issues have not been a problem for Esterel, however, since the quality of generated code—not compilation speed—has been the main challenge.

2 Introduction to Esterel

In this section, I briefly introduce the Esterel language and its semantics since these are what esuif was designed to support. More comprehensive introductions can be found elsewhere [4,5]. This discussion is also relevant to compiling the ecl language [10], which is essentially C augmented with the Esterel constructs described here.

Esterel is an imperative synchronous language, meaning an Esterel program is a sequence of statements that execute in lockstep with a global clock. Most Esterel statements execute instantaneously, i.e., run and terminate in the same cycle, but some statements provide explicit delays. For example *pause* waits for one cycle before terminating.

Esterel programs communicate via signals: flags that are either present or absent in each clock cycle. Two statements manipulate signals: *emit S* makes signal S present the instant it runs; S is absent otherwise. The *present* statement is an if-then-else that tests signal presence. Thus,

```
present A then pause; pause; emit B end; emit C
```

means "if A is present in the first cycle, emit B and C in the third cycle, otherwise emit C in the first cycle."

Esterel is a concurrent language. Sequences of statements separated by double vertical bars—`||`—run concurrently. Esterel's signal coherence rule imposes a constraint on the order in which concurrently-running statements may execute: within a cycle, any *emit* statement for a signal must run before any *present* statement may test it. Communication is instantaneous between concurrently-running statements. Thus,

```
  emit A; present B then emit C end
||
  present A then emit B end
```

emits C in the first cycle because the presence of A is instantaneously communicated to second *present* statement, which immediately emits B and causes the first *present* to emit C.

Esterel's infinite loop statement *loop* restarts its body instantly after it terminates. Thus,

```
loop emit A; pause; emit B end
```

emits A in the first cycle, and both B and A in all subsequent ones. Since Esterel insists each cycle's computation is finite, a loop body must take at least one cycle, perhaps by including a *pause*.

Esterel provides two ways to escape from infinite loops. A loop can terminate itself by exiting a trap: executing *exit T* inside the body of a *trap T* statement terminates the *trap* statement. When an *exit* is executed concurrently with other threads of control within the body of a *trap*, the other threads are allowed to run until they reach a *pause* or equivalent before the *trap* terminates. If multiple *exit*s are executed within nested traps, the outermost *trap* takes precedence.

An *abort when S* statement terminates a loop (or any group of statements) from outside. When signal S is present, the body of the *abort* is terminated before it has an opportunity to run. Thus,

```
abort loop pause end when S
```

waits for the next cycle in which signal S is present. This behavior is so common that Esterel has a shorthand for it—*await S*.

Like *abort*, the *suspend* statement prevents its body from running when certain signals are present, but unlike *abort*, *suspend* only delays the execution of its body, holding its state in limbo while the preempting condition holds.

One of the main challenges in compiling Esterel is identifying programs that are contradictory under Esterel's signal coherence rule, e.g.,

```
present A else emit A end    % A is present if it is absent
abort pause; emit A; when A  % A is not emitted if it was
```

This is even more complicated than it appears because Esterel permits a program to contain a contradiction provided the program can never get in a state where all statements involved in the contradiction can run simultaneously.

Implementing the *trap* and *exit* statements is another challenge. Since an *exit* statement terminates the body of its enclosing trap, it may terminate concurrently-running statements. The rule is that any concurrently-running statements continue until they terminate or encounter a *pause* before being terminated. Furthermore, multiple *exit* statements may be executed by concurrently-running threads in the body of the same *trap*. In this case, the outermost enclosing *trap* is executed, necessitating an arbitration mechanism.

3 The Front End

Esuif includes a front end that parses Esterel source files and builds a suif 2 database for it. The front end is divided into a suif-independent parser and an abstract syntax tree walker that builds a suif database. I used the antlr compiler generator [4] to synthesize the code for the entire front end.

