Static Detection of Livelocks in Ada Multitasking Programs

Johann Blieberger[1], Bernd Burgstaller[2], and Robert Mittermayr[1,3]

[1] Institute for Computer-Aided Automation, TU Vienna, Austria
[2] School of Information Technologies, The University of Sydney, Australia[⋆]
[3] Mediornet GmbH, Vienna, Austria

Abstract. In this paper we present algorithms to statically detect livelocks in Ada multitasking programs. Although the algorithms' worst-case execution time is exponential, they can be expected to run in polynomial time. Since the problem is strongly connected to finding infinite loops, which is undecidable in general, our algorithms compute only an approximation to the real solution of the problem. As a consequence our algorithms may compute false positives.

1 Introduction

Concurrent programming is a complex task. One reason is that scheduling exponentially increases the possible program states. Thus a dynamic execution order of the statements executed in parallel is introduced. In general this leads to different behavior between different runs of a program, even on the same input. Because of the nondeterministic behavior, faults are difficult to detect. Static program analysis, which has been used since the beginning of software, can be a valuable aid for the detection of such faults.

One of the problems with concurrent programming are *livelocks*, sometimes also called *infinite internal chatter*. In the context of process calculi, e.g. CSP [15], the term *divergence* is frequently used to denote infinite internal actions. Livelocks are sometimes also referred to as *spinning*. From "outside", a deadlocked and a livelocked system look like no progress is made. In the case of a deadlock this is true, but in the case of a livelock computation still goes on.

The literature contains different definitions for livelock; [14] classifies livelock by three categories: starvation (e.g. [26]), infinite execution (e.g. [33]), and breach of safety properties (e.g. [27]). Our approach addresses programs in the second category. In [32], a livelock is defined as

> "... a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work."

This and other definitions indicate that in contrast to deadlocks, livelocked systems still do some (although not useful) work.

[⋆] This work has been partially supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190.

The above definition of livelock is dynamic in its nature. To devise a static analysis method, we need static program properties that imply a (potential) livelock at run-time.

To distinguish "useful" computations from "purposeless" computations which might be going on during a livelock, we require "useful" computations to have an externally observable effect. Externally observable effect can manifest itself in a multitude of ways in the source code of a program: it can be as simple as writing output to a terminal, or more involved as with memory-mapped I/O or messages sent on a communication channel. In this paper, we model the external observer by task τ_{ext}, which can be envisioned as an additional task that is external to the tasks of the system under consideration. We assume that communication statements to this external observer task are marked by a suitable `pragma` in the source code of the program.

Intuitively, a set T of tasks is livelocking, if each task $\tau \in T$ contains an infinite loop within which it communicates with another task from T, and no communication with the external task τ_{ext} takes place. (NB: this precludes indirect communication with τ_{ext} via further tasks $\tau' \notin T$ as well.)

Definition 1 *A task τ is said to communicate with task $\overline{\tau}$, if*
1. *τ calls an entry of $\overline{\tau}$, or*
2. *τ contains an accept statement which is called by $\overline{\tau}$, or*
3. *τ writes a protected object or a shared variable, which is read by $\overline{\tau}$ (or vice versa), or*
4. *τ and $\overline{\tau}$ perform memory mapped I/O on the same memory cells.*

To detect livelocks, we have to find all infinite loops in a set of tasks and determine those tasks with which they communicate. In this paper we restrict ourselves to communication patterns (1) and (2). An analysis including the other patterns as well is possible, but requires more technicalities such as a relation modeling tasks which execute concurrently (cf. [3]). We construct the *control flow graphs (CFGs)* of tasks and determine *extended regular expressions (EREs)* that contain loop properties (i.e., finite vs. infinite, executed at least zero times vs. executed at least once). Morphisms on EREs allow us to determine the communication behavior of tasks. Because finding infinite loops is *undecidable* in general, our approach has to find a conservative approximation of the problem. As a consequence our approach delivers *false positives*, but we present strategies to reduce the number of false positives.

The remainder of the paper is organized as follows. In Section 2 we give definitions and preliminary results. In Section 3 we describe our approach algorithmically. In Section 4 we survey related work, before we conclude the paper and describe future work in Section 5.

2 Definitions and Preliminaries

Definition 2 *A* control flow graph *(CFG) $G\langle N, E, r, x\rangle$ is a directed graph with node set N, edge set $E \subseteq N \times N$, and a root node r such that there exists a path from r to every other node in N. Node x denotes the exit node of G.*

CFGs are used to represent programs. In this case nodes represent basic blocks and edges represent the control flow between nodes. Note that CFGs for Ada programs can be generated as for any other program written in different languages. For example, select statements which are not present in other languages can be handled like case statements.

