Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes

Stefan Leue[1], Alin Ştefănescu[2]*, and Wei Wei[1]

[1] Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Email: {Stefan.Leue|Wei.Wei}@uni-konstanz.de
[2] SAP Research CEC Darmstadt
Bleichstr. 8, D-64283 Darmstadt, Germany
Email: alin.stefanescu@sap.com

Abstract. The execution of a reactive system amounts to the repetitions of executions of control flow cycles in the component processes of the system. The way in which cycle executions are combined is not arbitrary since cycles may depend on or exclude one another. We believe that the information of such dependencies is important to the design, understanding, and verification of reactive systems. In this paper, we formally define the concept of a *cycle dependency*, and propose several static analysis methods to discover such dependencies. We have implemented several strategies for computing cycle dependencies and compared their performance with realistic models of considerable size. It is also shown how the detection of accurate dependencies is used to improve a livelock freedom analysis that we developed previously.

1 Introduction

The main purpose of a concurrent reactive system is to maintain an ongoing interaction with its environment [14]. The execution of the system is therefore expected to last forever. Since each component process in the system has a finite control structure, any infinite execution of the system is essentially an infinite repetition of a certain set of control flow cycles in the concurrent processes that form the system. The way in which cycle executions are combined is certainly *not* arbitrary. For instance, the repetition of one cycle may rely on the repetitions of some other cycles; and the execution of one cycle may also eliminate the possibility of executing some other cycles.

We believe that the information of such cycle dependencies is important to the design, understanding, and verification of reactive systems. As one example, the knowledge of cycle dependencies may reveal potential design errors. In a reactive system, let us suppose that the repetition of a control flow cycle C relies on the repetitions of other cycles. When we expect C to be executed infinitely

* The work was done while this author was working at the University of Konstanz.

often, we may want to check statically whether there is any other cycle in the system on which C relies. The fact that no such cycles can actually be found hints at an incompleteness in the design or implementation of the system.

The knowledge of cycle dependencies is also useful in the verification of concurrent reactive systems. In our precursory work, we proposed an efficient system verification framework based on integer linear program (ILP) solving [12, 11]. Our approach does not count on the exhaustive exploration of a potentially exponential or even infinite number of interleavings of actions from concurrent processes, and thus avoids the notorious state explosion problem from which traditional explicit state model checking techniques suffer. Our verification methods abstract the original verification problem into an ILP problem that describes a necessary condition for the violation of the property under scrutiny. Any solution to the ILP problem corresponds to a counterexample in the form of a set of cycles. A counterexample is spurious if it is impossible to repeat the cycles in the counterexample forever without other cycles also being repeated infinitely often. Consequently, the dependency among cycles stands at the very core of the refinement procedure based on the detection of spurious counterexamples in [13, 11].

The central contribution of this paper is a formal framework capturing a notion of dependency between the control flow cycles of the concurrent processes. We also inspect different causes of dependencies, and develop techniques for discovering dependencies with respect to each cause. In this paper we choose Promela [8] as modeling language for the systems that we analyze. This choice is motivated by convenience since a large number of Promela models are available in the public domain [20] and some of the features of the SPIN tool environment, which interprets Promela, greatly facilitate our static analysis. We conjecture that applying our analysis ideas to other modeling and programming languages based on communicating finite state machines, such as UML-RT, could easily be accomplished.

Related Work. To the best of our knowledge, there is currently no work addressing control flow cycle dependencies. Control flow graphs of general programs were extensively studied in the area of static program analysis [19] with applications, e.g., in the area of compiler optimization. Slicing of programs [23, 17] checks dependences between statements but not cycles. The "may happen in parallel" [18] and "non-concurrency" [15] analyses also consider dependences between statements. Finally, the INCA verification framework [4, 22] studies the relation between acyclic paths and control flow cycles but not relations among cycles. Moreover, the above techniques are applied to synchronous communication settings, while we also address an asynchronous setting where exchanging messages via buffers is the dominant way of communication.

Structure of the Paper. Section 2 introduces the Promela modeling language, define cycles and some related concepts. Section 3 defines the concept of cycle dependencies. We propose in Sections 4 and 5 several static analysis methods for cycle dependency discovery. Section 6 briefly shows how the discovery of cycle dependencies can help improve the precision of a livelock freedom test. Section 7 reports the experimental results, before Section 8 concludes the paper. All the proofs of the theoretical results of the paper can be found in Appendix A.

2 Preliminaries

Promela. Promela is the input language of the SPIN explicit state model checker [8]. It has been successfully used for the modeling and analysis of many concurrent systems [9, 6]. The Promela language supports asynchronous communication as well as synchronous rendez-vous communication and synchronization via shared variables. The subset of the Promela language that we consider includes the definition of concurrently running processes ("`proctype`"), communication channels ("`chan`" declarations), message sending ("`!`") and receiving ("`?`"), assignments, condition statements, nondeterministic branching ("`if ... fi`"), looping ("`do ... od`"), and arithmetics. For the sake of simplicity we do not consider arrays and structured data types in this paper.

```
1   active proctype client()
2     int x = 0;
3     do
4     :: (x < 3) -> toServer!request; x++;
5     :: (x == 3) ->
6          fromServer?reply; x--;
7     od
8
9   active proctype server()
10    do
11    :: toServer?request -> fromServer!reply;
12    od
```

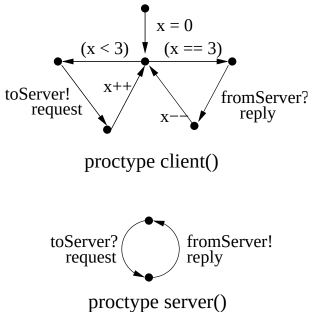


Fig. 1. An example Promela model and its control flow graphs.

