# Refining Schizophrenia via Graph Reachability in Esterel

Jeong-Han Yun Chul-Joo Kim
*Division of Computer Science*
*KAIST*
*Daejeon, Korea*
*jeonghan.yun / chuljoo.kim@gmail.com*

Sunae Seo
*Samsung Advanced Institute of Technology*
*Samsung*
*Yongin, Korea*
*sunae.seo@samsung.com*

Taisook Han Kwang-Moo Choe
*Division of Computer Science*
*KAIST*
*Daejeon, Korea*
*han@cs.kaist.ac.kr / choe@kaist.ac.kr*

***Abstract*—Esterel is an imperative synchronous language for control-dominant reactive systems. The combination of imperative structures and the perfect synchrony hypothesis often result in schizophrenic statements. Previous studies explain the characteristics of schizophrenia as the instantaneous reentrance to block statements: local signal declarations and parallel statements. In practice, however, most instantly-reentered block statements do not cause any problems in Esterel compilation. In this paper, we refine schizophrenic problems in terms of signal emissions, and suggest an algorithm to detect harmful schizophrenia using reachability on control flow graphs (CFGs) in Esterel. Our algorithm performs well in analyzing practical programs. Moreover, it can be easily applied to existing compilers.**

## I. Introduction

Synchronous languages [1], [2] with the perfect synchrony hypothesis [3] help in the design of reactive real time systems. Because synchronous languages abstract asynchronous physical clocks to synchronous logical clocks, hardware timing issues can be easily managed.

Esterel [4]–[8] is an imperative synchronous language. Unlike dataflow synchronous languages [1], [2], [9], [10], Esterel supports useful imperative constructs that describe control-dominant systems; these include sequences, loops, suspensions, preemptions [11], exceptions, and declarations of signals and variables. These imperative constructs are also useful to specify software.

In Esterel, a `loop` statement terminates and restarts in the same instant. Hence a statement that is executed at a loop terminating instant can be re-executed when the loop restarts. This is called a schizophrenic statement [5], [12]–[14]. To avoid various problems [5] caused by schizophrenic statements, Esterel compilers must cure schizophrenic statements.

Several techniques have been proposed to cure schizophrenic problems [5], [12]–[16]. The basic concept of the existing techniques is based on code replications (or loop unrolling). Some optimized solutions [12]–[16] replicate only a part of the loop body that can be executed in the loop starting instant. The code replications, however, necessarily increase the compiled code size. Even worse, many compilers apply curing techniques to all `loop` statements without checking whether each loop has schizophrenic statements.

```
loop
  trap DATA in
    signal IS_DATA in
      [ await immediate iclk;
        pause;
        present IS_DATA then
          emit data_val;
          pause;
          emit data_val
        end;
        exit DATA
      ||
        await immediate idata;
        sustain IS_DATA
      ]
    end
  end;
  emit data_ready;
  await immediate [ not iclk ];
end
```

Figure 1. A `loop` statement in `atds100` of Estbench Esterel Benchmark suite [17]

The static analysis [18] investigates whether local signal declarations or parallel statements terminate or exit, and restart in the same instant. The key concept of the analysis is detection of instantaneous terminations using abstract interpretation [19]. Without considering the signal information, the method of Tardieu and Simone checks instantaneous paths structurally based on simple abstract semantics [18]. This analysis helps to reduce the size of cured results [13] by decreasing the number of statements to be replicated.

In some practical situations, however, the schizophrenic statements detected by Tardieu and Simone's approach [18] may not cause problems during compilation [20]. Figure I is an example of harmless schizophrenic signal declarations[1]. Since the "`signal IS_DATA in ... end`" block can be reentered in an instant, this statement is considered

[1]This example code is examined again in Section VI.

| | |
|---|---|
| `nothing` | no operation |
| `pause` | time consumption |
| `emit S` | signal emission |
| $p$;$q$ | sequence |
| `present S then` $p$ `else` $q$ `end` | signal test |
| `loop` $p$ `end` | nonterminating loop |
| $p \| \| q$ | parallel execution |
| `suspend` $p$ `when S` | suspending program |
| `signal S in` $p$ `end` | local signal decl. |
| `trap T in` $p$ `end` | exception decl. |
| `exit T` | exception raise |

Figure 2. Kernel statements of Pure Esterel

as schizophrenic according to Tardieu and Simone's analyzer [18]. However, the signal "`IS_DATA`" emitted by the statement "`sustain IS_DATA`"[2] cannot reach the signal test statement "`present IS_DATA`" in the same instant, because the "`pause`" statement consumes one instant.

From the viewpoint of Esterel semantics, we have observed that harmful behaviors of schizophrenic statements originate from signal emissions and variable assignments in the block statements: local signal declarations and parallel statements. To distinguish harmless statements from schizophrenic statements more precisely, we refine the characteristics of the following schizophrenic problems: (i) the wrong signal test caused by old and new instances of a local signal, and (ii) multiple execution of an "`emit`" statement during an instant. We have developed an algorithm that detects harmful schizophrenic statements by graph reachability on CFGs of Esterel programs based on our observation. The proposed algorithm is readily applicable to existing compilers.

The paper is organized as follows. In Section II, we introduce Esterel briefly. In Section III, we discuss schizophrenic problems. Focusing on actual problems that schizophrenic statements may cause, we refine characteristics of schizophrenic problems in Section IV. In Section V, we present our detection algorithm based on the refined characteristics. Experimental results are shown in Section VI. Finally, Section VII concludes the paper, and discusses our future work.