Definition 3 *An* extended regular expression (ERE) *is a regular expression (RE) (see e.g. [34]) with additional iteration schemes. In more detail, an ERE has operators $\cup$, $\cdot$ and the iteration schemes $\oplus$, $\circledast$, $+$, and $*$; the empty word is denoted by ε. The iteration schemes are defined by the following table.*

$\oplus$	*finite loop with at least one iteration*
$\circledast$	*finite loop with at least zero iterations*
$+$	*infinite loop with at least one iteration*
$*$	*infinite loop with at least zero iterations*

Examples of $\oplus$- and $\circledast$-loops are `for`-loops,

```
for i in a..b loop ... end loop;
```

where `b`$\geq$`a` in case of a $\oplus$-loop. A simple example of a $+$-loop is

```
loop ... end loop;
```

A simple example of a $*$-loop is

```
while cond loop ... end loop;
```

where the value of `cond` does not change inside the loop. Note that if `cond` evaluates to false, the loop body is not executed.

Definition 4 *Let χ be a standard RE or an ERE. If for each $n \in \mathbb{N}$, $n \geq 1$ there exists a path $\pi_n(r, x)$ such that $\pi_n(r, x)$ contains χ n times, then we say that the CFG G contains an* infinite χ^+-path. *In addition, if there exists a path $\pi_0(r, x)$ such that $\pi_0(r, x)$ does not contain χ, then we say that G contains an* infinite χ^*-path.

Our next step is to define rewrite rules on EREs that determine whether a given task τ communicates with task τ' (cf. Definition 1). We consider extended regular expressions over the alphabet $\{\rho\}$, where ρ denotes a communication statement. An example for such an ERE is given by $(\rho \cup \rho \cdot (\varepsilon \cup \rho^*)) \cdot \rho^{\oplus}$. The rewrite rules are constructed in a way such that the information that ρ is contained in a $+$-loop is conserved.

If we use the conventions $\rho^0 = \varepsilon$ and $\rho^1 = \rho$, we can define the rewrite rules "in terms of the exponents". For example $((\rho^1) \wedge +) \wedge \circledast = \rho^+ \wedge \circledast = (\rho^+)^{\circledast} = \rho^*$ corresponds to $(1 \wedge +) \wedge \circledast = + \wedge \circledast = *$.

Let $\triangle = \{0, 1, \oplus, \circledast, +, *\}$ be the set of exponents. The rules are given in Tables 1(a), 1(b), and 1(c). Left operands can be found on the left of the tables, right operands on the top. For example $* \cdot 1 = +$ and $1 \cdot * = +$.

The "$\wedge$"-table is set up as follows: We consider nested loops such that the behavior of the inner loop (the left operand) is described by $x \in \triangle$ and that of the outer loop (the right operand) is described by $y \in \triangle$. In order to find a description of $(\rho^x)^y$ we have to determine the behavior of the nested loops.

$\wedge$	0	1	$\oplus$	$\circledast$	$+$	$*$
0	0	0	0	0	0	0
1	0	1	$\oplus$	$\circledast$	$+$	$*$
$\oplus$	0	$\oplus$	$\oplus$	$\circledast$	$+$	$*$
$\circledast$	0	$\circledast$	$\circledast$	$\circledast$	$*$	$*$
$+$	0	$+$	$+$	$*$	$+$	$*$
$*$	0	$*$	$*$	$*$	$*$	$*$

(a) Operator "$\wedge$"

$\cdot$	0	1	$\oplus$	$\circledast$	$+$	$*$
0	0	1	$\oplus$	$\circledast$	$+$	$*$
1	1	$\oplus$	$\oplus$	$\oplus$	$+$	$+$
$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$+$	$+$
$\circledast$	$\circledast$	$\oplus$	$\oplus$	$\circledast$	$+$	$*$
$+$	$+$	$+$	$+$	$+$	$+$	$+$
$*$	$*$	$+$	$+$	$*$	$+$	$*$

(b) Operator "$\cdot$"

$\cup$	0	1	$\oplus$	$\circledast$	$+$	$*$
0	0	0	0	0	0	0
1	0	1	$\oplus$	$\circledast$	$+$	$*$
$\oplus$	0	$\oplus$	$\oplus$	$\circledast$	$+$	$*$
$\circledast$	0	$\circledast$	$\circledast$	$\circledast$	$*$	$*$
$+$	0	$+$	$+$	$*$	$+$	$*$
$*$	0	$*$	$*$	$*$	$*$	$*$

(c) Operator "$\cup$"

Γ	
0	*false*
1	*false*
$\oplus$	*false*
$\circledast$	*false*
$+$	*true*
$*$	*false*

(d) Function Γ

Fig. 1. Operators and Function Γ

For example consider $(\rho^{\circledast})^{+}$. In this case the inner loop can be executed zero times which means that even when the outer loop is executed more than once, the overall behavior can only be either $\rho^{\circledast}$ or ρ^{*}. On the other hand, if the inner loop is executed at least once, and since the outer loop allows for an arbitrary number of iterations, the overall number of iterations cannot be bounded. Thus we get $(\rho^{\circledast})^{+} = \rho^{*}$ or given in terms of the exponents $\circledast \wedge + = *$.

By similar observations the contents of the "$\wedge$"-table can be completed. Surprisingly, the "$\wedge$"-operator commutes.

