PAPER Special Section on Formal Approach

Over-Approximated Control Flow Graph Construction on Pure Esterel*

Chul-Joo KIM[†a)], **Jeong-Han YUN**[†b)], ***Student Members***, **Seonggun KIM**[†c)], **Kwang-Moo CHOE**[†d)], ***and*** **Taisook HAN**[†e)], ***Nonmembers***

SUMMARY Esterel is an imperative synchronous language for control-dominant reactive systems. Regardless of imperative features of Esterel, combination of parallel execution and preemption makes it difficult to build control flow graphs (CFGs) of Esterel programs. Simple and convenient CFGs can help to analyze Esterel programs. However, previous researches are not suitable for flow analyses of imperative languages. In this work, we present a method to construct over-approximated CFGs for Pure Esterel. Generated CFGs expose invisible interferences among threads and show program structures explicitly so that they are useful for program analyses based on graph theory or control-/data- flows.

key words: *esterel, control flow graph, synchronous language*

1. Introduction

Most embedded systems belong to reactive systems; they check and react to environmental changes as time flows. It is very difficult to synchronize an embedded system and environment. Furthermore, as system requirements become more complicated, an embedded system consists of more sub-components. General programming languages are not proper to express the synchronization among sub-components explicitly.

Synchronous languages [1], [2] with the perfect synchrony hypothesis [3] help to specify synchronization by abstracting time flow into a sequence of discrete time unit. Esterel [4]–[7] is an imperative synchronous language. Unlike dataflow synchronous languages [1], [2], [8], [9], Esterel supports various imperative features. These include sequence, loop, suspension, preemption [10], exception, and declaration of signals and variables, and they are useful to design control-dominant systems.

Control-/data- flow analysis [11] has been a static analysis tool in compilers of general imperative languages for decades. Because an Esterel program is written with imperative constructs, control flows exist within the program, and a control flow graph (CFG) can be built for the program. Using the CFG, the program can be diagnosed by various analysis techniques such as graph reachability or flow analyses.

However, a CFG of an Esterel program cannot be easily constructed due to synchronous parallel execution and preemption. When two threads of a program are executed in parallel, one thread can be interfered by the other, and vice versa. Unfortunately, such interferences are not explicitly described in the source code.

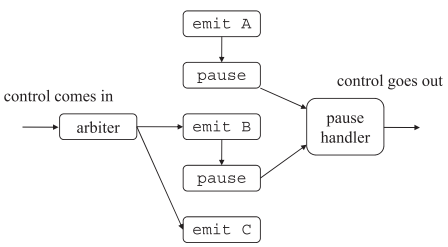
1.1 Related Works

Flow information is a basis of most program analyses, and some analyzers [12]–[14] have their own structures for flow information.

An Esterel slicing tool [12] generates CFGs to construct program dependency graphs, and Fig. 1.(b) is a CFG for a program depicted in Fig. 1.(a). It deals with control flows of parallel execution and preemption with *pause handlers* and *arbiters*. Pause handlers gather state information within an instant, and arbiters decide control flows of the next instant using the information. Though they exactly represent actual execution process, all following nodes of a `pause` node in this CFG are dependent to arbiters, and an-

```
1  emit A;
2  pause;
3  emit B;
4  pause;
5  emit C
```

(a)



(b)

Fig. 1 Control flow graphs by [12].

Manuscript received July 17, 2009.
Manuscript revised October 28, 2009.

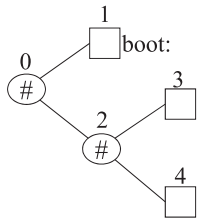
†The authors are with Division of Computer Science, KAIST, Daejeon, Korea.

*This work was supported by 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 and the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) Support program supervised by the NIPA (National IT industry Promotion Agency) (NIPA-2010-C1090-1031-0004).

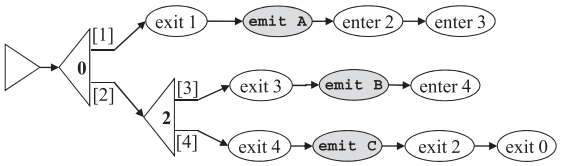
a) E-mail: chuljoo.kim@gmail.com
b) E-mail: jeonghan.yun@gmail.com
c) E-mail: seonggun.kim@arcs.kaist.ac.kr
d) E-mail: choe@kaist.ac.kr
e) E-mail: han@cs.kaist.ac.kr

DOI: 10.1587/transinf.E93.D.985

Copyright © 2010 The Institute of Electronics, Information and Communication Engineers



(a) A Concurrent CFG for Fig. 1.(a)



(b) A Hierarchical State Representation for Fig. 1.(a)

Fig. 2 Control flow graphs by [13], [15].

alyzers can hardly distinguish the gathered information at pause handlers. For instance, in Fig. 1.(a), `pause` in line 4 is sequentially following `pause` in line 2, but in Fig. 1.(b), the upper `pause` node is connected to the pause handler and the lower `pause` node succeeds the arbiter. Therefore, analyzers must exactly simulate pause handlers and arbiters to get precise control flows and analysis results.