Figure 1 shows a simple Promela model consisting of two processes: `client` and `server`, whose behavior is described by the sequential Promela code within the respective `proctype` definition. The process `client` may send a `request` message to the buffer `toServer` if $x < 3$ (Line 4). Otherwise, it waits until a `reply` message is available in the buffer `fromServer` and then receives the message (Line 6). A condition statement such as `(x < 3)` is a boolean expression enclosed in parentheses, which acts as a guard to the subsequent statements. It is executable if and only if the enclosed expression evaluates to true. We can construct a control flow graph from each of the `proctype` definitions (see Figure 1). Each transition corresponds to one statement in the code, and its *source* state and *target* state respectively denote the control points before and after the execution of the statement.

Control flow cycles. We define a *control flow cycle* (or simply *cycle*) in a control flow graph as a sequence of consecutive transitions in the graph such that the source state of the first transition in the sequence is the same as the target state of the last transition. A cycle is *elementary* (or *simple*) if no two transitions in the defining sequence of the cycle have a same source state. Informally, an elementary cycle cannot be decomposed further into smaller cycles. In the control flow graph of the process `client` in Figure 1, there are two elementary cycles.

Even though a finite control flow graph may contain infinitely many cycles, the number of elementary cycles is always finite and in the worst case exponential in the number of transitions. Since any non-elementary cycle can be decomposed into elementary cycles, our analysis considers only elementary cycles. Unless otherwise specified, all the cycles mentioned in the following are elementary.

If two cycles share states, then they are *neighbors* of each other. Any such shared state is an *exit state* of the cycles that contain it, because one can exit one cycle at that state and enter another cycle. The two cycles in the process `client` in Figure 1 are neighbors sharing one exit state.

Cycle executions. An infinite run of a Promela model amounts to the repeated executions of cycles in some processes of the model. For an infinite run r of a Promela model, let $r/_p$ denote the projection of r on the set of transitions in a process p. Thus, $r/_p$ corresponds to the local execution of p in r. Any $r/_p$ can be decomposed into two parts: (1) an acyclic path from the initial state, and (2) repeated executions of cycles. Given a cycle c in p, one execution of c in $r/_p$ may be interrupted by the executions of other cycles in p: Some part of c is executed until some exit state s is reached where it starts to execute other cycles. The execution of c is later resumed from s after the executions of those interrupting cycles are completed. Since r is an infinite run, at least one cycle in the model is repeated infinitely often. We denote by $\mathit{IRC}(r)$ (*infinitely repeated cycles*) the set of cycles that are executed infinitely often in r. For a process p, $\mathit{IRC}(r/_p)$ is the subset of $\mathit{IRC}(r)$ consisting of only cycles in p. It is easy to see that $\mathit{IRC}(r/_p)$ is either empty or forms a strongly connected subgraph of the control flow graph of p.

3 Cycle Dependencies

We now define the concept of cycle dependencies. Intuitively, a cycle c depends on a set of cycles S if the infinite execution of c must be accompanied by the infinite executions of some cycles in S.

Definition 1. *Given a Promela model, a cycle c and a set of cycles S in the model, we call the pair (c, S) a* cycle dependency *if they satisfy the following conditions: a) $c \notin S$; and b) for any infinite run r of the model where $c \in \mathit{IRC}(r)$, there exists a cycle $c' \in S$ such that $c' \in \mathit{IRC}(r)$. In this case, we say that c* depends on *S.*

In the above definition, if all the cycles in S are in the same process as c is, then (c, S) is a *local* dependency. Otherwise, (c, S) is a *global* dependency. Moreover, if c does not depend on any subset of S, then we say that (c, S) is a *minimal* dependency. In the model in Figure 1, we denote by c_l (resp. c_r) the left (resp. right) cycle in the process `client` and by c_s the only cycle in the process `server`. $(c_r, \{c_l, c_s\})$ is a cycle dependency, while $(c_r, \{c_l\})$ and $(c_r, \{c_s\})$ are two minimal cycle dependencies. In particular, $(c_r, \{c_l\})$ is a local dependency, and $(c_r, \{c_s\})$ is a global dependency.

If we interpret all message buffers in a Promela model to have only finite capacities, then the Promela model possesses a finite global state space. In this case, we show as follows that it is decidable whether (c, S) is a cycle dependency: We construct the global state space for the model and then look for any elementary of non-elementary cycle in the global state space that contains c but no cycles from S. If no such global cycles exist, then (c, S) is a cycle dependency. However, we are more interested in infinite state models. If we assume that buffers in Promela models have infinite capacities and variables may have infinite domains such as integer variables, then a Promela model may have an infinite global state space, for which we show in the following theorem that the above problem becomes undecidable.

Theorem 1. *Given a cycle c and a set S of cycles, it is undecidable in general whether (c, S) is a cycle dependency.*

3.1 The Causes of Cycle Dependencies

The root cause for cycle dependencies lies in the executability of Promela statements. Given a cycle, if the executability of every statement along the cycle is unconditional, then the cycle can be repeated without interruption forever once the cycle is entered. Such a cycle does not depend on any other cycles. On the contrary, consider a cycle c that contains a statement s whose executability is conditional. If s cannot be continuously enabled forever by only repeating c, then some other cycles need to be executed in order to re-enable s by, e.g., modifying the values of some variables, sending a message etc. In Promela there are two kinds of statements with conditional executability: condition statements and message receiving statements, when we take the assumption that message buffers have unbounded capacities and message sending statements are therefore always enabled. In the following we explain how cycle dependencies may be imposed by these two kinds of statements.

Condition statements. Consider the right cycle c_r in the process `client` in Figure 1. c_r contains a condition statement `(x == 3)`. The condition $x = 3$ cannot remain true after c_r is executed because x is decremented by 1 in the cycle. Then, c_r can be repeated infinitely often only if the left cycle c_l is also repeated infinitely often to modify the value of x such that x can always acquire the value 3 again. This is one example that a cycle is terminating on a condition statement along the cycle. Since we focus on discovering cycle dependencies in this paper, it is out of scope how to determine whether a cycle is terminating, which is a well-known undecidable problem. In [13] we proposed an incomplete procedure to prove termination for control flow cycles. There are also many existing techniques [21, 3, 5, 1] to prove termination for certain kinds of loops in programs, which can be adapted to prove termination for control flow cycles. In Section 4 we will show how to determine cycle dependencies from a condition statement on which a cycle is terminating.