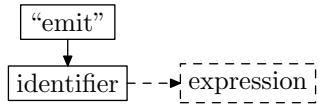
From a grammar file (Fig. 1a shows a fragment), antlr builds a recursive-descent parser that generates an abstract syntax tree (ast). The grammar for Esterel is clean and suif-independent because it uses antlr's ability to automatically generate code that builds an ast. Single-character annotations in the grammar direct this process: each symbol normally becomes an ast node, but a symbol followed by `^` becomes the root of a new subtree, and a symbol marked with `!` does not generate a node (used to avoid generating unwanted nodes for delimeters such as parentheses). Fig. 1b shows an ast fragment generated by the grammar in Fig. 1a.

The second half of the front end, for which antlr also generates much of the code, builds a high-level suif database from the ast and performs static semantic checks. It is specified using a grammar that directs a "walk" of the ast and contains C++ code for rules invoked during this process. Fig. 1c shows a fragment of the grammar file that handles the *emit* statement: it identifies a subtree rooted at an ast node labeled "emit," locates the named signal in a symbol table using the `find_signal` function, creates a suif database object for the *emit* statement, and signals an error if a pure signal is emitted with a value.

[4] `www.antlr.org`

```
emit
  : "emit"^ signalIdentifier
    (LPAREN! expression RPAREN!)?
  ;
```

(a)



(b)

```
| #( "emit"
     esig:ID
     {
       SignalSymbol *ss = find_signal(c, esig);
       Expression * e = 0;
     }
     ( e=expression[c] )?
     {
       EmitStatement *es = create_emit_statement( suif_env, ss, e );
       if ( e && ss->get_type()->get_base_type() == voidType ) {
         report_error(esig, "illegal valued emission of pure signal `%s'",
                      esig->getText().c_str());
       }
       st = es;
     }
  )
```

(c)

Fig. 1. (a) A fragment of the Esterel grammar written in ANTLR. The ^ marks "emit" as a root, and the two !s prevent ANTLR from building AST nodes for the parentheses. (b) The AST built from the fragment. Dotted lines indicate optional. (c) ANTLR tree parser code that generates a SUIF object for the fragment.

4 The Database

I built ESUIF using the SUIF 2 compiler infrastructure [5] developed at Stanford University so I could take advantage of its persistent, customizable object-oriented database and existing facilities for compiling imperative programs, such as a complete database schema for the C language (which ESUIF builds on) and basic optimizations such as constant propagation. Furthermore, its modular design is ideal for research because it allows new analysis and transformation passes to be added and tested independently. The result is a very flexible compiler, although not the smallest or fastest.

The SUIF 2 database is object-oriented and customizable. To add new object types to the database, a user describes them in an object-oriented schema language called "hoof" that resembles C++ class definitions. From the hoof file, a macro processor generates C++ code that is dynamically linked into the running system.

Fig. 2 shows a fragment of the ESUIF schema that represents Esterel's *loop* and *suspend* statements. It defines four object classes. A *BodyStatement* abstractly represents statements that contain a block of code. The definition of

[5] `suif.stanford.edu`

```
abstract BodyStatement : Statement {
  Statement * owner body in child_statements;
};

concrete LoopStatement : BodyStatement {};

concrete PredicatedStatement : BodyStatement {
  Expression * owner predicate in source_ops;
};

concrete SuspendStatement : PredicatedStatement {};
```

Fig. 2. A fragment of the "hoof" file describing SUIF 2 objects for Esterel's *loop* and *suspend* statements.

the class reads "the field `body` is a pointer to a *Statement* object that I own that is a member of my `child_statements` array field". For memory management purposes, each object in SUIF 2 may be referred to by many objects, but has exactly one owner. A *LoopStatement* is a *BodyStatement* that represents Esterel's *loop* statement. A *PredicatedStatement* is a *BodyStatement* with an expression that controls the execution of its body. The *SuspendStatement* is a *PredicatedStatement* that represents Esterel's *suspend* statement.