For the "$\cdot$"-table we consider concatenation of loops. For example $\rho \cdot \rho^{*}$ gives ρ^{+} or for short $1 \cdot * = +$. For simplicity we assume $1 \cdot 1 = \oplus$. Note that also the "$\cdot$"-operator does commute.

For the "$\cup$"-table we have to consider the effects of if, case, and select statements. For example $\rho \cup \rho^{\oplus}$ means that on both branches there exists at least one ρ. Hence $1 \cup \oplus = \oplus$. On the other hand, $\rho \cup \rho^{\circledast}$ contains a path without a ρ on the right side. Thus $1 \cup \circledast = \circledast$. By symmetry, the "$\cup$"-operator has to commute.

Assume that an extended regular expression is represented by an expression tree. Then the rules given in Tables 1(a), 1(b), and 1(c) can be applied to the operators in the tree from bottom up. Thus after a finite number of rewrites we arrive at an expression of the form ρ^{x}, where $x \in \triangle$. Obviously, the number of rewrites is equal to the number of operators in the extended regular expression.

Definition 5 *Let $\mathbb{B} = \{false, true\}$ be the set of boolean truth values. Then we define a function $\Gamma : \triangle \rightarrow \mathbb{B}$ by enumeration in Table 1(d).*

If function Γ is applied to the simplified result of an ERE, it delivers *true* if and only if the underlying CFG contains an infinite ρ^{+}-path. If this is not the case, Γ returns *false*.

In [35] Tarjan presents a fast algorithm to determine a regular expression for a given CFG. These regular expressions describe all paths in the CFG from the start node to the exit node compactly. Tarjan's or any other algorithm suitable for this problem can easily be adapted to extended regular expressions.

Theorem 1 *Let $G\langle N, E, r, x\rangle$ be a CFG. By $\alpha[\![\rho|E']\!]$ we denote the extended regular expression which we get from α by replacing each occurrence of $e \in E' \subseteq E$ with ρ and by replacing each occurrence of all the other $e' \in E \setminus E'$ with ε.*

As noted above, by successively applying the rules given in Tables 1(a), 1(b), and 1(c) to $\alpha[\![\rho|E']\!]$ we finally get a result $\sigma = \sigma(\alpha, E') = \rho^{x}, x \in \triangle$. Now,

$$\Gamma(x) = \begin{cases} true, & \textit{if } G \textit{ contains an infinite } \rho^{+}\textit{-path,} \\ false, & \textit{otherwise.} \quad \Box \end{cases}$$

```
Compute $\mathcal{C}(\tau)$
1   $\mathcal{C}(\tau) := \{\}$
2   for all $a \in R(\tau)$ do
3       Compute $\alpha[\![\rho \mid E'_\tau(a)]\!]$
4       Apply rules to get $\rho^x, x \in \triangle$
5       Let $\mathcal{C}_a(\tau)$ be the set of tasks with which $\tau$ communicates via $a$.
6       if $\Gamma(x) = true$ then
7           $\mathcal{C}(\tau) := \mathcal{C}(\tau) \cup \mathcal{C}_a(\tau)$
8       endif
9   endfor
```

Fig. 2. Algorithm to Compute $\mathcal{C}(\tau)$

3 Algorithm

Let $R(\tau)$ be the union of all entry calls performed by task τ and of all accept statements being part of τ. If $\tau = \tau_{\text{ext}}$, then $R(\tau) = R(\tau_{\text{ext}})$ contains all communication statements to this external observer task that, as stated above are marked by a suitable `pragma` in the source code of the program.

Each statement $a \in R(\tau)$ can appear in several places in the source code, and thus it can appear in several nodes of the corresponding CFG. Let $N_\tau(a)$ be the set of nodes of the CFG where a appears, or more formally: Let $a \in R(\tau)$ and $N_\tau(a)$ be the set of nodes of the CFG $G\langle V, E, r\rangle$ of τ where a appears in the basic block mapped to $N_\tau(a)$.

Let $E'_\tau(a) = \{(u \to v) \in E \mid v \in N_\tau(a)\}$ be the set of all edges targeting the nodes $N_\tau(a)$.

Let $\mathcal{C}(\tau)$ denote the set of tasks with which task τ communicates. Set $\mathcal{C}(\tau)$ is computed by the algorithm given in Figure 2.

The algorithm proceeds as follows:

1. For each task (type) τ and for each communicating statement a, it computes EREs.
2. The result of simplifying the ERE determines whether τ communicates to some task via a.
3. In $\mathcal{C}(\tau)$ the tasks with which τ communicates are aggregated.
4. Upon completion, $\mathcal{C}(\tau)$ contains all tasks with which τ communicates.

Concerning interprocedural analysis we assume that inlining is performed. This works well as long as there are no recursive subroutines. For recursive subroutines we apply the following solution: If there exist paths (from the start to the exit node of the CFG of the recursive subroutine) not containing a recursive call which can be described by some ERE β and which simplifies to $\rho^x, x \neq 0$ by applying our rules from above then we replace all recursive calls with ρ^x. Otherwise we replace all recursive calls with ε. Obviously, this construction is correct.