Columbia Esterel compiler [13], [15] translates Esterel programs to GRC models [16]. Figure 2 shows a GRC model of the program in Fig. 1.(a). A GRC model consists of a concurrent CFG and a hierarchical state representation which are optimized for simulation and compilation. A hierarchical state indicates a structured data memory that preserves state information across instants, and a concurrent CFG represents the computation of an instant. Therefore, analyzers must examine source program and the model simultaneously and interpret the control flows on the model.

Quartz [14], a dialect of Esterel, is transformed to not CFGs but equation systems [17]. Compiled equations consist of control and data flows. Because the equations are designed for theorem provers, they need additional transformations for flow analyses [11].

1.2 Our Approach

To apply flow analyses for imperative languages to Esterel, we develop syntax-directed rules to construct over-approximated [†] CFGs of Esterel programs. Based on CFG construction methods for imperative languages, we extend to deal with parallelism and preemption of Esterel.

Our over-approximation is necessary to generate simple and practical CFGs for the combination of parallelism and preemption. Generated CFGs by our rules include all the possible control flows and show imperative features intuitively. However, our over-approximation may leave infeasible execution paths in generated CFGs. But, due to programming patterns of Esterel programs, infeasible paths are rare in practical programs.

Our CFGs can be used for various flow analyses. Unreachable nodes indicate dead codes [18] because our CFGs include all the possible control flows. Instantaneous paths can be detected by graph reachability, so that we can detect instantaneous loops [4], [19] and schizophrenic problems [19]. In addition, because our CFGs explicitly expose all control flows, data flow analysis for synchronous programs [20] can be applied to Esterel programs.

Statement p, q

::=	`nothing`	no operation
|	`pause`	consuming a clock tick
|	`emit s`	signal emission
|	`exit t`	exception raise
|	p `;` q	sequence of statements
|	p `||` q	parallel execution
|	`present s then` p `else` q `end`	signal test
|	`loop` p `end`	nonterminating loop
|	`signal s in` p `end`	local signal
|	`suspend` p `when s`	suspending program
|	`trap t in` p `end`	exception declaration

Fig. 3 Pure Esterel.

Our paper is organized as follows. In Sect. 2 we briefly introduce Esterel. In Sect. 3 we describe how to construct a CFG of Esterel program with syntax-directed rules. Experimental results are shown in Sect. 4. Finally, Sect. 5 concludes the paper and discusses future work.

2. Pure Esterel

2.1 Esterel Syntax and Semantics

We use the kernel language of Pure Esterel [4]–[6], [21]. 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 3 lists the syntax and the intuitive meanings. The nonterminals p and q denote statements, `s` signals, and `t` exceptions, respectively. Signals and exceptions are identifiers lexically scoped and declared within statements by `signal` and `trap` statements, respectively.

For each input event, Esterel programs instantly react. Except for the `pause` statement, other statements do not consume time. The explicit synchronization control using `pause` statements is an important feature of Esterel. An Esterel program pends at each `pause` statement, and the synchronization occurs within this unit. We call this unit of program computation an *instant*. An instant is a consistent sequence of zero-delay actions for one input event. After one instant passes, all signals are reset.

We explain the informal semantics of Esterel syntax. `nothing` does nothing. `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`

[†] An over-approximated CFG, though it may contain some unreachable paths in run-time, must represent all possible execution paths of an Esterel program, including every implicit control flows caused by the parallel execution and preemption.

denotes the infinite loop in Esterel. In order to exit from the loop, one must use exception constructs. $p||q$ 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. Note that the signal `s` can never be emitted by the process p itself, and tests for presence of `s` are allowed in p. The scope of a local signal is determined by `signal s in` p `end`. The signal `s` outside the scope is regarded as different from the local signal. It is not technically difficult to distinguish the signal names inside and outside of the local signal scopes, and hence we assume that all signal names are distinct in an Esterel program. `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.

2.2 Preemption

Control flows of the preemption statements, `suspend` and `exit`, are not explicitly exposed in source codes. There are two kinds of preemption in Esterel.

Strong preemption: halts the remaining task immediately and performs the preempted task when the preemption condition takes place. For example, in the case of `suspend` p `when s`, if signal `s` is present, it immediately holds p and waits until `s` becomes absent.

Weak preemption: when the preemption condition takes place, a program finishes the remaining task during the current instant and then performs the preempted task. For example, in the case of $p||q$, even though p has already been ended by `exit t` in an instant, q cannot recognize it before the synchronization. After performing all the tasks defined in the instant, q recognizes the end of p and also finishes itself regardless of the existence of the following tasks.