Message receiving statements. The above mentioned cycle c_r contains a message receiving statement `fromServer?reply`. Thus, the cycle c_s sending `reply` messages has to be repeated infinitely often when c_r is to be repeated infinitely often. In Section 5 we will present a method to determine cycle dependencies from message receiving statements, which are usually global dependencies.

4 Discovering Dependencies from Condition Statements

We show some types of cycle dependencies imposed by condition statements on which a cycle is terminating. In order to derive them, we need to discriminate between different ways in which the variables in a condition statement are modified in the cycle. A variable is *local* if its value can be referenced and modified only by one process. Otherwise, it is a *global* variable. However, the runtime value of a local variable may still depend on the executions of other processes. For instance, given a local variable x, if there is an assignment `x =` $e(y)$ where e is an arithmetic expression containing a global variable y, then the runtime value of x may depend on how y is modified in other processes.

Definition 2. *For a cycle c and a variable x, x is* globally modified *in c if one of the following is satisfied: a) x is global, or b) there is a message receiving statement* `b?msg(`x_1`,...,`x_n`)` *in c where x is some x_i, or c) there is an assignment* `x =` $e(y)$ *in c where y is globally modified in c. Otherwise, x is* locally modified *in c.*

Note that in the above definition we disregard the dependency of the runtime value of a local variable on a condition statement. The reason is that a control flow cycle contains only one branch of a condition statement. Therefore, the impact of the condition statement is fixed in the cycle. Note that we are only interested in how a variable is modified inside a particular cycle when the cycle is repeated without interruption.

For a boolean condition B in a cycle c, we denote by $var(B)$ the set of variables occurring in B. If all the variables in $var(B)$ are locally modified in c, then B is a *locally determined* condition. Otherwise, it is *globally determined*. In Subsection 4.1 and 4.2, we show how to determine cycle dependencies from these two kinds of conditions.

4.1 Locally Determined Conditions

First, we can easily see that, if a cycle is terminating on a locally determined condition, then it depends on some of the cycles in the same process for an infinite number of executions. In particular, the cycle must depend on one of its neighbors.

Proposition 1. *Given a cycle c in a process p such that c is terminating on a locally determined condition B, if c is repeated infinitely often in a run r, then one of the neighbors of c is also repeated infinitely often in r.*

Let C_p denote the set of cycles in p, and N_c denote the set of the neighbors of c. The above discussion gives two cycle dependencies, namely $(c, C_p - \{c\})$ and (c, N_c). The cycle $(c, C_p - \{c\})$ is usually coarse because not necessarily all the cycles in p contribute to the re-satisfaction of B. In the following, we propose several methods to refine the dependency $(c, C_p - \{c\})$.

Refinement 1. In general, it is impossible to determine which cycles make a contribution to the re-satisfaction of the condition B. We define $E_c(B)$ as the set of variables occurring in c such that at least one of the variables in $E_c(B)$ must be modified in order to make B true again. The set $E_c(B)$ subsumes but not necessarily equals $var(B)$. In the example in Figure 2, an infinite number of repetitions of the left cycle relies on an infinite number of repetitions of the right one that resets the value of y. However, the enabling condition of the left cycle contains only the variable x which is not modified by the right cycle. We propose the following recursive method to compute $E_c(B)$. A variable v is in $E_c(B)$ if one of the following is satisfied: a) $v \in var(B)$, or b) there is an assignment $v' = e(v)$ in c such that $v' \in E_c(B)$. For a set S of variables, we denote by $MC_p(S)$ the set of cycles in p which modify at least one variable in S. We obtain a finer dependency $(c, MC_p(E_c(B)) - \{c\})$ by disregarding all cycles that do not modify any variables in $E_c(B)$.

```
active proctype p()
  int x = 5; int y = 5;
  do
  :: (x > 0) -> y--; x = y;
  :: y = 5;
  od
```

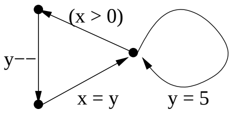


Fig. 2. An example Promela model and its control flow graph.

Refinement 2. The above cycle dependency may still be coarse. Consider the control flow graph in Figure 3. Note that, whenever leaving $C1$ to execute $C3$, $C2$ is always executed. So, in any run in which $C1$ is repeated infinitely often, no matter whether $C3$ is repeated infinitely often or not, $C2$ is always repeated infinitely often. Based on this observation we can refine the cycle dependency $(C1, \{C2, C3\})$ by safely removing $C3$. The above simple example leads to the following definition.

Definition 3. *Given a cycle c in a process p such that c is terminating on a locally determined condition B, and a cycle $c' \in MC_p(E_c(B))$ such that c and c' are reachable from each other, c' is* preemptive *with respect to c and B if there exist one exit state s in c and one exit state s' in c' such that a) there is an*

```
active proctype p()
  int x;                     x = 5;
  do                         do
  :: x = 5;                  :: (x > 0) -> x--;
     do                      :: break;
     :: (x > 0) -> x--;      od;
     :: break;            od
     od;
```

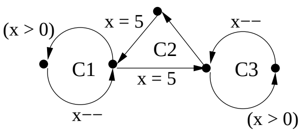


Fig. 3. An example Promela Model and its control flow graph.

acyclic path from s to s' that does not modify any variables in $E_c(B)$, and b) there is an acyclic path from s' to s that does not modify any variables in $E_c(B)$. Otherwise, c' is preempted.