It is possible that a task calls an entry of the same task type (e.g. consider an array of tasks where each task communicates with its neighbors). Such "recursive" entry calls are handled correctly by our algorithm in that $\mathcal{C}(\tau)$ simply contains τ itself.

```
procedure Main is
  task Server is
    entry E;
  end Server;
  task Client is
  end Client;
  task body Server is
  begin
    loop
      select
        accept E;
      or
        terminate;
      end select;
    end loop;
  end Server;
  task body Client is
  begin
    loop
      Server.E;
    end loop;
  end Client;
begin
  Server.E;
end Main;
```

Fig. 3. A Simple Livelocking Example

Definition 6 *We define the* communication graph $CG\langle T, E_{\mathcal{C}}\rangle$ *where T is the set of all task types including the environment task, i.e., the main program and the external observer task τ_{ext} in the analyzed program and*

$$E_{\mathcal{C}} = \{(\tau_1 \rightarrow \tau_2) \mid \tau_2 \in \mathcal{C}(\tau_1)\}.$$

Theorem 2 *If the communication graph $CG\langle T, E_{\mathcal{C}}$ of a program is not weakly connected (cf. [29]), then the program contains a livelock.*

Proof. By construction of $CG\langle T, E_{\mathcal{C}}\rangle$, the algorithm to compute $\mathcal{C}(\tau)$ (see Figure 3), and Theorem 1.

3.1 Example

Figure 3 shows the Ada source code of a simple livelocking task set. The CFGs of tasks `Server` and `Client` are shown in Figures 4(a) and 4(b). The CFG of the environment task (`Main`) is very simple and is not depicted.

The accept statement in `Server` is situated on edge $(1 \rightarrow 2)$. Node 3 is the `terminate` statement. Hence we obtain the ERE $\alpha[\![\rho|\{(1 \rightarrow 2)\}]\!] = (\rho \cdot \varepsilon)^+ \cdot \varepsilon$ for `Server`. Applying the simplification rules we get $\alpha[\![\rho|\{(1 \rightarrow 2)\}]\!] = \rho^+$. Hence $\mathcal{C}(\texttt{Server}) = \{\texttt{Client}\}$.

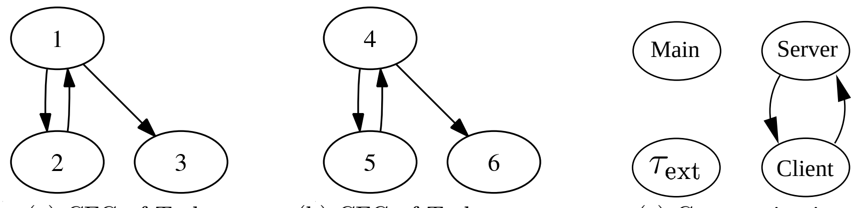
The entry call in `Client` is performed on edge $(4 \rightarrow 5)$. Thus we obtain the ERE $\alpha[\![\rho|\{(4 \rightarrow 5)\}]\!] = (\rho \cdot \varepsilon)^+$. Applying the simplification rules we get $\alpha[\![\rho|\{(1 \rightarrow 2)\}]\!] = \rho^+$. Hence $\mathcal{C}(\texttt{Client}) = \{\texttt{Server}\}$.

For `Main` we get ERE ρ and $\mathcal{C}(\texttt{Main}) = \{\}$.

The communication graph CG is depicted in Figure 4(c). Note that one single entry call in `Main` does not generate an arc in CG from `Main` to `Server`. Since thus CG is not weakly connected, the example contains a livelock by Theorem 2.

3.2 Notes on Time and Space Behavior

Generating CFGs for programs is straight-forward and can be done in linear time. Determining (extended) regular expressions for a given CFG can be done in



(a) CFG of Task `Server` (b) CFG of Task `Client` (c) Communication Graph CG of Example

Fig. 4. Graphs for Example

almost linear time (cf. [35]). In addition there exist linear algorithms to determine whether a graph is weakly connected (see e.g. [11]).

Detecting the correct iteration scheme of loops requires heuristics for non-trivial cases like loops with exit statements. However, we assume that such heuristics can be implemented efficiently.

Applying the rules given in Tables 1(a), 1(b), and 1(c) needs time and space linear in the number of operators in α, which is denoted by $\|\alpha\|$ in the following.

Different algorithms used to solve the path problem produce different values for $\|\alpha\|$. In the worst-case $\|\alpha\| = O(a^n)$ for some $a > 1$, where n denotes the number of nodes of the underlying CFG.

We have observed that Tarjan's algorithm [35] applied to the SPEC2000 benchmark suite produces regular expressions both of polynomial and of exponential sizes. However, we are unable to spot a pattern in the CFGs that would make it possible to predict whether Tarjan's RE size is polynomial or exponential.

In contrast, an algorithm [30] based on decomposition properties of reducible CFGs delivered only polynomial sizes for the SPEC2000 benchmark suite. This algorithm is known to produce exponential RE sizes only if the number of backedges (i.e., loops) is large compared to the overall number of edges in the CFG. Since this is rarely the case in practical applications, this algorithm seems to be a good candidate for producing the extended regular expressions for the purposes of this paper.