5 The Intermediate Representation

Fig. 3 lists the primitives used to represent the Esterel program after the first set of dismantling passes run. They were chosen to be easily translated into C statements, yet also provide enough of Esterel's high-level control constructs so that it is easy to translate Esterel into them. In particular, they provide facilities for preemption, resumption, exceptions, and concurrency.

These primitives more closely resemble those in traditional programming languages than those in the IC format used in the compilers from Berry et al. (first described in Gonthier's thesis [9]; I also describe them in a recent paper [7]). Much like control constructs such as *for* and *while* in traditional languages, each of these has a straightforward translation into sequences of assignments, conditionals and gotos, making them easy to translate to C.

The assignment statement, *if*, label, and *goto* statements have their usual meaning.

The other primitives in the IR deal with generating, catching, and recovering from exceptions in a way that enables preemption, resumption, and concurrency. Specifically, these primitives implement a variant of the numeric encoding of exceptions used in Esterel's formal semantics [2,3]. Each statement, after it has finished for the cycle, implicitly returns a small integer completion code that indicates whether it has terminated (0), paused (1), or exited a trap (2 and higher). This encoding elegantly captures priorities

var := *expr*
if (*expr*) { *stmts* } else { *stmts* }
Label:
goto *Label*
break *n*
continue
try { *stmts* } catch 2 { *stmts* } catch 3 { *stmts* } ...
resume { *stmts* } catch 1 { *stmts* } ...
parallel { *resumes* } catch 1 { *stmts* } catch 2 { *stmts* } ...
fork *Label1*, *Label2*, ...
join

Fig. 3. Statements in the ESUIF intermediate representation, designed to express Esterel's facilities for exceptions, concurrency, and pausing between cycles. The *try* statement runs its body. Executing an enclosed *break* statement passes control to a matching *catch* clause. *Resume* is a *try* that restarts its body after the last *break 1* statement when a *continue* statement is executed within one of its *catch* clauses. *Parallel* is a *resume* that runs the *resume* statements in its body concurrently. *Fork* and *join* are low-level initiators and collectors of concurrent behavior.

when concurrently-running statements try to exit multiple traps simultaneously. The completion code for concurrent statements is simply the maximum code from any of the statements.

In ESUIF, statement completion codes are encoded with control flow. Termination at level 0 is implemented by simply passing control to the next statement in sequence. When a sequence in a compound statement such as *if* terminates, it terminates the compound statement.

The *break* statement terminates at level 1 or higher, sending control to the innermost enclosing *catch* that matches the level. Fig. 4 shows how this mechanism is used to implement Esterel's *trap* statement.

Exception handling in Esterel differs from that in traditional languages in two ways. First, Esterel does not require "unrolling the call stack" because there is no stack; Esterel prohibits recursion. Throwing an exception in Esterel therefore usually becomes a simple unconditional branch; the *break* keyword was chosen to suggest this.

The main difference is that Esterel's concurrency enables two or more exceptions to be thrown simultaneously. The body of a *trap* statement may contain two or more threads, each capable of executing an *exit*. Esterel's semantics state the outermost trap takes precedence over any inner ones. The *parallel* statement does this arbitration implicitly and is the main source of complexity in Esterel's exception mechanism.

Termination at level 1 is special. It corresponds to Esterel's *pause* statement and needs the ability to return control to the statement following it in the next cycle. Together, the *break*, *continue*, and *resume* statement implement this behavior in the ESUIF IR as shown in Fig. 5.

```
trap T1 in          try {
  exit T1             break 2              goto Catch2;
                                           goto Catch0;
handle T1 do        } catch 2 {     Catch2:
  c := 1              c := 1                c = 1;
end                 }               Catch0:
```

(a) (b) (c)