## II. Esterel Syntax and Semantics

We use the kernel language of Pure Esterel [4]–[7]. The kernel language has four unit statements – `nothing`, `pause`, `emit S`, and `exit T` – and seven block-structured constructs – signal test, loop, sequence, parallel, suspension, local signal declaration, and exception declaration. Figure II lists the syntax and the intuitive meanings. The non-terminals $p$ and $q$ denote statements, `S` signals, and `T` exceptions, respectively.

For each input event, Esterel programs react in an instant. Except for the "`pause`" statement, other statements do not change instants. After one instant passes, all signals are reset.

We explain the informal semantics of Esterel syntax: "`nothing`" does nothing. "`pause`" consumes one clock tick. "`emit S`" emits the signal `S`. "$p$;$q$" runs $p$ and $q$ sequentially. "`present S then` $p$ `else` $q$ `end`" tests the presence of the signal `S`, and one of the sub-statements $p$ or $q$ executes according to the test result. "`loop` $p$ `end`" denotes the infinite loop in Esterel. "$p||q$" simultaneously executes $p$ and $q$ with the global clock. The parallel execution terminates when both $p$ and $q$ terminate. "`suspend` $p$ `when S`" is a construct for preemption between threads. When the signal `S` is present, $p$ is suspended until `S` is absent. The scope of a local signal is determined by "`signal S in` $p$ `end`". "`trap T in` p `end`" defines a new exception and its scope, and "`exit T`" in $p$ raises the exception `T`. When the exception `T` arises in $p$, $p$ instantly exits to the end of the corresponding "`trap`" statement.

## III. Schizophrenic Problems

In general, a statement is schizophrenic if its sub-statements are executed more than once in a single instant [12], [14]. In Esterel, schizophrenia can be classified into *schizophrenic signal declarations* and *schizophrenic parallel statements* [5], [13]. In the following we further investigate problems of schizophrenic statements.

### A. Schizophrenic Signal Declarations

When a signal declaration statement is reentered in an instant, two instances of the local signal exist at the same time: an *old instance* and a *new instance* [20]. The main problem of schizophrenic signal declarations is a scoping-rule; a signal test of the new (resp. old) instance must not use the old (resp. new) instance of the signal.

Tardieu and Simone focused on the instantaneous termination of statements at the front and the rear of signal declarations. According to their characterization [13], (a), (c)[3], (d), and (e) in Figure 3 are schizophrenic signal declarations.

As the signal declaration of (b) cannot restart structurally, Tardieu and Simone's static analyzer [18] determines that (b) is not schizophrenic. It states that (f) is schizophrenic due to neglecting signal information on signal `I`, but in fact (f) is not schizophrenic, because all actual paths in the rear of signal declaration in (f) consume an instant.

(a) is a typical example of schizophrenic signal declarations; the old instance of signal `S` in "`emit S`" can meet the new instance of signal `S` in "`present S then emit O end`". There are no signal tests for the new instance of signal

[2]For a signal `S`, "`sustain S`" means a loop statement "`loop emit S; pause end`".

[3]Figure I is similar to (c) in Figure 3

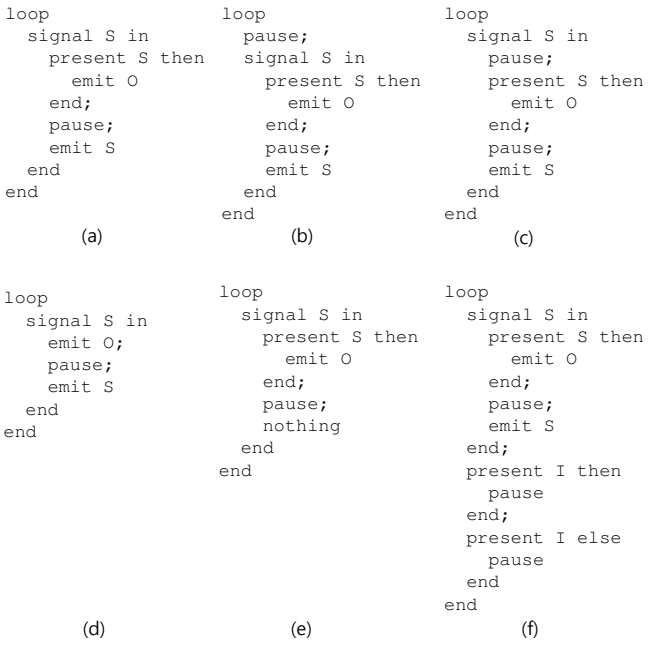


Figure 3. Candidates of Schizophrenic Signal Declarations

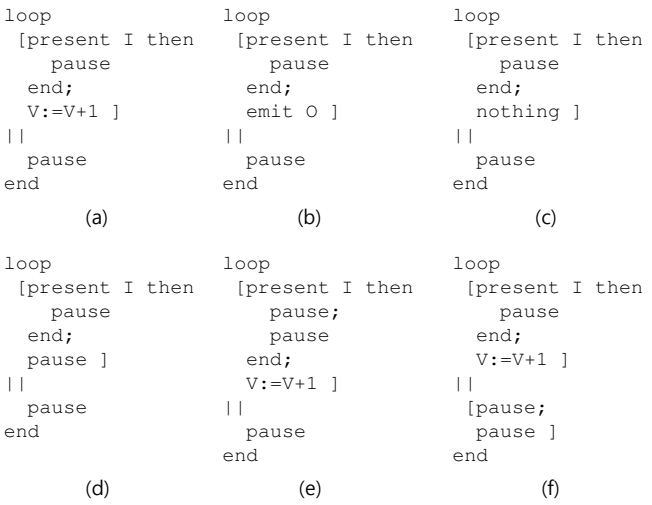


Figure 4. Candidates of Schizophrenic Parallel Statements

S in (c) and (d) during the instant when the signal declaration is restarted, and no signal emission of S's old instance in (e) during the instant when the signal declaration is terminated. Therefore, (a) is the only harmful schizophrenic statement among (a), (c), (d), and (e).