3.3 False Positives

First we would like to note that dead code can lead to *false negatives* in our live lock analysis. In addition, performing *dead code analysis* before our algorithm significantly reduces the number of false positives.

As already mentioned above, detecting livelocks requires the detection of infinite loops in programs. Since this problem is undecidable, we cannot solve the livelock problem in the general case, i.e., we have to live with false positives.

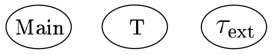
Although Ada has the `loop ... end loop;` kind of statement that allows to explicitly[‡] program infinite loops, general loop statements with exits make it necessary to develop heuristics to determine the iteration schemes needed by our approach. These heuristics are the primary source for false positives. We give an example for such a case below.

If $+$- and $\oplus$-loops are treated as $*$- and $\circledast$-loops, respectively, false positives may arise. For example consider the ERE $(\rho^x)^+$ which simplifies to ρ^+ if $x \in \{+, \oplus\}$ and ρ^* if $x \in \{*, \circledast\}$.

It should however be noted that most Ada multitasking programs are supposed to use the `loop ... end loop;`-construct for infinite loops which is easy to spot. In addition, we do not expect complex loop structures (exit, ...) in safety related or embedded programs.

[‡] In contrast to the `while true loop ... end loop;` style of other programming languages.

```
procedure Main is
  X : Boolean;
  task body T is
  begin
     Write_Ln();
  end;
begin
  if X then
     l: loop
        Write_Ln();  -- Path p
     end loop;
  end if;
end Main;
```



(a) Example CG

Fig. 5. Example: False Positive

3.4 Reducing the Number of False Positives

Theorem 2 is a conservative approximation of the set of programs that may produce a livelock. It requires the CG of a program to be weakly connected, which holds if every task τ (including the program's environment task) meets the following constraints.

1. Every task τ must contain a loop l with a "+" iteration scheme,
2. on every path through the body of l task τ must communicate with the external task τ_{ext} (or with a task that communicates with τ_{ext} and so on...), and
3. every path through the CFG of τ must contain loop l.

The above constraints are so strict that many non-livelocking programs occurring in practice fail to meet them. (i.e., their CGs are disconnected despite the programs being non-livelocking). Fig. 5 contains a program with two tasks that fail to produce a weakly connected CG (the CG is depicted in Fig. 5 (a)). Task T communicates with the external task τ_{ext}, but not within a "+" iteration scheme, and the environment task (corresponding to the body of "Main") contains a program path that does not contain the "+" iteration scheme (i.e., if variable X is false).

To reduce the number of false positives, we lower the abstraction level of our analysis from tasks to loop bodies. Informally, livelocks are caused by infinite loops. Lowering the abstraction level from tasks to loops allows for a more fine-grained analysis and excludes tasks without loops altogether. Deviating from Def. 2, we move the basic blocks of the CFG to the CFG edges (cf. [2]). We defer discussion of nested loops to a latter part of this section and focus our analysis on acyclic program paths through loops. In what follows, the term "loop" denotes the ERE corresponding to the argument of a "$*$" or "+" ERE operator. Exhaustive application of the rewrite rules

$$(R_1 \,|\, R_2) \cdot R_3 \Rightarrow (R_1 \cdot R_3) \,|(R_2 \cdot R_3) \tag{1}$$

$$R_1 \cdot (R_2 \,|\, R_3) \Rightarrow (R_1 \cdot R_2) \,|(R_1 \cdot R_3) \tag{2}$$

to a loop l results in an ERE $p_1 \,|\, \cdots \,|\, p_3$ where each p_i denotes an acyclic program path through the body of l (recall that the discussion of nested loops is deferred).

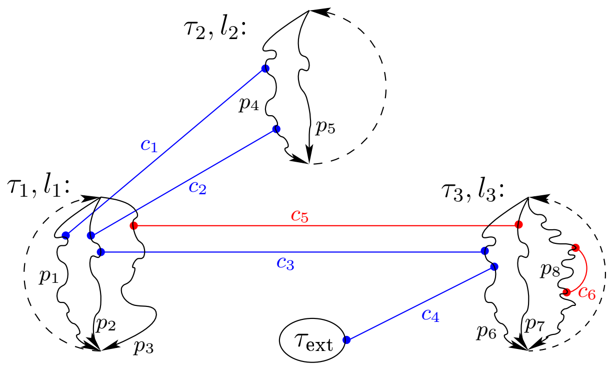


Fig. 6. Example: Communication Behavior of Three Loops $l_1, \ldots, l_3$

Predicate $\text{Com}\tau_{\text{ext}} : \text{ERE} \rightarrow \{\mathit{true}, \mathit{false}\}$ is *true* iff every path through the ERE provided as argument contains a communication with the external task τ_{ext} (e.g., a read/write operation on memory mapped I/O, etc.). Predicate $\text{Com} : \text{ERE} \times \text{ERE} \rightarrow \{\mathit{true}, \mathit{false}\}$ is *true* iff a synchronous communication (i.e., a rendezvous) between the two argument EREs R_1 and R_2 occurs on every path through R_1 and R_2. Note that for the sake of exposition the above predicates are defined on EREs; for our analysis we will apply them only to single acyclic program paths.