In the previous example, $C2$ is preemptive and $C3$ is preempted. It is easy to prove that, on the way from any cycle c to execute one of its preempted cycles and then back to c, at least one preemptive cycle must be executed. We can therefore refine the cycle dependency $(c, MC_p(E_c(B)) - \{c\})$ by removing all the preempted cycles from $(MC_p(E_c(B)) - \{c\})$.

```
proc compute_cd(cycle c_0, condition B_0)
  set[cycle] visited = {}
  set[cycle] ccs = {}
  queue[cycle] open = {}
  search_for_preemptive_cycles(c_0, B_0)
  return (c_0, ccs) // return the determined cycle dependency

proc search_for_preemptive_cycles(cycle c, condition B)
  add c to visited

  for each nc in neighbors(c)
    if (nc not in visited) and (nc not in open)
      if (nc modifies some variables in E_c(B))
      then
        add nc to visited
        add nc to ccs
      else
        enqueue(open, nc)

  if (open not empty)
    c’ = dequeue(open)
    search_for_preemptive_cycles(c’, B)
```

Fig. 4. An algorithm to determine cycle dependencies from locally determined conditions.

Whereas Definition 3 can be used to determine whether a cycle is preempted, Figure 4 gives an efficient algorithm to collect preemptive cycles during the computation of cycle dependencies. In a Breadth First Search manner, the algorithm visits each cycle at most once, and thus is linear in the number of cycles. This is a generalization of the so-called “next door” strategy first mentioned in [11]. In Appendix A.3 the termination and soundness of the algorithm are proved.

4.2 Globally Determined Conditions

If a cycle is terminating on a globally determined condition, then it may not only depend on cycles in the same process, because cycles in other concurrent processes can possibly influence the runtime value of the condition. This can be illustrated in the example in Figure 5. The cycle in Process p is actually the only cycle in p, and it depends on the cycle in q. We will not consider any globally determined condition whose value is influenced by a message receiving statement, which will be discussed in the next section.

```
int y = 5;

active proctype p()
  int x = 5;
  do
  :: (x > 0) -> y--; x = y;
  od

active proctype q()
  do
  :: y = 5;
  od
```

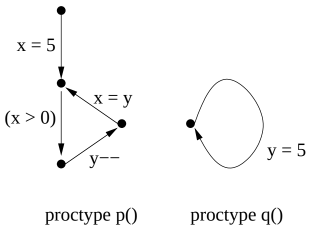


Fig. 5. An example Promela model and its control flow graphs.

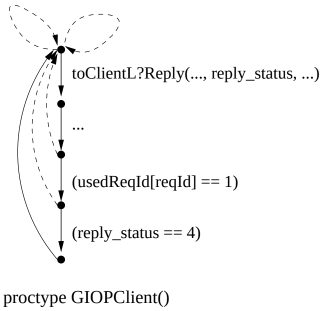
For a Promela model M, we denote by $proc(M)$ the set of processes in M. Suppose a cycle c in a process in M such that c is terminating on a globally determined condition B. We can easily derive that c depends on $(\bigcup_{q \in proc(M)} MC_q(E_c(B)) - \{c\})$. We may refine this cycle dependency by using the algorithm in Figure 4 to rule out all the preempted cycles in $MC_p(E_c(B))$ if c is in the process p.

5 Discovering Dependencies from Message Receiving Statements

When a cycle c contains a message receiving statement `b?msg(`$x_1, \ldots, x_n$`)`, it needs an infinite number of `msg` messages to be repeated infinitely often. Consequently, c depends on some cycles that send such messages. Let $SC_{b,msg}$ be the set of the cycles sending messages `msg` to b. If $c \notin SC_{b,msg}$, then $(c, SC_{b,msg})$ is cycle dependency. In the remainder of the section, we assume that a cycle never receives messages sent by itself.

A cycle that receives messages may contain a condition statement in which the condition contains some variables used to store components of received messages. Usually, the cycle can be executed only if the received message contains such components that make the condition true. Consider a cycle that contains a message receiving statement s_1 and a condition statement s_2 such that the condition in s_2 contains variables used in s_1. The following pattern for s_1 and s_2 are observed in most real life Promela models: (1) all the variables in s_1 are local; (2) the condition in s_2 contains only variables used in s_1; (3) no variable in the condition is modified between s_1 and s_2 in the cycle. We call such a condition a *message determined* condition. Figure 6 shows two processes GIOPClient and GIOPAgent. In the control flow graph of GIOPClient, there is a cycle depicted using only solid lines that contains a message determined condition *reply_status* $= 4$. We show in the remainder of the section how to derive cycle dependencies from such a message determined condition.

```
active proctype GIOPClient()
  ...
  do
  :: toClientL?Reply(..., reply_status, ...) ->
       ... // replay_status is not modified here
       if
       :: (usedReqId[reqId] == 1) ->
          if
          :: (reply_status == 4) -> ...
          ...
          fi;
       ...
       fi;
  ...
  od
```



```
active proctype GIOPAgent()
  ...
  do
  :: toAgent?Request(...) ->
     ...
     if
     :: (registered[objKey] == 255) ->
        ...
        rs = 5;
     :: else ->
        ...
        rs = 4;
     fi;
     toClientL!Reply(..., rs, ...);
  ...
  od
```

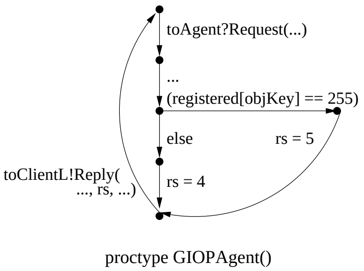


Fig. 6. An excerpt from a Promela model for CORBA GIOP [9].

In Figure 6, let c_1 denote the solid-lined cycle in Process GIOPClient, and c_2 and c_3 denote the cycles that respectively assign 4 and 5 to *rs* in Process GIOPAgent. We have a dependency $(c_1, SC_{toClientL,Reply})$ and both c_2 and c_3 are in $SC_{toClientL,Reply}$. However, this dependency is coarse because not necessarily every cycle in $SC_{toClientL,Reply}$ may send a Reply message to make *reply_status* $=$ 4 true in c_1. As an example, c_3 assigns 5 to *rs* whose value is passed to *reply_status* in c_1 through message passing. Thus, c_3 cannot make *reply_status* $= 4$ true, and it can be safely removed from $SC_{toClientL,Reply}$ to obtain a finer dependency. Now the question is how to determine which cycle *cannot* send messages to make *reply_status* $= 4$ true.