Fig. 4. (a) An Esterel *trap* statement with handler. (b) Its translation into the ESUIF IR. The *exit* becomes a *break*. (b) Its translation into C. The *break* becomes a *goto*.

```
abort       resume {                    goto Ent
                                Cont:   switch (s) {
                                        case 0: goto State0;
                                        case 1: goto State1;
                                        }
  pause       break 1           Ent:    s = 0; goto Catch1; State0:
  pause       break 1                   s = 1; goto Catch1; State1:
                                        goto Catch0;
            } catch 1 {         Catch1:
              break 1                   s1 = 0; goto Catch1o; State0o:
when A        if (!A) continue          if (!A) goto Cont;
            }                   Catch0:
```

(a) (b) (c)

Fig. 5. (a) An Esterel *abort* statement with *pause*s. (b) Its translation into the ESUIF IR. (c) Its translation into C. Note the unusual placement of labels on the right to make the translation line-to-line. The `Catch1o` and `State0o` labels belong to the *resume* statement that encloses this one (not shown).

The *resume* statement has an implicit multiway branch, accessed by the *continue* statement, that is used to restart sequences of instructions. In C, each *break 1* statement expands into an assignment to the state variable for the *resume*, a branch to the *catch 1* handler for the *resume*, and a label.

The implementation of *resume* is complex but its behavior is simple: a *resume* simply runs its body and catches exceptions like a *try*. The difference is that a *resume*'s *catch* clause may execute a *continue* statement, which sends control to just after the last *break 1* executed in the body. For example, when the *resume* in Fig. 5b executes, it immediately executes the first *break 1* statement, which sends control to the body of the *catch 1* handler after setting the state variable `s`. The *catch* clause immediately executes a *break 1* and sends control to a handler in an enclosing *resume* (not shown).

```
trap T1 in                          try {
  trap T2 in                          try {
                                        parallel {
                                          resume {
    exit T1                               break 3 }
  ||                                      resume {
    exit T2                               break 2 }
                                        } catch 1 { break 1; continue }
  handle T2 do emit B end             } catch 2 { B := 1 }
handle T1 do emit A end             } catch 3 { A := 1 }
```

(a)

Fig. 6. Exceptions interacting with concurrency. (a) An Esterel program that emits A only in the first instant because the outermost trap takes priority over the inner one.

In the next cycle, this outer *resume* returns control to the handler (to the `State0o:` label). Then signal A is checked and if absent (i.e., the body was not aborted), *continue* is executed and sends control to just after the *break 1* executed in the last cycle by branching to the *switch* statement that checks the `s` state variable.

When multiple traps are exited simultaneously, such as in Fig. 6, the outermost one takes precedence. As in the formal semantics, the completion code of a group of threads is defined as the maximum completion code of all the threads, and the outermost *trap* is given the highest code.

This behavior is implicit in the IR *parallel* statement. After all *resume* statements (parallel threads) in a *parallel* have finished for the cycle, the body of the *parallel* terminates with the maximum of all the completion codes.

The behavior of ESUIF's *parallel* and the numerical encoding of traps differs slightly from those in IC. To simplify the semantics and implementation, ESUIF assigns a completion code to each trap and uncaught completion codes pass *parallel* statements unmolested (In IC, codes decrease if they pass through a *parallel* uncaught, so the same trap may use different codes). While IC's encoding keeps completion codes smaller (rarely do they exceed 3 in real programs), this is irrelevant in hardware and unlikely to be a problem in software.

The *fork* and *join* statements in Fig. 3 are only generated by dismantling *parallel* statements for software. Their semantics are simple: *fork* is a multiway *goto* that sends control to all of its labels. These independent threads of control run separately, subject to Esterel statement ordering rules, until they all reach a *join*. A group of Esterel threads have multiple entry points, corresponding to multiple *fork*s, but only a single *join*. Fig. 9 illustrates how these statements arise from the translation of a *parallel*.

Table 1
List of dismantling passes in the order they run. The first group transforms preemption statements into *abort*, the second dismantle statements into Fig. 3's primitives, and the third dismantle the primitives into C.