### B. Schizophrenic Parallel Statements

If a statement, such as a variable assignment or valued signal emission, is executed more than once in an instant, the statement's reaction may be unstable. We must consider "what statement" is executed "how many times" in an instant.

When a loop body can instantly terminate, the loop is called an instantaneous loop [18]. The body of an instantaneous loop can be executed infinite times in an instant, and the reaction of the instant does not finish. This erroneous loop statement is rejected by Esterel semantics [5], [21].

A parallel statement in a loop sometimes leads a statement to be executed finitely many times in an instant. Like schizophrenic signal declarations, schizophrenic parallel statements are defined by the reentrance to parallel statements [13]. Each parallel statement in Figure 4 can be reentered during the same instant when its loop terminates and restarts. Therefore, every example in Figure 4 has a schizophrenic parallel statement according to the previous definition.

Code (a) in Figure 4 is an example of a schizophrenic parallel statement, given in [13]. If the input signal I is present at the first instant and absent at the second instant, then "V := V + 1" is executed twice.

"emit O" is executed twice in case of (b), and "nothing" is executed in the case of (c). If the signal O is a pure signal [5], double emissions may cause no problems. Moreover, no sub-statements of the parallel statement in (d) are executed more than once in the same instant.

In (e) and (f), we change the number of instants for each thread's termination. If the input signal I is present at the first instant and absent at the third instant, "V := V + 1" in (e) is executed twice. Case (f) is different from (e). The thread having "V := V + 1" in (f) always terminates before the other thread does, and "V := V + 1" in (f) cannot be executed in the loop-terminating instant due to early termination of the thread. Thus, any sub-statement of (f) cannot be executed more than once in the same instant.

## IV. Refining Schizophrenia

In [13], Tardieu and Simone focused on the reentrance to block statements: signal declarations and parallel statements. However, the reentrance may not cause any problems during compilation of Esterel programs. In short, it is a necessary but not a sufficient condition for schizophrenic problems.

In Pure Esterel, a reaction of a program consists of only signal emissions. Therefore, we refine the characteristics of schizophrenia from the viewpoint of signal emissions in this section. Our refinements can be easily extended to additional features of full Esterel.

### A. Refining Schizophrenic Signal Declarations

*Definition 1:* A signal declaration statement is called a *schizophrenic signal declaration* if

1) the signal declaration terminates or exit, and restarts during the same instant; and
2) a signal test of the local signal's new (resp. old) instance and a signal emission of the local signal's old (resp. new) instance is executed during the same

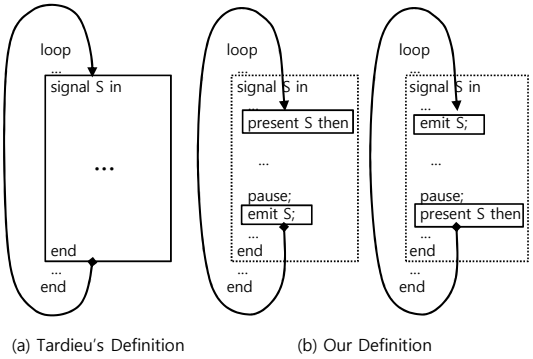


Figure 5. Definitions of schizophrenic signal definitions

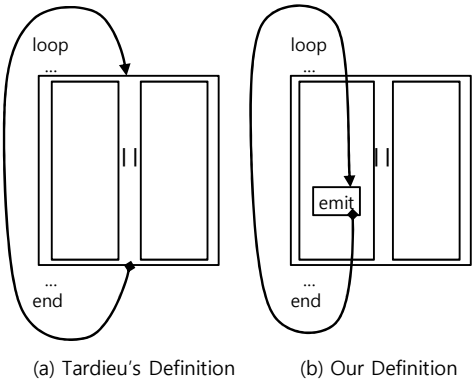


Figure 6. Definitions of schizophrenic parallel statements

instant.

Condition 1) of Definition 1 is the same as Tardieu and Simone's definition [13]. We add condition 2) to describe actual problems noted in Section III.A. Figure 5 illustrates the characteristics of our definition.

The first condition is already a known condition for schizophrenic signal declaration. After investigating the falsely-estimated examples, we add the second condition, which necessarily distinguishes accurate schizophrenic signal declarations.

### B. Refining Schizophrenic Parallel Statements

*Definition 2:* A parallel statement is called a *schizophrenic parallel statement* if a signal emission statement in the parallel statement terminates and restarts during the same instant.

The reentrance to parallel statements is a source of wrong reactions, not a wrong reaction in itself. We investigate whether an `emit` statement is executed more than once in an instant. Parallel statements are necessary conditions for an "`emit`" statement to be executed finitely many times[4].

For full Esterel, we can easily extend the definition by considering additional features: valued signal emission, variables, assignment, and so on. Figure 6 compares the previous definition with ours.

## V. Detecting Schizophrenia on CFGs

Without loss of generality, we assume that all signal names are distinct in an Esterel program. This assumption simplifies our detection algorithm. We implement a unique-naming function and apply it to target programs before the analysis.

### A. Control Flow Graph of Esterel

We briefly introduce our CFG construction of Esterel programs. Our CFG representations for Esterel are presented in Figure 7.

When constructing CFGs of Esterel programs, it is necessary to treat two statements specially: `suspend` statement and `trap` statement. We can translate other statements to CFGs with a simple constructive method.