3. Control Flow Graph Construction

In order to simplify a CFG construction process, we assume that every identifer in a program is globally unique. This assumption can be guaranteed by a preprocessor that renames identifiers with the same name into unique ones, while abiding the scoping rules of Esterel. In addition, we also assume that there is no `exit` statement whose target exception is not in its valid scope. If any `exit t` targets an invalid exception `t`, the preprocessor substitutes the `exit` statement with `nothing` statement, which has the same behavioral semantics.

3.1 Definitions

Figure 4 shows the definitions we used in this study. We represent a CFG as a 5-tuple, $\langle s, f, N, E, W\rangle$, where s and f are the indices of the start and finish nodes, respectively,

$$\textit{Control Flow Graph} ::= \langle\ s,\ f,\ N,\ E,\ W\rangle$$

$$\begin{pmatrix} \boldsymbol{s}: & \text{index of start } \textit{node} \\ f: & \text{index of finish } \textit{node} \\ N: & \text{a set of } \textit{node} \\ E: & \text{a set of } \textit{edge} \\ W: & \text{a set of } (\texttt{t},\ i,\ [\curvearrowright|\rightsquigarrow]) \end{pmatrix}$$

node	::=	$\overset{i}{\textit{type}}$	
type	::=	`nothing` | `pause` | `emit s` | `exit t`	
	|	`B`	(entry node)
	|	`E`	(exit node)
edge	::=	$i \xrightarrow{l} j$	(normal edge)
	|	$i \twoheadrightarrow j$	(parallel edge)
	|	$i \curvearrowright j$	(exit edge)
	|	$i \rightsquigarrow j$	(may-exit edge)
$i,\ j$	::=	unique index of *node*	
		(generated when new *node* is introduced)	
l	::=	ϵ	(unconditional)
	|	`s`	(presence of signal)
	|	$\neg$`s`	(absence of signal)
	|	$l \wedge l$	(conjunction of signal)
	|	$l \vee l$	(disjunction of signal)
`s`	::=	signal identifier	
`t`	::=	trap identifier	

Fig. 4 Control flow graph.

N is a set of nodes, E is a set of edges, and W is a set of incomplete edges.

Each node of a CFG is either entry (`B`), exit (`E`), or one of the four unit statements and identified by an integer index i. The integer index is generated when a new node is introduced. A transfer of control flow is represented by a directed edge that leaves a source node i and arrives at a destination node j.

We categorize the control flow transfers into four types (*normal, parallel, exit, may-exit*) according to their implications on program execution. Parallel edges leaving or targeting a node represent that the control flow forks or joins at the node, respectively. Control flow transfers due to exceptions are represented by exit or may-exit edges, where may-exit edges indicate that the transfer is due to the exception raised by other threads. The rest are called normal edges.

Since a control flow transfer may be guarded by presence of a signal, we annotate each edge with a condition l that is either unconditional (ϵ), the presence of a signal (`s`), the absence of a signal ($\neg$`s`), the conjunction of signal statuses ($l \wedge l$), or the disjunction of signal statuses ($l \vee l$). Note that for simplicity we omit the annotation if it is unconditional in the rest of this paper.

While constructing a graph, the target node of an exit or may-exit edge remains unknown until the corresponding exception declaration statement is processed. Until then, we keep its trap identifer (`t`), index of the source node (i), and edge type in W. The destination nodes of edges in the set W are all resolved when the construction process finished. The

$$(nothing) \quad \texttt{nothing} \triangleright \langle\, i,\ i,\ \{\overset{i}{\texttt{nothing}}\},\ \emptyset,\ \emptyset \rangle$$

$$(pause) \quad \texttt{pause} \triangleright \langle\, i,\ i,\ \{\overset{i}{\texttt{pause}}\},\ \emptyset,\ \emptyset \rangle$$

$$(emit) \quad \texttt{emit s} \triangleright \langle\, i,\ i,\ \{\overset{i}{\texttt{emit s}}\},\ \emptyset,\ \emptyset \rangle$$

$$(exit) \quad \texttt{exit t} \triangleright \langle\, s,\ f,\ \{\overset{s}{\texttt{exit t}},\ \overset{f}{\texttt{E}}\},\ \emptyset,\ \{(\texttt{t}, s, \curvearrowright)\} \rangle$$

$$(sequence) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle \quad q \triangleright \langle\, s_q,\ f_q,\ N_q,\ E_q,\ W_q \rangle}{p\,;q \triangleright \langle\, s_p,\ f_q,\ N_p \cup N_q,\ E_p \cup E_q \cup \{f_p \to s_q\},\ W_p \cup W_q \rangle}$$

$$(loop) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle}{\texttt{loop}\ p\ \texttt{end} \triangleright \langle\, s,\ f,\ N_p \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\},\ E_p \cup \{s \to s_p,\ f_p \to s_p\},\ W_p \rangle}$$