Statement	**Becomes**
await S	abort halt when S
do b watching S	abort b when S
do b upto S	abort b; halt when S
loop b each S	loop abort b; halt when S end
halt	loop pause end
sustain S	loop emit S; pause end
present S	if (S)
if *expr*	if (*expr*)
abort	resume…: see Fig. 7
loop b end	Again: b; goto Again
emit S	$S := 1$
nothing	(empty)
||	parallel { }
trap T in b handle T do h	try { b } catch k { h }
exit T	break k
pause	break 1
break k	goto *Catchk*
continue	goto *Continue*
try { b } catch k { h }	see Fig. 4
resume { b }	see Fig. 5
parallel	see Fig. 9

6 Dismantling

Dismantling is implemented as a collection of passes for flexibility. In general, each pass replaces each appearance of one type of instruction with a collection of lower-level instructions. While this is not the most efficient implementation (time is wasted traversing the program multiple times), it makes changing the dismantling process easy. This is a reasonable tradeoff for a research platform.

Table 1 lists the various dismantling passes in the order they run. Most are trivial, such as the pass that transforms Esterel's *present* statement, which can be a sequence of cases, into cascaded *if* statements. Dismantling *abort*, parallel, *trap*, and *exit* statements is more complicated.

6.1 Abort

All preemption statements are first translated into *abort* statements, which are then translated into a *resume* statement with conditionals to check the conditions, such as in Fig. 7b. In the first cycle, immediate conditions are

```
abort
  body
when

  case immediate p1 do s1
  case p2 do s2
  case e3 p3 do s3




end
```

(a)

```
if (p1) goto S1
c3 = e3
resume {
  body
} catch 1 {
  break 1
  if (p1) { S1: s1 }
  else if (p2) { s2 }
  else if (p3) {
    c3 := c3 − 1
    if (c3 ≤ 0) { s3 }
    else { goto Next }
  } else { Next: continue }
}
```

(b)

```
parallel {
  resume { body; break 2 }
  resume {
    if (p1) { break 5 }
    c3 = e3
    Again: break 1
    if (p1) { break 5 }
    else if (p2) { break 4 }
    else if (p3) {
      c3 := c3 − 1
      if (c3 ≤ 0) { break 3 }
      else { goto Next }
    } else { Next: goto Again }
  }
} catch 1 { break 1; continue }
catch 2 { /* body terminated */ }
catch 3 { s3 }
catch 4 { s2 }
catch 5 { s1 }
```

(c)

Fig. 7. Dismantling the *abort* statement. (a) The general case. (b) Expansion of (strong) *abort*. (c) The *weak abort* variant.

checked first (e.g., p_1), followed by the initialization counters for any counted delays (e.g., p_3). Finally, control passes to the *resume* and the body executes.

The *resume* implements the *abort*'s ability to resume execution from a pause reached at the last cycle. Executing a *pause* (actually a *break* 1) in the body sends control to the *catch* 1 clause, which starts with a *break* 1 that sends control to the surrounding *resume* (every thread has an outermost *resume*). In the next cycle, this *resume* returns control to the conditionals just after the *break* 1 in the *catch*. These check the preemption conditions and if none hold, the clause executes the *continue* statement, which resumes the body by returning control to just after the *break* 1 executed last cycle.

Weak abort has a more complex translation (Fig. 7c) that addresses two challenges: only immediate conditions are tested in the first cycle (the *break* 1 statement handles this), and preemption conditions are always checked after the body has terminated for the cycle, either normally or by pausing. The translation uses completion codes to defer the execution of handlers until the body has suspended or terminated for the cycle.

A *break* 2 was added after the body of the first resume in Fig. 7c to allow the body to terminate. Normally, control reaches the end of a body and implicitly signals completion code 0, but the the second thread, which checks the preemption conditions, always terminates at code 1 or higher, taking precedence over a code of 0 from the first thread. Therefore, the first thread signals normal termination with a *break* 2, which, unless a preemption condition holds, sends control to the *catch* 2 clause to terminate the *parallel*.

This complex translation, derived by examining the output of the IC-based compilers, appears necessary given the semantics of *weak abort*, but a simpler one would be possible if the preemption conditions were checked only after the body has paused, even in the first cycle.