In `suspend` $p$ `when S`, the signal `S` must be tested at the successors of every pause node in $p$. Therefore, a test($S$) node is inserted at the next place of every pause node in a suspend-block.

In `trap T in p end`, special consideration is necessary for a statement `exit T` in a parallel statement. A parallel statement consists of two threads in Pure Esterel. Suppose a parallel statement has two threads, named P and Q. When thread P of the parallel statement exits, thread Q must exit together. If thread P has an exit($T$) node, the potential exit points of thread Q are the predecessors of pause nodes, exit nodes themselves, and the predecessor of the parallel-end node in thread Q. We insert additional edges, so-called *may-exit edges*, from those nodes to the trap-end node for potential exits. May-exit edges are depicted as dashed lines in Figures 7 and 8. In particular, if thread P has an exit($T$) node and thread Q has an exit($U$) node, then the may-exit edge from the exit($U$) node to the trap-end($T$) node is omitted when trap $U$ is located outside of trap $T$.

To add may-exit edges, we over-approximate for exiting a parallel statement. To exit from parallel statements, we assume that all combinations of potential exit points are possible between two threads in a parallel statement. In Figure 8, our approximation inserts a may-exit edge from the emit($A$) node for the "`exit U`" statement. However, since

[4]The following is a proof idea on kernel statements. Consider an "`emit`" statement that is not contained in any "parallel" statement. After the "`emit`" statement is executed, there must be a "`pause`" statement that cannot be executed at the first time and then must be executed later. This selectively executed "`emit`" statement must be contained in some "`present`" statement. Since the after-parts of the "`emit`" statement are executed several times in an instant and each signal has unique status in an instant, all of the signal tests have to make the same decisions in every execution, and selective execution of a "`pause`" statement is impossible. This is contradiction.

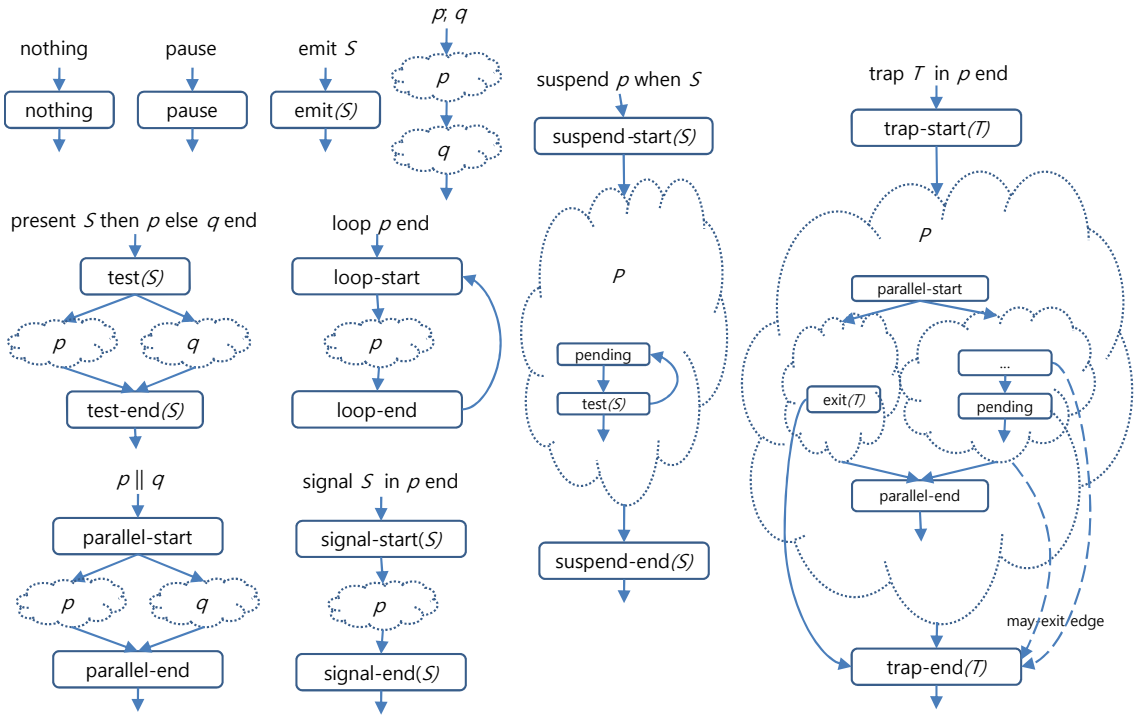


Figure 7. CFG as statements

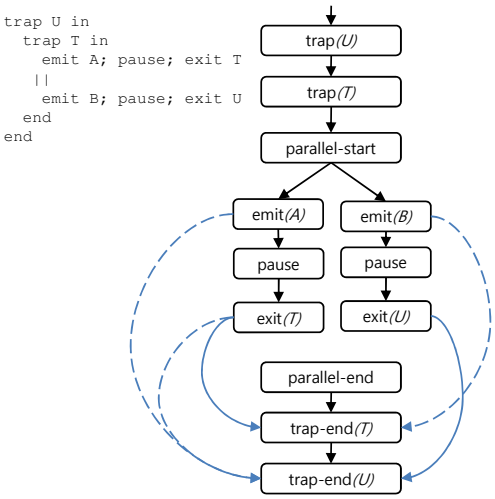


Figure 8. CFG example: exiting parallel statement

"`exit U`" and "`emit A`" cannot be executed simultaneously, the may-exit edge from the emit($A$) node is unnecessary in real environments. The combination of emit($B$) and exit($T$) falls into the same situation.

### B. First-surfaces and Last-surfaces of Loops

Intuitively, the *surface* of a statement means the sub-statements that are executed in the starting instant of the statement. This term is widely used in [5], [12]–[14], [16].