First, we need to determine which kind of Reply messages must be received by c_1 to make *reply_status* $= 4$ true. More precisely, we need to know which condition must be satisfied by the components of such a message. According to the definition of message determined conditions, *reply_status* is not modified in c_1 between the message receiving statement and `(reply_status == 4)`. However, after a message is received, *reply_status* may still be modified before `(reply_status == 4)` is reached. This is because the execution of c_1 can be interrupted, e.g., at the source state of the transition corresponding to `(usedReqId[reqId] == 1)`. Then, when the execution of c_1 is resumed, *reply_status* may be already modified by other cycles. However, in this concrete example, if c_1 is interrupted, then before c_1 is resumed the last completed interrupting cycle always receives a `Reply` message. Moreover, this message contains a component whose value is passed to *reply_status*. The value of *reply_status* is afterward unchanged before reaching the message determined condition. This is because c_1 and its neighbors satisfy the following structural property named *fastened cycles*: Given a cycle c that contains a message receiving statement s_1 and a condition statement s_2, we denote by t_1 the transition corresponding to s_1, by t_2 the transition corresponding to s_2, and by p the path from the source state of t_1 to the source state of t_2. For each neighbor c' of c, if c' and c contain a common state s within p, then c' contains also the path in c from the source state of t_1 to s. The pattern in the fastened cycles property results from nested `if` statements inside `do` loops which are a common control structure of concurrent processes in an asynchronous reactive system.

Proposition 2. *Let c be a cycle that contains a condition statement* `(`B`)` *in which the condition B is determined by messages received via the statement* `b?msg`$(x_1, \ldots, x_n)$ *in c. If the fastened cycles property is satisfied by c and all of its neighbors, then one execution of c needs a* `msg`$(d_1, \ldots, d_n)$ *message such that* $B[x_i \leftarrow d_i]$[3] *is true.*

Using Proposition 2, if we can determine that the execution of c requires a message `msg(`$d_1, \ldots, d_n$`)` such that $B[x_i \leftarrow d_i]$ is true, then we can use the following method to determine whether a cycle c' may *not* send such a message. Given a cycle c' that contains a message sending statement `b!msg(`$d_1, \ldots, d_n$`)`, if all d_i's are constant values, then we directly evaluate $B[x_i \leftarrow d_i]$ which is a constant truth value. If it is false, then we can exclude c' from $SC_{b,msg}$. When some d_i is a variable, we traverse backward in c' from the source state s' of the transition t corresponding to `b!msg(`$d_1, \ldots, d_n$`)`, and locate the first state $s \neq s'$ such that s has an incoming transition outside c' but within other cycles. If no such s exists, then we take as s the predecessor of s' in c'. The path p from s to s' is then the longest acyclic path within c that must be consecutively executed immediately before reaching the message sending statement. We compute the postcondition *Post(p)* of p by Floyd-Hoare-style forward inference starting with the precondition `true`[4]. This assumes that all the variables initially contain arbitrary values before p is consecutively executed. If $Post(p) \wedge B[x_i \leftarrow d_i]$ is unsatisfiable, then c' can be removed from $SC_{b,msg}$. If the Promela model contains

[3] $B[x_i \leftarrow d_i]$ is a boolean expression obtained from B by substituting simultaneously each occurrence of x_i with d_i.

[4] Since the path p is acyclic, *Post(p)* can be computed fully automatically.

only linear arithmetic expressions in assignments and conditions, then the satisfiability of $Post(p) \wedge B[x_i \leftarrow d_i]$ can be decided fully automatically using either an automated theorem prover or a linear programming solver. In the example in Figure 6, we illustrate how to determine that c_3 cannot send a message to satisfy $reply\_status = 4$. The longest consecutively executed path p in this example starts from the source state of the transition corresponding to the message receiving statement, i.e., the topmost state in the control flow graph of `GIOPAgent`. Then $Post(p) = (\cdots \wedge (rs = 5))$. Since $Post(p) \wedge (reply\_status = 4)[reply\_status \leftarrow rs]$ is false, c_3 can be safely removed from $(c_1, SC_{toClientL,Reply})$.

6 The Refinement of a Livelock Freedom Test

We show how the discovery of cycle dependencies can be used to improve the precision of a livelock freedom test that we developed [11]. We sketch this test using the example in Figure 1.

In a Promela model we may label a set of statements as *progress* statements. Let us assume the message receiving statement (Line 6) in the process `client` is the only progress statement in Figure 1. A model is said to be *free of livelock* if and only if at least one of the progress statements must be repeated infinitely often in any infinite run of the model. Therefore, our example model is free of livelock if the client always receives replies from the server infinitely often. Moreover, we define a cycle to be a *progress cycle* if it contains at least one progress statement. So, the right cycle c_r of `client` is the only progress cycle. We have shown in [11] that livelock freedom is undecidable for infinite state systems.

The basic idea of our livelock freedom test in [11] is to check whether there is any infinite run of a model in which no progress cycle is repeated infinitely often. If no such run exists, then the model is livelock free. In our test, we first abstract from arbitrary program code in the model and retain only the message sending and receiving statements. Next, we abstract from message orders and denote the message passing effect of a statement by an integer vector called an *effect vector*. Each component of an effect vector corresponds to one type of messages. A positive component represents the number of messages of the corresponding type being sent by the statement. A negative component represents the number of messages being received. We abstract further from the activation conditions and dependencies of cycles. The resulting abstract system is a set of cycles with their summary effect vectors. In our example, there are three cycles: c_l with the effect vector $(1, 0)$, c_r with $(0, -1)$, and c_s with $(-1, 1)$.