6.2 Trap and Exit

Esuif dismantles Esterel's exception constructs *trap* and *exit* by first assigning an integer completion code to each trap so outermost traps—those with higher priority—have higher codes, then translating each *exit* into a *break* and each *trap* into a *try*. Fig. 8 illustrates this on a small example. With this choice of completion codes, the arbitration decision made at a *parallel* is a simple maximum computation.

A simple recursive calculation assigns a code to each *trap* statement, i.e., the code for a trap is one more than the highest code used by any of the statements it contains. The superscripts in Fig. 8 show the codes chosen on an example. Weak abort statements, because they also use the completion code machinery as explained explained earlier, consume one code for normal termination of their body plus one code per predicate.

Multiple traps caught by a single *trap* statement are all given the same code (e.g., T2 and T3 in Fig. 8). Esterel's semantics says such traps have equal priorities and that the handlers are invoked concurrently. Each trap has an additional flag tested by handlers for multiple traps.

6.3 Parallel

Normally, the effect of a completion code is simply to send control elsewhere and can be implemented with a *goto*, but the arbitration decision at a parallel—selecting the highest code—is not easily represented using simple software-like control flow. This behavior is easily synthesized in hardware; Berry's translation [3] includes a parallel synchronizer that does exactly this.

Dismantling *parallel* for software generates code that temporarily places completion codes into variables (c1 and c2 in Fig. 9), computes the maximum code, then performs a multiway branch that effectively translates the completion codes back into control flow.

The translation first adds *catch* clauses to each *resume* in the *parallel*, one for each code the *resume* can generate, that writes the code into a variable and jumps to a *join*. Next, an inverse clause is added to the *parallel* for each code it can produce. These clauses simply issue a *break* at the same code, effectively re-throwing the exception.

Two *fork* statements start concurrent threads of control in the translation of Fig. 9. When control reaches these it splits and goes to both labels. The two threads synchronize when they both reach the matching *join* statement.

```
trap T1[5] in
  trap T2[2], T3[2] in
    exit T1[5];
    exit T2[2];
    exit T3[2]



  handle T2[2] do emit A
  handle T3[2] do emit B

  end trap;
  weak abort[2]

    nothing

  when

    case[4] X do emit C
    case[3] Y do emit D









  end abort

handle T1[5] do emit E
end trap
```

(a)

```
try {
  try { T2 := false; T3 := false
    break 5
    T2 := true; break 2
    T3 := true; break 2
  } catch 2 {
    parallel {
      resume { if (T2) { A := true } }
      resume { if (T3) { B := true } }
    } catch 1 { break 1; continue }
  }
  parallel {
    resume {
      break 2
    }
    resume {
      Again: break 1
      if (X) { break 4 }
      else if (Y) { break 3 }
      else { goto Again }
    }
  } catch 1 { break 1; continue }
  catch 2 { /* body terminated */ }
  catch 3 { D := true }
  catch 4 { C := true }
} catch 5 {
  E := true
}
```

(b)

Fig. 8. Translating *trap*, *exit*, and *weak abort*. Superscripts indicate assigned completion codes. The dismantlers transform (a) into (b) the ESUIF IR. Traps T2 and T3 share code 2; the handler distinguishes them with variables. Trap T1 uses code 5 because the *weak abort* uses codes 2–4.

7 Conclusions and Future Work

ESUIF is an ongoing project. Currently missing is a backend that generates sequential C code from the concurrent intermediate representation. While implementing any of the known methods (e.g., automata or logic networks) would be straightforward, one of the goals of ESUIF is to provide an environment for experimenting with new techniques. Plans are underway to provide backends that statically unroll dynamically causal systems (i.e., those with apparent causality cycles), synthesize sequential code from a program dependence graph [8] representation, and variations on the event-based approach.

Another project currently underway will fuse the ECL front-end [6] with the ESUIF environment to provide integrated compilation flow for that language.