When a loop statement terminates and restarts in an instant, the surface of the loop statement is the only candidate that can be re-executed in the same instant, and schizophrenic problems can occur in it. Based on this observation, some curing techniques [12]–[16] replicate the surface of loop statements.

In this paper, we name the surface of a loop statement the *first-surface* of the loop statement in order to distinguish the sub-statements that are executed in the terminating instant of the loop statement, which are named the *last-surface* of the loop statement. We define the first-surfaces and the last-surfaces of loop statements on CFGs.

*Definition 3:* The *first-surface* of a loop statement is the set of nodes in the CFG that are reachable in the instant when the loop statement starts.

The first-surface of a loop refers to the set of nodes that the loop-start node can reach without passing pause nodes. Figure 10 is an example of a first-surface.

Figure 9 is an algorithm to compute the first-surface of a loop statement. For parallel-statements, lines 15∼19 reflect that a parallel statement cannot terminate until two threads terminate. To exit a trap statement in a loop starting instant, there must exist an exit node that is instantly reachable from the loop-start node. Hence we do not follow may-exit edges at line 27 in Figure 9.

**Require:** $loop$ is the set of nodes in the target loop's CFG. loop_start_node is the target loop's starting node.

```
1: // initialization
2: for all node ∈ loop do
3:   if node is a parallel_end node then
4:     // a parallel statement consists of two threads
5:     node.thread_num := 2;
6:   end if
7: end for
8: // start to compute first_surface
9: worklist := {loop_start_node};
10: first_surface := ∅;
11: while worklist ≠ ∅ do
12:   node := pop(worklist);
13:   if node is not a pause node then
14:     add_worklist := false;
15:     if node is a parallel_end node then
16:       node.thread_num := node.thread_num-1;
17:       if node.thread_num = 0 then
18:         first_surface := first_surface ∪ {node};
19:         add_worklist := true
20:       end if
21:     else
22:       first_surface := first_surface ∪ {node};
23:       add_worklist := true
24:     end if
25:     if add_worklist = true then
26:       for all edge: (node→target) ∧ target∈ loop do
27:         if target ∉ first_surface
            ∧ edge is not a may-exit edge then
28:           worklist := worklist ∪ {target};
29:         end if
30:       end for
31:     end if
32:   end if
33: end while
34: return first_surface
```

Figure 9. The algorithm to compute first-surface

*Definition 4:* The *last-surface* of a loop statement is the set of nodes in the CFG that are reachable in the instant when the loop statement terminates.

The last-surface is the opposite case of the first-surface. The last-surface and the first-surface of a loop statement are executed simultaneously when the loop is terminates and restarts.

We ignore the case where target loop statements are instantaneous loops[5]. Therefore, a loop terminating instant must begin from some pause nodes inside the loop.

[5]We can detect potential instantaneous loops using first-surfaces. If the first-surface of a loop statement includes the loop-end node, the loop statement may be an instantaneous loop. This detection technique ignores the signal information as the previous technique [18] does.

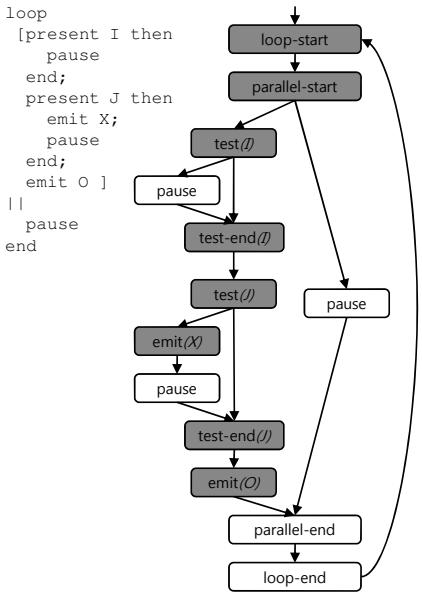


Figure 10. First-surface: an example

Figure 11 is an algorithm to compute the last-surface of a loop statement. The algorithm is comprised of two phases. In the first phase, the algorithm identifies pause nodes that are instantly reachable to the loop-end node, and marks the pass-through nodes while backward-searching the pause nodes from the loop-end node. This process is a backward version of the algorithm to compute first-surfaces. In the second phase, instantly reachable nodes from the selected pause nodes are chosen from among the nodes marked during the first phase.

Figure 12 presents an example. In the first phase, our algorithm selects all pause nodes in the program, and marks emit($O$), test-end($J$), test($J$), test-end($I$), and test($I$) nodes. Even though there is a forward-path from a selected pause node to the emit($X$) node, we can exclude the emit($X$) node from the last-surface, because the emit($X$) node is not marked in the first phase. In addition, because the test($I$) node, which is marked in the first phase, is not instantly reachable from any selected pause node, our algorithm does not add the test($I$) node to the last-surface.

### C. Detecting Schizophrenic Signal Declarations

**Schizophrenic Signal Declaration detection procedure**

1) Construct the CFG of a `loop` statement.
2) Compute the first-surface and the last-surface of the loop statement on CFG.
3) If a) the first-surface has a test($S$) node and the signal-start($S$) node of the local signal $S$ and b) the last-surface has an emit($S$) node, then *the signal declaration of the local signal $S$ is a schizophrenic signal declaration*.