We now give a necessary condition for the existence of a livelocked run, i.e., a run in which no progress cycle is repeated infinitely often, in the form of an integer linear programming (ILP) problem. The ILP problem is shown in Figure 7. It can be solved in polynomial time. Intuitively, any solution to this ILP problem represents a combination of cycle effects that (1) can be repeated forever since it does not consume any type of messages (Inequalities 1–2); and (2) does not include any progress cycle (Inequality 3). The last inequality 4 restricts the number of times that a cycle is repeated to be non-negative. If the ILP problem has no solutions, then such cycle combination does not exist, which proves livelock freedom for the model. Unfortunately, the ILP problem has a solution: $x_1 = 1, x_2 = x_3 = 0$. In this case, we do not know whether the model is livelock free or not because the abstraction used in our test is over-approximating and may introduce spurious behavior.

$$\begin{aligned}
\begin{pmatrix}1\\0\end{pmatrix} x_1 + \begin{pmatrix}0\\-1\end{pmatrix} x_2 + \begin{pmatrix}-1\\1\end{pmatrix} x_3 \geq \begin{pmatrix}0\\0\end{pmatrix} && (1)\\
x_1 + x_2 + x_3 > 0 && (2)\\
x_1 = 0 && (3)\\
x_i \geq 0 && (4)
\end{aligned}$$

Fig. 7. The livelock freedom determination ILP problem for the model in Figure 1.

The above obtained ILP solution represents a counterexample suggesting the scenario that only the cycle c_l is repeated infinitely often in some runs. However, by the help of our cycle dependency discovery, we can see that the cycle c_l depends on c_r. Since c_r is not included in the counterexample, the counterexample is spurious. Furthermore, we can use the cycle dependency to refine the abstraction by adding the following inequality to the ILP problem in Figure 7:

$$x_1 \leq 3x_2 \qquad (5)$$

Intuitively, the new constraint says that the cycle c_r must be executed at least once for every 3 times that c_l is repeated. The determination of this constraint also relies on the estimation of the maximal iteration counts of the cycle c_l, which we discussed in [13].

The quality of the above described refinement procedure largely relies on the accuracy of the cycle dependency discovery techniques. The smaller a detected cycle dependency is, the more spurious behavior can be excluded through refinement.

7 Experimental Results

We have implemented different strategies to detect both local and global cycle dependencies for the models[5] listed in Table 1. The experimental results were obtained on a Pentium IV 1.60GHz machine with 1GB memory.

[5] *MVCC* [7] models the *Model View and Concurrent Control* protocol used in the *Clock* toolkit for the development of groupware applications; *i-Protocol* [6] models a

Model	# cycles	# detected cycle dependencies
i-Protocol	22	30
MVCC	30	51
GIOP	66	203
SMCS	171	541

Table 1. Test models.

We detected three types of dependencies: (1) dependencies on neighbors (see Corollary 1); (2) dependencies on cycles that may render the considered condition to be re-satisfied; (3) dependencies caused by message receiving statements. Table 1 lists the total number of dependencies of all three types that were detected for each model. Different strategies are used to over-approximate dependencies of Type 2 and 3, and their performances are compared as explained below.

	summary size of dependencies			% reduction w.r.t. MC		runtime (secs.)		
Model	MC	ND	PC	ND	PC	MC	ND	PC
i-Protocol	63	63	43	0	31.7	13.02	13.27	13.66
MVCC	60	59	41	1.7	31.7	3.55	3.59	3.52
GIOP	837	788	714	5.9	14.7	29.25	30.12	33.17
SMCS	5200	5200	3424	0	34.2	136.63	143.75	175.05

Table 2. The comparison of different strategies to detect dependencies of Type 2. By the size of a cycle dependency (c, S), we refer to the size of the set S. For each model and each strategy, we list the sum of the sizes of all the detected dependencies of this type.

Table 2 compares three different strategies for the detection of dependencies of type 2: *MC* is the coarsest one that includes a cycle in the dependency as long as it may influence at least one variable in the considered condition; *ND* is the next-door strategy; *PC* is the finest one that collects only preemptive cycles for computing dependencies. We observe that ND leads to only a minor improvement of the accuracy of the detected dependencies. PC reduces the sizes of dependencies much more effectively at the expense of a modest or even no runtime penalty (see the results for MVCC[6]). In particular, PC reduces the sizes

sliding-window protocol for *Unix-to-Unix-Copy*; *GIOP* [9] models inter-*ORB* message exchange and server object migration in the *CORBA* architecture; *SMCS* [16] models the *T.122* and *T.125* multi-point communication service protocol.

[6] The reason is that ND and PC sometimes check only a small number of cycles for computing dependencies for one cycle while MC has to check all the cycles in the same process.

of dependencies by more than one third for the model SMCS while ND does not reduce the cycle number at all.

	summary size of dependencies		% reduction w.r.t. SC	runtime (secs.)	
Model	SC	FC	FC	SC	FC
i-Protocol	62	33	46.8	0.01	0.05
MVCC	35	35	0	0.01	0.66
GIOP	274	242	11.7	0.10	10.78
SMCS	410	338	17.5	1.37	45.17

Table 3. The comparison of different strategies to detect global dependencies caused by message receiving statements.

Table 3 shows two strategies to detect global dependencies caused by message receiving statements: *SC* is coarser and includes any cycle in the dependency as long as it may send the same type of messages as received by the considered receiving statement. *FC* checks the fastened cycles property in order to exclude cycles that cannot contribute a desired message. We observe that FC can reduce the sizes of dependencies quite considerably at the expense of a moderate to significant runtime penalty. The fact that FC did not reduce the dependency sizes for MVCC is expected because few variables are used in the model to store components of incoming messages. Moreover, those component storing variables are not used to control the behavior of the model, i.e., there are no message determined conditions. The extra runtime required by FC on MVCC was spent on checking the existence of message determined conditions. If we know a priori that no such conditions exist in a model, which can be achieved by a manual scan of the Promela code, then FC is unnecessary.

To illustrate the benefit of our analysis we applied our approach to the counterexample refinement of our livelock freedom analysis for Promela models [11]. We have mentioned that, by obtaining smaller and more dependencies, we stand a better chance to determine spuriousness for the counterexample. The previous version of our prototype livelock freedom checker *aLive* used the ND strategy to discover local dependencies and was not able to detect global dependencies. In [11] we reported that the local cycle dependency detection helped to remove 7 counterexamples for a model of the Group Address Registration protocol for which livelock freedom was successfully proved. For the GIOP model, 8 counterexamples were found and *aLive* failed to prove spuriousness for one of them. The spuriousness of this counterexample is caused by abstracting away a global dependency. After we employed the FC strategy proposed in this paper in *aLive*, the one remaining counterexample in GIOP was determined to be spurious and subsequently excluded from the abstraction. The same was observed during the checking of the i-Protocol model for which 4 more spurious counterexamples were discovered due to the detection of global dependencies.