[6] `ecl.sourceforge.net`

```
parallel {      parallel {                          fork Start1, Start2
                                                    Cont: fork Cont1, Cont2

  resume {        resume {                            Start1: goto Entry1
                                                      Cont1: on s1 goto St01
    break 1         break 1                           Entry1: s1 := 0; goto C11; St01:
    break 2         break 2                             goto C21
  }               } catch 0 { c1 := 0; goto Join }    C01: c1 := 0; goto Join
                  catch 1 { c1 := 1; goto Join }      C11: c1 := 1; goto Join
                  catch 2 { c2 := 2; goto Join }      C21: c1 := 2; goto Join

  resume {        resume {                            Start2: goto Entry2
                                                      Cont2: on s2 goto St02
    break 1         break 1                           Entry2: s2 := 0; goto C12; St02:
    break 3         break 3                             goto C32
  }               } catch 0 { c2 := 0; goto Join }    C02: c2 := 0; goto Join
                  catch 1 { c2 := 1; goto Join }      C12: c2 := 1; goto Join
                  catch 3 { c2 := 3; goto Join }      C32: c2 := 3; goto Join

                                                    Join: join
                                                    c = max(c1, c2)
} catch 1 {     } catch 1 {                         on c goto C0, C1, C2, C3
  break 1         break 1                           C1: s0 := 0; goto C10; St00:
  continue        continue                            goto Cont
}               } catch 2 { break 2 }               C2: goto C20
                catch 3 { break 3 }                 C3: goto C30
                                                    C0:

    (a)               (b)                                 (c)
```

Fig. 9. Translating Parallel for software. (a) After dismantling from Esterel. (b) After annotation for software: catch clauses that save the completion code in variables added to the resumes, and clauses that trivially re-throw the completion code added to the parallel. (c) After dismantling into "parallel C." The *on goto* statement is a shorthand for a *switch* statement such as that in Fig. 5.

Currently, the C-like portions of the ECL language are forcibly separated from the Esterel-like portions and later linked, precluding certain optimizations.

I hope ESUIF will further research in these fascinating languages by forming a flexible platform freely available to the synchronous languages community.

Acknowledgements

Mike Kishinevsky and Ellen Sentovich provided valuable early feedback on this paper.

References

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

[2] Berry, G., *Preemption in concurrent systems*, in: *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science **761** (1993), pp. 72–93.
URL `ftp://cma.cma.fr/esterel/`

[3] Berry, G., *The constructive semantics of pure Esterel* (1999), book in preparation available at http://www.esterel.org.
URL `ftp://cma.cma.fr/esterel/constructiveness.ps.gz`

[4] Berry, G., "The Esterel v5 Language Primer," Centre de Mathématiques Appliquées (2000), part of the Esterel compiler distribution from http://www.esterel.org.

[5] Berry, G., "Proof, Language and Interaction: Essays in Honour of Robin Milner," MIT Press, 2000 .

[6] Berry, G. and G. Gonthier, *The Esterel synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), pp. 87–152.
URL `ftp://cma.cma.fr/esterel/BerryGonthierSCP.ps.Z`

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

[8] Ferrante, J., K. J. Ottenstein and J. D. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems **9** (1987), pp. 319–349.

[9] Gonthier, G., "Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel. [Semantics and models of execution of the synchronous reactive languages: application to Esterel]," Thèse d'informatique, Université d'Orsay (1988).

[10] Lavagno, L. and E. Sentovich, *ECL: A specification environment for system-level design*, in: *Proceedings of the 36th Design Automation Conference*, New Orleans, Louisana, 1999, pp. 511–516.

[11] Weil, D., V. Bertin, E. Closse, , M. Poize, P. Venier and J. Pulou, *Efficient compilation of Esterel for real-time embedded systems*, in: *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, California, 2000, pp. 2–8.

Esuif is available at `http://www.cs.columbia.edu/~sedwards/`
The suif system is available from `http://suif.stanford.edu/`
The antlr system is available from `http://www.antlr.org/`