**Require:** $loop$ is the set of nodes in the target loop's CFG.
loop_end_node is the target loop's ending node.

1: // initialization
2: **for all** node $\in$ $loop$ **do**
3: node.visited := **false**
4: **if** node is a $parallel\_start$ node **then**
5: node.thread_num := 2;
6: **end if**
7: **end for**
8: // first phase: gather loop-terminating pause nodes
9: worklist := {loop_end_node};
10: pause_set := $\emptyset$;
11: **while** worklist $\neq \emptyset$ **do**
12: node := pop(worklist);
13: add_worklist := **false**;
14: **if** node is a *pause* node **then**
15: pause_set := pause_set $\cup$ {node};
16: **else if** node is a *parallel_start* node **then**
17: node.thread_num := node.thread_num - 1;
18: **if** node.thread_num = 0 **then**
19: add_worklist := **true**;
20: **end if**
21: **else**
22: add_worklist := **true**;
23: **end if**
24: **if** add_worklist = **true then**
25: node.visited := **true**;
26: **for all** edge: (source$\rightarrow$node) $\wedge$ source$\in$ $loop$ **do**
27: **if** source.visited = **false then**
28: worklist := worklist $\cup$ {source};
29: **end if**
30: **end for**
31: **end if**
32: **end while**
33: // second phase: compute last-surface
34: worklist := pause_set;
35: last_surface := $\emptyset$;
36: **while** worklist $\neq \emptyset$ **do**
37: node := pop(worklist);
38: **if** node.visited = **true then**
39: last_surface := last_surface $\cup$ {node};
40: **end if**
41: **if** node is a *pause* node $\vee$ node.visited = **true then**
42: **for all** edge: (node$\rightarrow$target) $\wedge$ target $\in$ $loop$ **do**
43: **if** target $\notin$ last_surface
$\wedge$ target.visited = **true then**
44: worklist := worklist $\cup$ {target};
45: **end if**
46: **end for**
47: **end if**
48: **end while**
49: **return** last_surface

Figure 11. The algorithm to compute last-surface

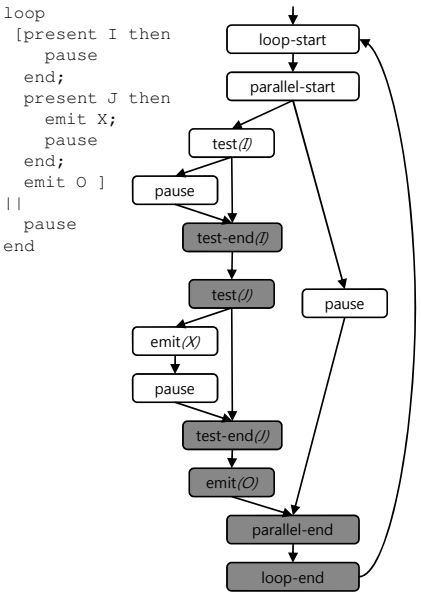


Figure 12. Last-surface: an example

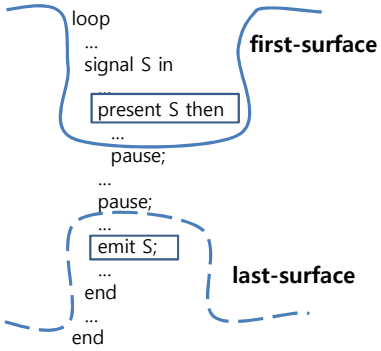


Figure 13. Detecting Schizophrenic Signal Declarations

With the refined schizophrenic signal declaration, we detect local signal emissions of the old instance that affect the local signal tests of the new instance. Figure 13 depicts a case of schizophrenia with our detection procedure.

Suppose a signal $S$ is declared. If the signal-start($S$) node is in the first-surface, we know that the signal declaration of signal $S$ is in the loop. In case of exiting the signal declaration, the signal-end($S$) node may not be in the last-surface. However, from our unique-naming assumption, we know that $S$ means the local signal without having to check where the name $S$ is.

According to the definitions of the first-surface and the last-surface, the nodes of the last-surface may be executed in the loop terminating instant and the nodes of the first-surface may be executed in the loop starting instant. When the target loop statement terminates and restarts during the same instant, an emit($S$) node in the last-surface is the emission of the signal $S$'s old instance and a test($S$) node

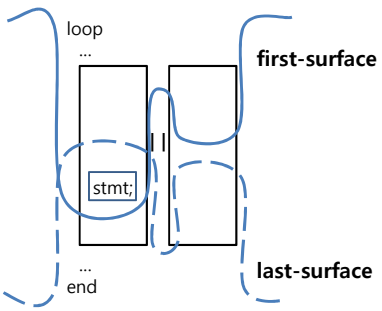


Figure 14. Detecting Schizophrenic Parallel Statements

in the first-surface is the test of the signal $S$'s new instance. Because nodes of the last-surface and nodes of the first-surface are executed in the same instant, the emit($S$) node can affect the test($S$) node. Therefore, if both conditions a) and b) are true, we state that the signal declaration of signal $S$ may be a schizophrenic signal declaration.

### D. Detecting Schizophrenic Parallel Statements

**Schizophrenic Parallel Statement detection procedure**

1) Construct the CFG of a `loop` statement.
2) Compute the first-surface and the last-surface of the loop statement on CFG.
3) If a) the loop-end node is not in the first-surface and b) a signal emission node is in the intersection of the first-surface and the last-surface, then *the parallel statement may be a schizophrenic parallel statement*.

In accordance with our refined schizophrenic parallel statement, our detection criterion is whether a signal emission statement is executed more than once in an instant. Figure 14 shows our detection algorithm of schizophrenic parallel statements. For Pure Esterel, we only check signal emission nodes.

We can detect instantaneous loops by examining whether the first-surface contains the loop-end node. Condition a) of step 3) checks if a target loop is an instantaneous loop. If a loop violates condition a), it is not cured, but rather rejected [22].