We also performed experiments in which we used the cycle dependency analysis in the spuriousness determination of counterexamples found during our buffer boundedness analysis [12]. The increase in precision that we achieved lies within the range of increase that we obtained for the livelock freedom analysis.

8 Conclusion

The first contribution of our work is a formalization of the concept of control flow cycle dependencies. The second contribution is that we presented several incomplete but efficient static analysis methods for the detection of both local and global cycle dependencies for reactive systems of concurrent processes. Furthermore, we conducted experiments that show the precision of this analysis when applied to a set of models of real-life systems. We also show that the precision of our approach compared to naive cycle dependency detection techniques improves the precision of our livelock freedom and buffer boundedness analyses since more spurious counterexamples can be detected.

Future work will include improving our analysis by incorporating data flow analysis. We will also consider broadening the approach to a wider range of programming and modeling languages. Finally, we see a potential for the application of cycle dependency analyses to other application areas, such as the prediction of temporal conflicts and spatial localities of code blocks for the improvement of instruction cache hit rates [10].

Acknowledgment. The work of the second author was supported by the DFG-funded research project IMCOS (Grant No. LE 1342/1-/2). We thank Daniel Butnaru for his assistance in programming the implementation prototype.

References

1. A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*, pages 113–129. Springer, 2005.
2. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
3. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Proc. of SAS'05*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.
4. J.C. Corbett and G.S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, 1995.
5. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.
6. Y. Dong, X. Du, G.J. Holzmann, and S.A. Smolka. Fighting livelock in the GNU i-Protocol: a case study in explicit-state model checking. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 4(4):505–528, 2003.
7. T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *ACM Symposium on User Interface Software and Technology*, pages 1–10, 1996.

8. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.
9. M. Kamel and S. Leue. Formalization and validation of the general Inter-ORB protocol (GIOP) using PROMELA and SPIN. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4):394–409, 2000.
10. R. Kumar and D. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *Proc. of MICRO'02*, pages 419–429. ACM/IEEE, 2002.
11. S. Leue, A. Ştefănescu, and W. Wei. A livelock freedom analysis for infinite state asynchronous reactive systems. In *Proc. of CONCUR'06*, volume 4137 of *LNCS*, pages 79–94. Springer, 2006.
12. S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for the boundedness of UML RT models. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 327–341. Springer, 2004.
13. S. Leue and W. Wei. Counterexample-based refinement for a boundedness test for CFSM languages. In *Proc. of SPIN'05*, volume 3639 of *LNCS*, pages 58–74. Springer, 2005.
14. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer Verlag, 1992.
15. S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *PPOPP'93*, pages 129–138. ACM Press, 1993.
16. P. Merino and J.M. Troya. Modeling and verification of the ITU-T multipoint communication service with SPIN. In *Proc. of SPIN'96*, 1996.
17. L.I. Millett and T. Teitelbaum. Issues in slicing Promela and its applications to model checking, protocol understanding, and simulation. *STTT*, 2(4):343–349, 2000.
18. G. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *SIGSOFT FSE'98*, pages 24–34. ACM Press, 1998.
19. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2nd edition, 2005.
20. Radek Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
21. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. of VMCAI'04*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.
22. S.F. Siegel and G.S. Avrunin. Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Trans. Software Eng.*, 28(2):115–128, 2002.
23. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

A Appendix

A.1 The Proof of Theorem 1

We prove the theorem by a reduction from the problem below proved to be undecidable in [2][7].

Problem 1. Executability of a message reception in a system of communicating finite state machines (CFSM).

Instance: A CFSM M and a local state s of M having an outgoing transition t labeled by the receive action '$?a$'

Question: Does there exist a run of M such that the message reception '$?a$' is executed at s?

We construct a CFSM system M' from M by (1) introducing a new state s' in the same state machine as s is; (2) adding at s' a self-transition labeled with '$!b$' where b is a newly introduced type of message; (3) changing the target state of the transition t to the newly introduced state s'; and finally (4) adding a new state machine consisting of a single state s'' and a self-transition at s''. Moreover, let c be the self-loop at s' and S be the singleton cycle set consisting of the self-loop at s''.

We prove that '$?a$' can be executed at s in M if and only if c does *not* depends on S in M'.

For the "if" part, assume that c does *not* depends on S. Then, there exists an infinite run of M' in which c is executed infinitely often while the self-loop at s'' is not. From the construction of M', c can be executed only if '$?a$' can be executed at s in M', which means that '$?a$' can be executed also in M.

For the "only if" part, assume that '$?a$' can be executed at s in M. Then, '$?a$' can be also executed in M'. After '$?a$' is executed, c can be repeated alone forever, which means c does not depend on any other cycles for an infinite number of executions. □

A.2 The Proof of Proposition 1

If c is repeated an infinite number of times, then some other cycle c' in p must be repeated also infinitely often. On every path from a state in c to a state in c', there must be a transition t from an exit state of c such that t is not contained in c. There are only finitely many such transitions, so one of them is taken an infinite number of times and it belongs to at least one of the neighbors of c. □

[7] The proof is actually for the undecidability of the same problem for communicating finite state machines (CFSM). However, Promela models with unbounded buffers can simulate CFSM systems. Thus, the undecidability result also holds for Promela models.