Our analysis is based on the observation that not all paths through a loop need to generate communication patterns that contribute to a livelock. As an example, consider Fig. 6. It depicts three loops $l_1 \cdots l_3$. For each loop the acyclic program paths p_i are depicted (a single post-body node collects acyclic program paths; a single back edge connects the post-body node with the loop header; back edges are depicted with dashed arrows). Lines connecting paths p_i and p_j denote intertask communication, i.e., $\text{Com}(p_i, p_j) = \mathit{true}$. Loops l_1 and l_3 livelock at runtime only iff l_1 enters an infinite execution sequence along path p_3, *and* l_3 enters an infinite execution sequence along path p_7. Path p_8 of loop l_3 contains communication with itself, which is possible with task types. Two tasks entering an infinite execution sequence along path p_8 constitute another livelock. No other paths across loops $l_1 \cdots l_3$ can contribute to a livelock.

Eq. (3) defines a predicate to determine paths that cannot contribute to a livelock. Informally, a path p_1 cannot contribute to a livelock if no communication occurs on p_1, or if p_1 communicates with the external task τ_{ext} or with another path that communicates with τ_{ext}. A path that cannot contribute to a livelock is called a *safe* path, otherwise the path is *unsafe*.

$$\begin{aligned}\text{Safe}(p_1) \Leftrightarrow\ & \big(\forall p_2 : \neg\,\text{Com}(p_1, p_2)\big) \\ & \vee \text{Com}\tau_{\text{ext}}(p_1) \vee \big(\exists p_3 : \text{Com}(p1, p_3) \wedge \text{Safe}(p_3)\big)\end{aligned} \tag{3}$$

Returning to the example in Fig. 6, all paths except p_3, p_7 and p_8 are safe.

Fig. 7 depicts the algorithm that we use to detect unsafe paths. The algorithm proceeds in two steps: (1) across all CFGs of all task (types) it computes the

```
Input:
  set $\mathcal{G}$ of CFGs of the input program's task (types)
Output:
  set $\mathcal{U}$ containing sets of acyclic paths across loops that generate livelocks
Algorithm:
  COMPUTE UNSAFE PATHS ($\mathcal{G}$)
    $\mathcal{U} := \{\}, \;\; \mathcal{L} := \{\}$
    for all CFGs $g \in \mathcal{G}$ do
       compute ERE $R = R(g)$
       $\mathcal{L} := \mathcal{L} \cup \mathrm{Rewrite}(\mathrm{Slice}(R))$
    endfor
    for all loops $l \in \mathcal{L}$ do
       for all paths $p \in \mathrm{Pathset}(l)$ do
          if $\neg\,\mathrm{Safe}(p)$ then
             $\mathcal{U} := \mathcal{U} \cup \mathrm{Com}^*(p)$
          endif
       endfor
    endfor
```

Fig. 7. Detection Algorithm for Unsafe Paths

set $\mathcal{L}$ of loops, and (2) determines if the paths through those loops are safe. Procedure Slice uses a pattern matching algorithm to determine EREs that constitute loops (see [4]). Procedure Rewrite applies the rewrite rules of Eq. (1) and Eq. (2). Procedure Com*(p) uses the transitive closure on predicate Com and returns a set containing all paths that path p communicates with. Each such set constitutes a set of unsafe paths that may produce a livelock at run-time.

Nested Loops. Consider a loop l with path expression R. Assume that $R_1 = (a|b)^*$ is a subexpression of R. To determine the communication behavior of l, a conservative approximation of communication with τ_{ext} assumes that R_1 is executed zero times or once. For a loop with a "+" iteration scheme, the conservative approximation is to assume one iteration. The following rewrite rules replace nested loops with their conservative approximations.

$$R^* \Rightarrow (\epsilon \,|\, R) \qquad R^{\circledast} \Rightarrow (\epsilon \,|\, R) \qquad R^+ \Rightarrow (R) \qquad R^{\oplus} \Rightarrow (R) \tag{4}$$

These rules are added to procedure Rewrite. Procedure Slice returns a loop body for every "$*$" and "+" operator across all loop nesting levels (cf. [4]). These loop bodies contain possible nested loops. As an example, consider a loop l with path expression $a \cdot ((b^* \,|((c \cdot (d)^*)^*)$. The sliced loop bodies are $\{b, c \cdot (d)^*, d\}$.

Implementation Considerations. The practicality of the path-based approach depends critically on the number of acyclic paths across loops. This number grows exponentially in the number of if- and case statements and nested loops. In [4] we conducted a study on the complete SPEC95 benchmark suite with over 5000 CFGs. The purpose of this study was to determine the number of acyclic paths across loops for large real-world applications. For 90% of all surveyed CFGs this number was below 4000, which means that our approach is tractable for real-world applications.