Condition b) of step 3) reflects the refined characteristics of schizophrenic parallel statements. Suppose a node is contained in the intersection of the first-surface and the last-surface. Because the node is in the last-surface, the node may be executed in a loop-terminating instant. Because the node is in the first-surface, the node may be executed again when the loop restarts during the same instant.

If both conditions a) and b) of step 3) are true, the target loop may not be an instantaneous loop, and a signal emission node can be executed more than once in an instant. Therefore, we can detect schizophrenic parallel statements.

## VI. Experimental results

We implemented the proposed detection algorithm based on first-surfaces and last-surfaces in OCaml. Our detector works as follows:

1) inlining of submodules using Columbia Esterel Compiler [23], [24],
2) parsing and translating additional language features to kernel language referring to [7], and
3) constructing CFG of a target program and analyzing each loop.

| Figure 3 | Reentrance | Our result | Manual check |
|---|---|---|---|
| (a) | yes | yes | yes |
| (b) | no | no | no |
| (c) | yes | no | no |
| (d) | yes | no | no |
| (e) | yes | no | no |
| (f) | yea | yes | no |

Table I
Examples: Schizophrenic signal declarations

| Figure 4 | Reentrance | Our result | Manual check |
|---|---|---|---|
| (a) | yes | yes | yes |
| (b) | yes | no | no |
| (c) | yes | no | no |
| (d) | yes | no | no |
| (e) | yes | yes | yes |
| (f) | yes | yes | no |

Table II
Examples: Schizophrenic parallel statements

For Tables I, II, and III, the column "Reentrance" states whether target block statements can be reentered[6]. The column "Our result" indicates whether our detector says it is a schizophrenic statement. The column "Manual check" shows whether corresponding examples cause actual *problems* as stated in Section IV. To obtain precise results, we counted the exact numbers of "Manual check" by hand.

In Figure 3 of Section III.A, there are some candidate schizophrenic signals. Table I compares the results. Because we do not consider signal information, similar to Tardieu and Simone's approach [18], our algorithms for first-surfaces and last-surfaces cannot analyze `present` statements of (f) in detail.

Table II presents the analysis results for the examples in Figure 4. Our algorithm computes last-surfaces based-on backward reachability on CFGs. It assumes that both threads of a parallel statement may terminate simultaneously. That is, the V:=V+1 node is contained in the last-surface by our algorithm. However, the first thread of (f) in Figure 4 always terminates before the second thread does. Thus, the V:=V+1 statement cannot be executed when the loop statement terminates.

[6]We check whether a target block statement can be reentered in an instant using the first-surface and the last-surface. If the start node of the target block statement is in the first-surface of a loop and any node of the block statement is in the last-surface of the loop, the target block statement can terminate or exit and restart during the same instant. This analysis is similar to that of Tardieu and Simone [18].

| Programs | Lines of code | Num. of loops | Schizophrenic Signal Declarations | | | | Schizophrenic Parallel Statements | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Num. of Candidates | Reentrance | **Our result** | Manual check | Num. of Candidates | Reentrance | **Our result** | Manual check |
| atds100 | 622 | 55 | 16 | 9 | **5** | 0 | 18 | 7 | **5** | 5 |
| mca200 | 5,354 | 138 | 0 | 0 | **0** | 0 | 0 | 0 | **0** | 0 |
| mejia | 361 | 21 | 0 | 0 | **0** | 0 | 5 | 0 | **0** | 0 |
| tcint | 357 | 32 | 1 | 1 | **0** | 0 | 2 | 2 | **1** | 1 |
| ww | 360 | 83 | 1 | 1 | **1** | 1 | 6 | 1 | **1** | 1 |
| dlx | 334 | 37 | 5 | 5 | **0** | 0 | 5 | 5 | **0** | 0 |
| fbus | 285 | 76 | 0 | 0 | **0** | 0 | 3 | 0 | **0** | 0 |
| **Total** | 7,673 | 442 | 23 | 16 | **6** | 1 | 39 | 15 | **7** | 7 |

Table III
OUR EXPERIMENTS

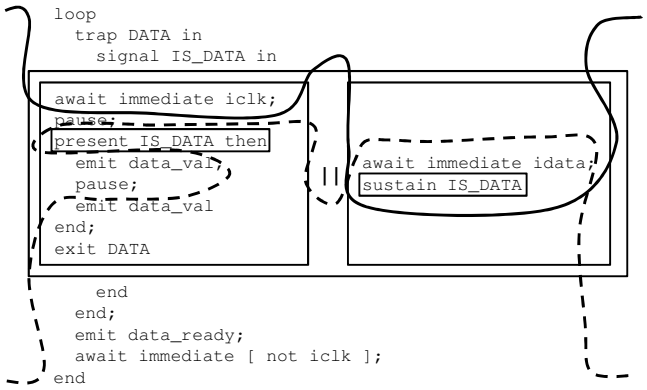


Figure 15. Our false alarms in atds100

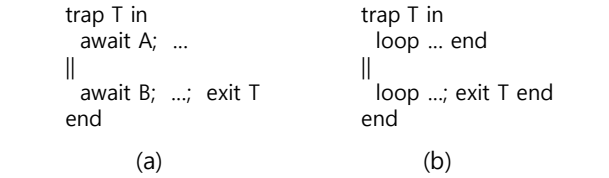


Figure 16. Coding Patterns for exiting parallel statements

Table III shows the results for benchmark programs[7]. We count the lines of submodule-inlined codes. The column "Num. of loops" is the number of `loop` statements and `repeat` statements [7]. The column "Num. of Candidates" of schizophrenic signal declarations (or schizophrenic parallel statements) is the number of loops that have signal declarations (or parallel statements).