$$(present) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle \quad q \triangleright \langle\, s_q,\ f_q,\ N_q,\ E_q,\ W_q \rangle}{\begin{array}{l}\texttt{present s then}\ p\ \texttt{else}\ q\ \texttt{end} \triangleright \langle\, s,\ f,\ N_p \cup N_q \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\} \\ \qquad ,\ E_p \cup E_q \cup \{s \xrightarrow{\texttt{s}} s_p,\ s \xrightarrow{\neg\texttt{s}} s_q,\ f_p \to f,\ f_q \to f\},\ W_p \cup W_q \rangle\end{array}}$$

$$(signal) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle}{\texttt{signal s in}\ p\ \texttt{end} \triangleright \langle\, s,\ f,\ N_p \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\},\ E_p \cup \{s \to s_p,\ f_p \to f\},\ W_p \rangle}$$

$$(parallel) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle \quad q \triangleright \langle\, s_q,\ f_q,\ N_q,\ E_q,\ W_q \rangle}{\begin{array}{l}p\,||\,q \triangleright \langle\, s,\ f,\ N_p \cup N_q \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\},\ E_p \cup E_q \cup \{s \twoheadrightarrow s_p,\ s \twoheadrightarrow s_q,\ f_p \twoheadrightarrow f,\ f_q \twoheadrightarrow f\} \\ \qquad ,\ add\_may(W_p, N_q, E_q, f_q) \cup add\_may(W_q, N_p, E_p, f_p) \rangle\end{array}}$$

where, $add\_may(W, N', E', f') = W \cup \bigcup_{(\texttt{t},i,-) \in W} \{(\texttt{t}, j, \rightsquigarrow) \mid (j \to f' \in E') \vee (j \to k \in E' \wedge \overset{k}{\texttt{pause}} \in N') \vee (j \to k \in E' \wedge \texttt{exit t}' \wedge \texttt{t} \text{ is outer than } \texttt{t}')\}$

$$(suspend) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle}{\texttt{suspend}\ p\ \texttt{when s} \triangleright \langle\, s,\ f,\ N_p \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\},\ adjust\_edge(\texttt{s}, N_p, E_p) \cup \{s \to s_p,\ f_p \to f\},\ W_p \rangle}$$

where, $adjust\_edge(\texttt{s}, N, E) = \forall \overset{i}{\texttt{pause}} \in N.\ \ \forall (i \to j) \in E.\ (E/\{i \to j\}) \cup \{i \xrightarrow{\neg\texttt{s}} j,\ i \xrightarrow{\texttt{s}} i\}\ (i \neq j) \cup \forall (i \xrightarrow{l} j) \in E.\ (E[(i \xrightarrow{l} j) \mapsto (i \xrightarrow{l \wedge \neg\texttt{s}} j)])\ (i \neq j) \cup \forall (i \xrightarrow{l} i) \in E.\ (E[(i \xrightarrow{l} i) \mapsto (i \xrightarrow{l \vee \texttt{s}} i)])$

$$(trap) \quad \frac{p \triangleright \langle\, s_p,\ f_p,\ N_p,\ E_p,\ W_p \rangle}{\begin{array}{l}\texttt{trap t in}\ p\ \texttt{end} \triangleright \langle\, s,\ f,\ N_p \cup \{\overset{s}{\texttt{B}},\ \overset{f}{\texttt{E}}\},\ add\_exit(\texttt{t}, E_p, W_p, f) \cup \{s \to s_p,\ f_p \to f\} \\ \qquad ,\ W_p / \{(\texttt{t}', -, -) \in W_p \mid \texttt{t}' = \texttt{t}\} \rangle\end{array}}$$

where, $add\_exit(\texttt{t}, E, W, f) = Ep \cup \{i \curvearrowright f \mid (\texttt{t}, i, \curvearrowright) \in W\} \cup \{i \rightsquigarrow f \mid (\texttt{t}, i, \rightsquigarrow) \in W\}$

Fig. 5 CFG construction rules.

detailed role of W is discussed in the following sections.

3.2 Construction Rules

For a given Esterel program, rules to construct its corresponding CFG are represented as below.

$$program \triangleright \langle\, s,\ f,\ N,\ E,\ W \rangle$$

Figure 5 lists the CFG construction rules for Pure Esterel statements.

`nothing`, `pause`, and `emit s`: A graph with a single node, which has the node type for the corresponding statement, is generated.

`exit t`: A CFG of exception raising statement is an unconnected graph with two nodes, $\overset{s}{\texttt{exit t}}$ and a dummy node $\overset{f}{\texttt{E}}$. Control flow never falls through the statements

following the `exit t`. Therefore, CFGs of the statements are to be connected to the dummy node, which is unreachable from the `exit t` node. Since the target node of the control flow transfer is determined afterward, when the corresponding `trap t in` p `end` statement is processed, we add a tuple of the trap identifier `t`, the index of the source node s, and the edge type $\curvearrowright$ to W.

p;q: For a sequence of statements, the CFGs of p and q are connected by a normal edge from the finish node of p (f_p) to the start node of q (s_q). The start and finish nodes of the result CFG are the start node of p (s_p) and the finish node of q (f_q), respectively.