4 Related Work

There is a vast amount of work about the detection of deadlocks, but surprisingly few publications to livelock detection in concurrent programming languages like Ada and Java. Livelocks are mostly mentioned as a sideline while treating deadlocks. A variety of work has been done for detecting or avoiding livelocks in routing algorithms and client/server architectures. In this section we focus on the related work concerning detection of livelocks in concurrent non-distributed software.

Examples for early work on livelock and liveness properties in general are [20, 21, 27]. Recent work on liveness and fairness we are aware of include [36].

Techniques for the detection of livelocks in concurrent software are based on petri-nets, model-checking, or CSP.

Petri-Nets. Cheng and Ushijima use extended Petri nets to represent Ada multitasking programs [9]. This representation of a program is analyzed with linear algebra methods to statically detect deadlocks and livelocks. The model includes no cycles and therefore is not strongly connected which is a convenient property for their livelock analysis. If an infinite firing sequence in the Petri net exists a livelock is detected. The existence of a tasking deadlock and/or livelock is shown when a matrix equation has a positive integer solution. The approach cannot handle dynamic creation of tasks, abort statements, and exception handling. Like our approach it also detects some spurious livelocks. Because this method is based on an extended Petri net, it is fundamentally different to our work. It can only detect the effect not its cause. The authors state that much work has to be done to establish a practical static analysis for livelock detection.

Model Checking. The abstraction imposed by models derived from the actual source code generally results in differences between the model and the real program implementation. Thus results given by model checking are in general non-transferable to the real program. Static analysis like our approach do not have this drawback because the application source code is used directly.

In the model-checking tool Spin [17, 16] the models are build in a language called *Process Meta Language* (Promela). Spin enables analyzing liveness properties like non-progress cycles (i.e. livelocks). The detection of livelocks relies on special labels. Therefore such labels have to be placed by the user. Thus it is not fully automated. Both, *Java PathFinder* (JPF) [37] and Bandera [10], act as a front-end enabling model checking concurrent Java software (e.g. livelock checking).

Model checking approaches addressing livelock detection we are aware of include [1, 24, 22, 12, 8].

The indirect approach in [19] addresses generally starvation in Ada programs and is based on the finite delay property of [21].

CSP. The following tools use CSP [15] as a basis to enable the detection of race hazards, deadlock, starvation and especially livelock. All these approaches have similar flaws like in model checking i.e. design specifications or (the other way round) abstractions may have differences to the actual software.

Deadlock Checker is a tool which performs various checks on parallel programs written in CSP in order to prove freedom from deadlock and livelock [25]. It acts as a design workbench for designers of safety-critical distributed systems. The approach is efficient because of certain simplifications and is thus incomplete.

Joël Ouaknine's *Slap* [18] is a conservative livelock checker for processes written in a subset of CSP. If Slap outputs "LIVELOCK-FREE" then the CSP model is livelock free. If the tool outputs "POTENTIAL LIVELOCK" then no definite conclusion can be made. Slap is currently in beta stadium and its source is available.

Failures-Divergence Refinement (FDR) [23, 28] is a model-checking tool which enables CSP based proofs and analysis. FDR uses the so called "failures-divergences model" of CSP for detection of a set of traces after which a process may livelock. At present it is limited to analyzing systems of up to one hundred million global states.

Communicating Sequential Processes for Java (JCSP) [38], *Communicating Threads for Java* (CTJ) [13], and *Java2CSP* [31] can be used to enable a CSP-like verification of Java programs. Also C++CSP and JCSP.net for C++ and .net, respectively, are available.

An example using CSP for detecting livelocks in real applications is the work of Buth [6, 7]. occam code from the International Space Station (ISS) is being converted to CSP using abstraction techniques. The resulting CSP model is checked using FDR. Nevertheless the presented technique needs some manual steps and knowledge. Thus it is far from being fully automated.

5 Conclusion

We have presented simple algorithms for detecting livelocks in Ada multitasking programs. Although their worst-case execution time is exponential, the algorithms can be expected to run in time $O(|V|^2)$, where $|V|$ denotes the number of nodes of the CFG of the analyzed program.

Since the problem is strongly connected to finding infinite loops, which is undecidable, our algorithms compute only an approximation to the real solution of the problem. As a consequence our algorithms compute false positives.

A future direction of our work will be to include data-flow in our approach which might result in a more fine-grained analysis.

Another future direction of our work will be to set up a symbolic framework (e.g., based on [5]) for livelock detection. Such a framework can be expected to use more resources (time and space) than the approach presented in this paper, but will be less amenable to false positives.

References

1. P. A. Abdulla, B. Jonsson, A. Rezine, and M. Saksena. Proving Liveness by Backwards Reachability. In *CONCUR, LNCS*, volume 4137, pages 95–109, Bonn, Germany, August 2006.