In spite of naive CFGs, our detection algorithm performs well in analyzing the benchmark programs. The only false alarms happen in `atds100`: the code of Figure I. Figure 15 shows that "`present IS_DATA then`" is in the last-surface and "`sustain IS_DATA`" is in the first-surface. However, they are not executed simultaneously in real environment.

After examining the benchmark codes manually, we can identify two programming patterns for exiting parallel statements depicted in Figure 16. These patterns are natural in designing hardware. Each thread works after obtaining some information, like (a) of Figure 16, or performs its own work repeatedly, like (b). Because these coding patterns can make all pending combination of two threads possible, they reduce the inaccuracy of computed last-surfaces caused by may-exit edges.

[7]The programs of `atds100`, `mca200`, `mejia`, `tcint`, and `ww` are in the Estbench Esterel benchmark suite [17]. The programs of `dlx` and `fbus` are Ramesh's case-studies [25].

## VII. CONCLUSION

Imperative constructs in a synchronous language often cause schizophrenic problems. Esterel compilers must cure schizophrenic problems.

The existing schizophrenia detection algorithm [18] uses over-approximation based on abstract interpretation. Although the result decreases the number of statements to be cured dramatically, some harmless statements are regarded as schizophrenic. In practice, the inaccuracy is due to the roughly inferred characteristics of schizophrenic problems. They do not have sufficient information to distinguish harmless statements from harmful schizophrenia.

In this paper, we refine the characteristics of schizophrenic problems in terms of signal emissions in Pure Esterel, and develop a schizophrenia detection algorithm on CFGs of Esterel programs. Our key contributions are 1) observations about schizophrenic problems and 2) a detection algorithm using graph reachability. Our detector can distinguish most harmless statements from schizophrenic statements in practical programs in spite of naively-designed CFGs. As our algorithm utilizes simple graph reachability on CFGs, it can be easily applied to existing compilers.

Compilers generate some control signals [5] on circuit synthesis, and sharing the control signals [26] may cause other problems, especially in the synthesis of *parallel* statements [5]. We will extend our work to schizophrenia of synthesized circuits.

## ACKNOWLEDGMENT

This research was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) Support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2009-C1090-0902-0020) and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-02004-0.

## REFERENCES

[1] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht, 1993.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Embedded Systems, Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[3] A. Benvenistre and G. Berry, "The synchronous approach to reactive real-time systems," *Another Look of Real Time Programming, Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[4] G. Berry, *The Esterel Primer*, 1998.

[5] ——, *The Constructive Semantics of Pure ESTEREL*. Draft book available at http://www.inria.fr/meije/esterel/esterel-eng.html, 1999.

[6] ——, "The foundations of esterel," *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 425–454, 2000.

[7] D. Potop-Butucaru, S. Edwards, and G. Berry, *Compiling ESTEREL*. Springer, 2007.

[8] Esterel-Technologies, *The Esterel v7 Reference Manual Version v7.30 . initial IEEE standardization proposal*, Esterel-Technologies, 679 av. Dr. J. Lefebvre 06270 Villeneuve-Loubet, France, November 2005.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[10] P. L. Guernic, T. Goutier, M. L. Borgne, and C. Maire, "Programming real time applications with signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[11] G. Berry, "Preemption in concurrent systems," in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1993, pp. 72–93.

[12] K. Schneider and M. Wenz, "A new method for compiling schizophrenic synchronous programs," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems(CASES'2001)*. New York, NY, USA: ACM, 2001, pp. 49–58.

[13] O. Tardieu and R. de Simone, "Curing schizophrenia by program rewriting in esterel," in *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Codesign(MEMOCODE'2004)*, 2004, pp. 39–48.

[14] K. Schneider, J. Brandt, and T. Schuele, "A verified compiler for synchronous programs with local declarations," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 153, no. 4, pp. 71–97, 2006.

[15] A. Poigne and L. Holenderski, "Boolean automata for implementing pure esterel," *Arbeitspapiere der GMD 964, GMD, Sankt Augustin*, 1995.

[16] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, "Saxo-rt: Interpreting esterel semantic on a sequential execution structure," vol. 65, no. 5. Elsevier, 2002, pp. 80–94.

[17] S. Edwards, "Estbench esterel benchmark suite," http://www1.cs.columbia.edu/∼sedwards/software/estbench-1.0.tar.gz.

[18] O. Tardieu and R. de Simone, "Instantaneous termination in pure esterel," in *Proceedings of the 10th International Static Analysis Symposium(SAS'2003)*, ser. LNCS, vol. 2694. Springer Verlag, 2003, pp. 91–108.

[19] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *The 4th ACM Symposium on Principles of Programming Languages*. Los Angeles, CA.: ACM Press, 1977, pp. 238–252.

[20] O. Tardieu, "Loops in esterel: from operational semantics to formally specified compilers," *These de doctorat, Ecole des Mines de Paris*, 2004.

[21] ——, "A deterministic logical semantics for pure esterel," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 2, pp. 1–24, 2007.

[22] O. Tardieu and R. de Simone, "Loops in esterel," *Transactions on Embedded Computing Systems*, vol. 4, no. 4, pp. 708–750, 2005.

[23] S. Edwards, "Cec: The columbia esterel compiler," http://www1.cs.columbia.edu/∼sedwards/cec/.

[24] S. Edwards and J. Zeng, "Code generation in the columbia esterel compiler," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 1–31, 2007.

[25] S. Ramesh, "Ramesh's homepage," http://www.cse.iitb.ac.in/∼ramesh/.

[26] A. Su, Y. Hsu, T. Liu, and M. Lee, "Eliminating false loops caused by sharing in control path," *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, no. 3, pp. 487–495, 1998.