`loop` p `end`: For a `loop` statement, we add an edge $f_p \rightarrow s_p$, which connects the finish node of p to the start node of p in order to represent repetition of p. Then the start node of the loop, $\overset{s}{\texttt{B}}$, is connected to the start node of the loop body p by an edge $s \rightarrow s_p$. Since loops in Pure Esterel language are apparently unbounded, control flow from the end of a loop body never falls through the following statements. So the finish node $\overset{f}{\texttt{E}}$ is not connected to the rest of the CFG, similar to the case of exception raising statements.

`present s then` p `else` q `end`: From the start node $\overset{s}{\texttt{B}}$ a normal edge guarded by `s` is connected to the start node of p (s_p), and a normal edge guarded by $\neg$`s` is connected to the start node of q. The finish nodes of both p and q are connected to the finish node $\overset{f}{\texttt{E}}$ by normal edges.

`signal s in` p `end`: Since local signal declaration statements do not define actual execution, the start and finish nodes are simply connected to the start and finish nodes of p by normal edges, respectively.

p||q: This rule is described in 3.3

`suspend` p `when s`: A `suspend` statement pauses the execution of its body p in instants when the signal `s` is present. Thus, when `s` is present, the control flow in p stays at `pause` statement which causes an instant change in the previous instant. This behavior can be represented by adding a self edge whose condition is presence of `s` for every `pause` node in p. The edges leaving the `pause` nodes also need to be guarded with absent of `s`. If the edges to be added already exist, only their conditions are updated correspondingly. The start and finish nodes of the resulting CFG are connected to the start and finish nodes of p by normal edges.

`trap t in` p `end`: A `trap` statement catches exceptions raised in its body. As mentioned in the construction rule of `exit t`, the CFG of p keeps track of the information of exceptions raised in p using the set W_p. If there are any incomplete edges whose trap identifier is `t` in W_p, their target is set to the finish node $\overset{f}{\texttt{E}}$, and they are moved to the edge set E from W_p. Because we assume that any invalid `exit` is replaced with `nothing` through preprocessing, all incomplete edges in W_p should be completed by the corresponding `trap` statement. The start and finish nodes are connected to the start and finish nodes of p by normal edges, as with other block statements.

```
1  module test:
2  output A, B, C;
3
4  trap U in
5     trap T in
6          emit A;
7          pause;
8          exit T
9        ||
10         emit B;
11         pause;
12         exit U
13    end trap;
14    emit C;
15 end trap
16
17 end module
```

Fig. 6 An example of nested trap statements.

3.3 CFG Construction Rule for Parallel Execution

As mentioned in Sect. 2.2, when preemption takes place in the middle of parallel execution, it requires a delicate handling that suits the meaning of each statement. For an example in Fig. 6, `exit T` and `exit U` are performed in the same instant. Since `trap U` includes `trap T`, `exit U` has a higher priority and both control flows go outside of `trap U`.

For the case of p||q, if `exit` is performed from p, q will terminate either after performing until the current instant or after performing the end of q if the instant is the last one. If an `exit` statement is performed in q at the same instant, the one that corresponds to the outer `trap` is processed.

In this paper, we introduce a *may-exit* edge to represent the situations described above. A may-exit edge is an edge for the statement that the control of a program is likely to be terminated when the other program that is performed in parallel is terminated by `exit`.

As shown in Fig. 5, the rule of p||q specifies to construct CFGs for p and q and to connect the start (s_p, s_q) and end (f_p, f_q) nodes of subparts to the start (s) and end (f) nodes of the resulting CFG with parallel edges, respectively. Then, incomplete may-exit edges are added for the following nodes based on the information from W_p, which is a set of `exit`s included in p, and E_q, which is a set of edges in q.

- precedent node(j) of the last node in q
- precedent node(j) of `pause` included in q
- low priority node(j) among `exit t` in q

The same process is repeated for W_q and E_p.

3.4 Soundness of CFG Construction

A CFG of an Esterel program is *sound* if and only if the CFG

includes all possible execution paths of the Esterel program. Here, we discuss about the soundness of generated CFGs by our CFG construction rules.

Control flows of Esterel programs consist of *sequence*, `present`, `suspend`, `trap`/`exit`, and *parallel* statements.

Most control flows are explicitly represented by syntactic structures. *sequence*, `trap`/`exit`, and `present` statements in a single thread are base cases for control flows, and their control flows are exposed through program codes. A `suspend` statement generates control flows for all pending points in its sub-statements, and the (*suspend*) rule reflects this semantics to all `pause` nodes in the `suspend` block. A CFG for a *parallel* statement without `trap` statements is the combination of CFGs for all sub-threads.

There are implicit control flows that are not explicitly represented by program codes. When a *parallel* statement is contained in a trap block, even if one sub-thread exits the trap block, all sub-threads must exit the trap block by Esterel semantics [4]. We add may-exit edges to all possible program points of the other sub-threads for an exit statement of one thread in Sect. 3.3, and a generated CFG by our construction rules contains all possible exit paths of all sub-threads.