2. B. Burgstaller. Symbolic Evaluation of Imperative Programming Languages. Technical Report 183/1-138, Department of Automation, Vienna University of Technology, June 2005. Available at `http://www.auto.tuwien.ac.at/~bburg/reports.html`.
3. B. Burgstaller, J. Blieberger, and R. Mittermayr. Static detection of access anomalies in Ada95. In *Proc. Ada-Europe'2006, LNCS 4006*, pages 40–55, Porto, Portugal, June 2006.
4. B. Burgstaller, B. Scholz, and J. Blieberger. Tour de Spec — A Collection of Spec95 Program Paths and Associated Costs for Symbolic Evaluation. Technical Report 183/1-137, Department of Automation, Vienna University of Technology, June 2004. Available at `http://www.auto.tuwien.ac.at/~bburg/reports.html`.
5. B. Burgstaller, B. Scholz, and J. Blieberger. Symbolic Analysis of Imperative Programming Languages. In *Proceedings of the 7th Joint Modular Languages Conference*, Springer LNCS, pages 172–194, 2006.
6. B. Buth, J. Peleska, and H. Shi. Combining Methods for the Livelock Analysis of a Fault-Tolerant System. In A. M. H. (Ed.), editor, *AMAST '98: Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, volume 1548, pages 124–139, London, UK, 1998. Springer-Verlag.
7. B. Buth, J. Peleska, and H. Shi. *Deadlock- und Livelock-Analyse für die Internationale Raumstation ISS*, August 2006.
8. T. Cassidy, J. Cordy, T. Dean, and J. Dingel. Source Transformation for Concurrency Analysis. In *Proc. LDTA 2005, ACM 5th International Workshop on Language Descriptions, Tools and Applications*, pages 26–43, Edinburgh, Scotland, April 2005.
9. J. Cheng and K. Ushijima. Analyzing Ada Tasking Deadlocks and Livelocks Using Extended Petri Nets. In *Ada: The Choice for '92, Proc. of the Ada-Europe International Conference*, volume 499, pages 125–146, Athens, Greece, May 1991.
10. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000.
11. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1989.
12. P. Godefroid. Software Model Checking: The VeriSoft Approach. *Form. Methods Syst. Des.*, 26(2):77–101, 2005.
13. G. H. Hilderink, J. F. Broenink, and A. W. P. Bakkers. Communicating Threads for Java. In B. M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–261, Keele, UK, March 1999.
14. A. Ho, S. Smith, and S. Hand. On deadlock, livelock, and forward progress. Technical Report UCAM-CL-TR-633, University of Cambridge, Computer Laboratory, May 2005.
15. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, 1985.
16. G. J. Holzmann. Proving Properties of Concurrent Systems with Spin. In *Proc. CONCUR95, 6th Intern. Conf. on Concurrency Theory*, Philadelphia, PA., August 1995. (invited tutorial).
17. G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
18. Joël Ouaknine. SLAP tool (Version 0.1): A Static Livelock Analyzer for CSP Processes,
`http://web.comlab.ox.ac.uk/oucl/work/joel.ouaknine/software/slap.html`.

19. G. M. Karam and R. J. A. Buhr. Starvation and Critical Race Analyzers for Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, 1990.
20. Y. S. Kwong. *On reductions and livelocks in asynchronous parallel computation*. PhD thesis, 1978.
21. Y. S. Kwong. On the Absence of Livelocks in Parallel Programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation, LNCS*, volume 70, pages 172–190, London, UK, 1979. Springer-Verlag.
22. S. Leue, A. Ştefănescu, and W. Wei. A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems. In C. Baier and H. Hermanns, editors, *CONCUR, LNCS*, volume 4137, pages 79–94, Bonn, Germany, August 2006.
23. F. S. E. Ltd. *Failures-Divergence Refinement, FDR 2 User Manual*. www.fsel.com, 2005.
24. J. M. R. Martin and Y. Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. In P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 1–14, September 2000.
25. J. M. R. Martin and S. A. Jassim. A Tool for Proving Deadlock Freedom. In *Proc. WoTUG20: Parallel Programming and Java*, pages 1–16. IOS Press, April 1997.
26. J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.
27. S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
28. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
29. K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, New York, 3rd edition, 1995.
30. B. Scholz and J. Blieberger. A New Elimination-Based Data Flow Analysis Framework using Annotated Decomposition Trees. In *Proc. ETAPS 2007, LNCS*, Braga, Portugal, March 2007.
31. H. Shi. Java2CSP: A System for Verifying Concurrent Java Programs. In G. Schellhorn and W. Reif, editors, *FM-TOOLS 2000*, number 2000-07 in Ulmer Informatik-Berichte, pages 111–115, 2000.
32. W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Englewood Cliffs, NJ, 2001.
33. K.-C. Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *ICPP*, volume 2, pages 69–72, August 1994.
34. R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.
35. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.
36. R. J. van Glabbeek and M. Voorhoeve. Liveness, Fairness and Impossible Futures. In *CONCUR, LNCS*, volume 4137, pages 126–141, Bonn, Germany, August 2006.
37. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *In IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.
38. P. H. Welch and J. M. R. Martin. A CSP Model for Java Multithreading. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 114, Washington, DC, USA, 2000. IEEE Computer Society.