As discussed above, our construction rules consider every possible execution cases of Esterel programs. Therefore, a generated CFG by our rules is sound.

4. Experimental Results

We implemented a control flow analyzer based on the proposed algorithm in OCaml. The front-end of the control flow analyzer first *desugars* Esterel statements into Pure Esterel statements and then performs the preprocessing described in Sect. 3. If an input program consists of several modules, we dismantle each module into a set of statements and then merge them into a single module using Columbia Esterel Compiler [15].

Our CFG construction algorithm is depicted in Fig. 7. To construct a CFG of a program, we apply proper construction rules in Fig. 5 to the program recursively. The time and memory complexities of CFG construction for a program that has no `parallel`, `suspend`, and `trap` statements increase linearly ($O(kN)$) with respect to the size of input program (N) [†]. In case of `suspend` statements, because our algorithm must traverse its sub-CFG to adjust edges by (*suspend*) rule, the time comlexity increases to $O(N^2)$. When an input program contains combinations of `trap` and `parallel`, it is necessary to traverse their sub-CFGs for adding may-exit edges before joining them by (*parallel*) rule. In this case, the time complexity may be worse, but does not exceed $O(kN^2)$ [††].

Figure 8 shows the control flow graph of the nested trap example in Fig. 6. Bold, dashed, and dotted edges in the figure correspond to parallel, exit, and may-exit edges, respectively. The rest are normal edges. Exit and may-exit edges have their trap identifiers as labels.

When the execution of `exit U` (at node 11) causes weak preemption, the other thread may be preempted right after executing either `emit A` (at node 5) or `exit T` (at node 7). This situation is covered by the may-exit edges 5 $\rightsquigarrow$ 16 and 7 $\rightsquigarrow$ 16. Similarly, the weak preemption of `exit T` (at node 7) needs may-exit edges 9 $\rightsquigarrow$ 14 and 11 $\rightsquigarrow$ 14. In this case, however, the may-exit edge from node 11 is omitted because the statement at the node, `exit U`, itself raises the exception outer than `T`. Esterel semantics gives higher priority to the outer trap when multiple exceptions are raised in the same instant.

In the actual execution trail of the above example, `exit T` and `exit U` are always executed in the same instant and thus the control flows go through 7 $\rightsquigarrow$ 16 and 11 $\rightarrow$ 16. Even though the may-exit edge 7 $\rightsquigarrow$ 16 does not explicitly occur in the program text, the proposed CFG construction rule is able to find it. In the meantime the may-exit edges 5 $\rightsquigarrow$ 16 and 9 $\rightsquigarrow$ 14, which are never realized in the actual execution, are also included in the CFG due to the over-approximation taken by the proposed method.

Table 1 lists experimental results for several Esterel benchmark programs collected from Estbench [22] and Ramesh's case studies [23]. Table 2 gives the CFG construction time and memory usage on the test system - a XEON E5440@2.83 GHz (QuadCore)/16 GB running Linux.

```
1  proc generateCFG(ast) =
2    choose (ast.type)
3      /* base cases */
4      case NOTHING :
5        cfg := /* apply (nothing) */
6      case PAUSE :
7        cfg := /* apply (pause) */
8      ...
9
10     /* recursive cases */
11     case SEQUENCE :
12       P   := generateCFG(ast.child[0])
13       Q   := generateCFG(ast.child[1])
14       cfg := /* apply (sequence) with P,Q */
15     case LOOP :
16       P   := generateCFG(ast.child[0])
17       cfg := /* apply (loop) with P */
18     ...
19   end choose
20   return cfg
21 end proc
22
23 proc main(pgm) =
24   /* parsing and pre-processing */
25   ast := parse(pgm)
26
27   /* generate CFG */
28   cfg := generateCFG(ast)
29 end proc
```

Fig. 7 Pseudo code for CFG construction.

[†]All statements except `parallel`, `suspend`, and `trap` statements produce a constant number of nodes and edges.

[††]The number of exit and may-exit edges cannot over the maximum depth (k) of nested trap statements in the program. Although k may reach $O(N)$ theoretically, it usually remains constant in many practical programs.

Table 1 Our experiments: CFG construction.

Program	Description	Size (bytes)	SLOC	# of nodes	# of edges	may-exit #	may-exit ratio
atds100	video generator	22,038	622	987	1,738	245	14.1%
mca200	shock absorber controller from the Polis distribution	227,599	5,354	1,037	1,440	0	0.0%
mejia	Cyclic: non-commutative data actions prevents compilation	9,782	361	510	790	102	12.9%
tcint	turbochannel bus controller	9,364	353	504	779	33	4.2%
ww	wristwatch from the Esterel distribution	11,952	360	591	861	14	1.6%
dlx	Esterel model of the DLX processor	7,862	334	426	618	0	0.0%
fbus	implementation of future bus protocol in Esterel	6,287	285	673	943	21	2.2%
Average		42,122	1,096	675	1,024	59	5.8%

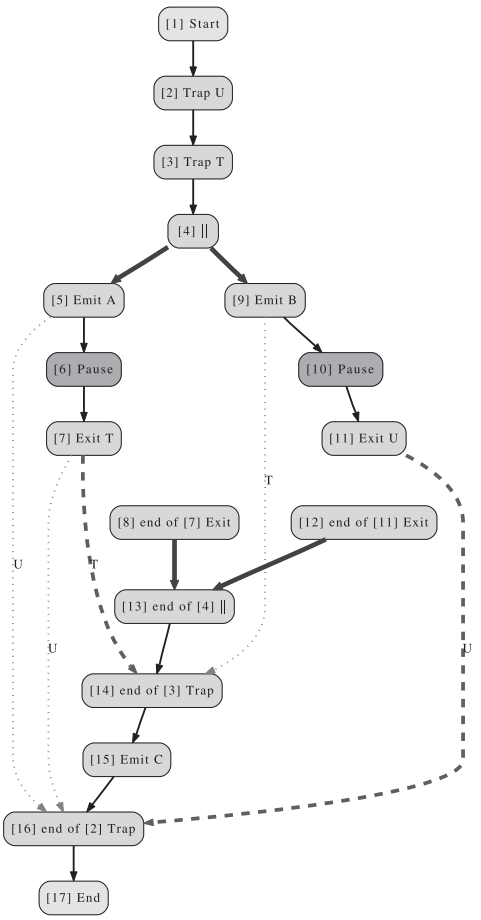


Fig. 8 Control flow graph of the example in Fig. 6.

Table 2 Our experiments: Time and memory usages.

Program	Time (ms) CFG	Time (ms) schizo.	Memory (KB) CFG	Memory (KB) schizo.
atds100	76	104	1,332	1,900
mca200	60	<1	1,247	95
mejia	20	4	713	209
tcint	16	8	624	366
ww	24	4	705	217
dlx	12	4	500	188
fbus	28	<1	811	50
Average	34	18	848	432

```
1  trap DETEC_ERR in
2      await ERR do
3          emit ERR_RECEPTION
4      end await
5    ||
6      await DD do
7          emit ERR_RECEPTION
8      end await
9    ||
10     await
11         case DF do
12             emit ERR_RECEPTION
13         case ATTEND_DF
14     end await
15   ||
16     await ERR_RECEPTION do
17         exit DETEC_ERR
18     end await
19 end trap
```

Fig. 9 An error handling pattern excerpted from `mejia`.

For the cases of `mca200` and `dlx`, no may-exit edge is created; `mca200` does not use `trap` that includes a parallel construct, and `dlx` does not use trap at all. Although `tcint`, `ww`, and `fbus` use various patterns of `trap`, the portions of the pattern that performs `exit` with the parallel execution are rare.

However, about 14% of the total edges were recognized as may-exit edges in `atds100` and `mejia`, because they consist of various combinations of `trap`, parallel execution, and `exit` statements as shown in Fig. 9. `exit` in line 17 inserts 9 may-exit edges at the precedence nodes of `await` [†] in lines 2, 6, and 10, and the end of each parallel process. Most edges have possibilities to be really executed, but some are not executed in real environment. In case of Fig. 9, the predecessors of `await` statements in lines 2, 6, and 10 contain the start node of the *parallel* statement in lines 2-18, so we add a may-exit edge to the node. This edge means that the *parallel* statement may be exited in its starting instant. However, `await` in line 16 consumes one tick itself, so the `exit` statement in line 17 cannot be executed in first instant, therefore the added may-exit edges are infeasible and they are unreachable in run-time. This is the reason why our algorithm over-approximates synchronization mechanism of Esterel. However, most may-exit edges are executed actually in the benchmark programs.

[†] `await` is an additional statement that waits until the target signal presents, and `await S` can be replaced with `trap T in loop pause; present S then exit else nothing T end end end`.

4.1 Case Study: Detecting Schizophrenia via Graph Reachability

In Esterel, since a `loop` statement terminates and restarts in the same instant, a statement to be executed when the loop terminates can be executed again when the loop restarts. A statement is called schizophrenic if it is executed more than once in an instant. Schizophrenic statements can cause problems in translation into hardware circuits [4]. So Esterel compilers must solve these problems.

We developed a schizophrenia detection algorithm [19] on our CFGs of Esterel programs and our detector shows more precise result than a previous detector [24], and the detection algorithm can be easily implemented using graph reachability. Moreover, our detection algorithm based on graph reachability can be more easily applicable to existing compilers than the previous one [24] based on abstract interpretation [25]. Table 2 gives the detection time and memory usage on the test system.

If a CFG does not include all control flows, the detecting algorithm cannot detect all schizophrenic problems. However, our CFGs compute all possible control flows in advance, so the detection algorithm can look into schizophrenic problems on all possible execution paths. So, if a schizophrenic problem exits, our algorithm must detect it. In other words, a program that passes the detector developed on our CFGs has no schizophrenic problems described in [19].

5. Conclusion

Esterel has various imperative features, and behaviors of an Esterel program can be described with a CFG. Because previous works [12]–[14] focus on correct and complete representations, they are not proper for practical analyzers based on flow analyses of existing imperative languages.

We present over-approximated CFG construction rules that are sound and practical. Our CFGs show program structures well. Though some CFGs contain infeasible paths that may be unreachable during execution time, the experimental results show that the fraction of such paths is neglectable. Moreover, the results also show that the CFG construction algorithm uses reasonably small amount of time and memory.

Our key contribution is to expose invisible interferences among threads without any accompanying data structures or handlers. So, flow analyses of existing imperative languages can be applied to Esterel programs on our CFGs. As a matter of fact, we used the proposed CFGs to detect a well-known problem in Esterel [19], and achieved precise results. This work can be a useful basis of graph-based analysis on Esterel programs.

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," Proc. IEEE Embedded Systems, vol.91, no.1, pp.64–83, 2003.

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

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

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

[6] D. Potop-Butucaru, S. Edwards, and G. Berry, Compiling Esterel, Springer, 2007.

[7] 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, Nov. 2005.

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

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

[10] G. Berry, "Preemption in concurrent systems," Proc. 13th Conference on Foundations of Software Technology and Theor. Comput. Sci., pp.72–93, Springer-Verlag, London, UK, 1993.

[11] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier Science Inc., New York, NY, USA, 1977.

[12] S. Ramesh, A. Kulkarni, and V. Kamat, "Slicing tools for synchronous reactive programs," ACM SIGSOFT Software Engineering Notes, vol.29, no.4, pp.217–220, 2004.

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

[14] K. Schneider, "The synchronous programming language Quartz," Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

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

[16] D. Potop-Butucaru and R.d. Simone, "Optimizations for faster execution of esterel programs," MEMOCODE '03: Proc. First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, pp.227–236, IEEE Computer Society, Washington, DC, USA, 2003.

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

[18] O. Tardieu and S.A. Edwards, "Approximate reachability for dead code elimination in esterel*," Automated Technology for Verification and Analysis, ed. D. Peled and Y.K. Tsay, LNCS, vol.3707, pp.323–337, Berlin Heidelberg, Springer, 2005.

[19] J. Yun, C. Kim, S. Seo, T. Han, and K. Choe, "Refining schizophrenia via graph reachability in esterel," Seventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE' 2009), Cambridge, Massachusetts, USA, July 2009.

[20] J. Brandt and K. Schneider, "Static data-flow analysis of synchronous programs," Seventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE' 2009), Cambridge, Massachusetts, USA, July 2009.

[21] G. Berry, The Esterel Primer, included in the esterel distribution. available on http://www.inria.fr/meije/personnel/gerard.berry.html ed., 1998.

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

[23] S. Ramesh, "Ramesh's homepage."

http://www.cse.iitb.ac.in/~ramesh/
[24] O. Tardieu and R. de Simone, "Instantaneous termination in pure esterel," Proc. 10th International Static Analysis Symposium (SAS'2003), LNCS, vol.2694, pp.91–108, Springer-Verlag, 2003.
[25] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," 4th ACM Symposium on Principles of Programming Languages, pp.238–252, ACM Press, Los Angeles, CA, 1977.

Chul-Joo Kim received his B.S degree in Computer Engineering from Dongguk University, Korea, in 2001 and M.S degree in Computer Science from KAIST, Korea, in 2003. He is currently a Ph.D. degree student in the Dept. of Computer Science, KAIST. His current research interests include programming languages, program static analysis and compiler optimization.

Jeong-Han Yun received his B.S and M.S degrees in Computer Science from KAIST, Korea, in 2001 and 2003. He is currently a Ph.D. degree student in the Dept. of Computer Science, KAIST. His current research interests include program analysis and embedded system design.

Seonggun Kim received his B.S in electrical engineering from KAIST, Korea, in 2003. He is currently an M.S.-Ph.D. joint degree student in the Dept. of Computer Science, KAIST. His current research interests include parallel processing, multi-core architectures and optimizing compilers.

Kwang-Moo Choe received his B.S degree in Electrical Engineering from Seoul National University, Korea, in 1976, and M.S and Ph.D. degree in computer science from KAIST, Korea, in 1978 and 1984. He is currently a professor in the Dept. of Computer Science, KAIST. His current research interests include formal language theory, parallel evaluation of logic programs, and optimizing compilers.

Taisook Han received his B.S and M.S degrees in electrical engineering from Seoul National University, Korea, in 1976, M.S degree in computer science from KAIST, Korea, in 1978, and Ph.D. degree in computer science from University of North Carolina at Chapel Hill, USA, in 1995. He is currently a professor in the Dept. of Computer Science, KAIST. His current research interests include programming language theory, and design and analysis of embedded systems.