A.3 The Termination and Soundness of the Algorithm in Figure 4

Proposition 3 (Termination). *The algorithm in Figure 4 always terminates.*

Proof. It is easy to see that no cycle can be added to `visited` more than once. Hence, each call to `search_for_preemptive_cycles` results in a new cycle being added to `visited` (Line 9). Note that our algorithm never removes any cycle from `visited`. Since there are only finitely many cycles, the algorithm must terminate. □

Proposition 4 (Soundness). *Given as an input a cycle c in process p such that c is terminating on a locally determined condition B, the algorithm in Figure 4 returns a cycle dependency (c, S) such that a cycle $c' \in MC_p(E_c(B))$ is preemptive if and only if $c' \in S$.*

Proof. We assign a natural number $level(d)$ to each cycle d that is added to `visited` as follow: (1) $level(c) = 0$; (2) if c_1 is enqueued (Line 18) or added to `ccs` (Line 16) inside the call to `search_for_preemptive_cycles(`c_2`,` B`)` and $level(c_2) = n$, then $level(c_1) = n + 1$. In the second case, we say that c_2 is the *parent* of c_1 and c_1 is a *child* of c_2. Then, we can build a *parent-child tree* (PCT).

For the "if" part, we prove that if $c' \in S$ then it is preemptive. It is easy to see that, in the path from the root c to c' in the PCT, no cycle except c and c' modifies any variable in $E_c(B)$. From this path, we can easily construct an acyclic path θ from an exit state t in c to an exit state t' in c', and an acyclic path θ' from t' to t. θ and θ' apparently do not modify any variable in $E_c(B)$.

For the "only if" part, assume that c' is preemptive. Then, there is an exit state t in c, an exit state t' in c', an acyclic path θ from t to t', and an acyclic path θ' from t' to t such that θ and θ' do not modify any variable in $E_c(B)$. The path $<\theta, \theta'>$ can be decomposed into a set of cycles, from which we can construct a sequence of pairwise distinct cycles $c_1, \ldots, c_n$ such that (1) c_1 is a neighbor of c, (2) c_n is a neighbor of c', and (3) each c_i and c_{i+1} are neighbors. It is easy to see no cycle in such a sequence modifies any variable in $E_c(B)$. Let *SEQ* be the set of shortest sequences of cycles as constructed in this way. Assume that the sequences in *SEQ* are of length n. For each sequence in *SEQ*, we add c to its head and attach c' to the end. We prove that there is one sequence $seq \in SEQ$ that is a path in the PCT, which implies that c' is added to *ccs*. The proof is by showing that, for any $k \leq n$, there is a sequence $seq \in SEQ$ such that its prefix of length i is a path in the PCT (*), by induction on the length i of prefixes of sequences in *SEQ*.

Induction base: The prefix of length 1 of any sequence in `SEQ` is c, which is a path in the PCT.

Induction step: Assume that (*) holds for k. Let P be the set of sequences in `SEQ` such that their prefixes of length k are paths in the PCT. Let C_k be the set of cycles $\{d \mid d \text{ is the } k\text{-th element in a sequence in } P\}$, and C_{k+1} be $\{d \mid d \text{ is the } (k{+}1)\text{-th element in a sequence in } P\}$. By contradiction, we assume that there is no sequence in P such that its prefix of length $(k+1)$ is a path in the PCT. Then, inside the call to `search_for_preemptive_cycles(`c_k`,` B`)` for each $c_k \in C_k$, none of the neighbors of c_{k+1} in C_{k+1} is enqueued or added to `visited`. This happens only when c_{k+1} is already in `open` or in `visited`. Let p be the parent of c_{k+1}. So, $p \notin P$. We have either that (1) $\mathit{level}(p) = k$, or that (2) $\mathit{level}(p) < k$. When $\mathit{level}(p) = k$, the path from c to p must be the prefix of length k of some sequence in *SEQ*, which means that $p \in P$. This leads to a contradiction. When $\mathit{level}(p) < k$, we construct a sequence of cycles from any sequence in P whose $(k+1)$-th element is c_{k+1}, by replacing the prefix of length k by p. The new sequence is shorter than the sequences in `SEQ`, which contradicts that `SEQ` contains the shortest sequences of pairwise distinct cycles connecting c and c'. □

A.4 The Proof of Proposition 2

Lemma 1. *Using the notation in the definition of the fastened cycles property in Section 5, the following is satisfied: For any path p_1 that ends at an exit state s within p, the path p_2 in c from the source state of t_1 to s is consecutively executed*[8] *in the end of p_1.*

Proof. We suppose that there are q exit states in p: $es_1, \ldots, es_q$. We prove the lemma by induction on the index k of es_k.

Induction base: es_1 is the source state of t_1. The path from es_1 to es_1 is an empty path which is always consecutively executed.

Induction step: Assume the lemma holds for es_m where $m < k$. Let p' denote the path from the source state of t_1 to es_k. Suppose that es_j is the last exit state at which the execution of p' is interrupted. We have that $j < k$. From the induction assumption, immediately before the execution p' is resumed at es_j, the path from the source state of t_1 to es_j is consecutively executed. Furthermore, after the execution of p' is resumed, the remaining part of p' is also consecutively executed. So, p' is consecutively executed. □

In the following, we prove Proposition 2 using the above lemma.

Proof. We denote by s_1 the statement `b?msg`$(x_1, \ldots, x_n)$, by s_2 the statement `(`B`)`, by t_1 the transition corresponding to s_1, by t_2 the transition corresponding to s_2, and by p the path from the source state of t_1 to the source state of t_2.

We denote by s_l the exit state within p at which the execution of c is interrupted at the last time in a run. We denote by p' the path from the source state of t_1 to s_l in c, and by p'' the path from s_l to the source state of t_2 in c . So, $p = <p', p''>$. Following Lemma 1, before the execution of c is resumed, p' is consecutively executed. Because s_l is the last state at which c is exited, p'' is also consecutively executed after c is re-entered. So, p is consecutively executed before the condition statement s_2 is reached. In this consecutive execution of p a message `msg`$(d_1, \ldots, d_n)$ is received and each variable $x_i \in \mathit{var}(B)$ is assigned

[8] Given two paths p_1 and p_2, we say that p_2 is executed in p_1 if p_2 is a subsequence of p_1. If p_2 is a consecutive subsequence of p_1, then we say that it is consecutively executed in p_1. In particular, an empty path is always consecutively executed.

with d_i. After p is executed, the execution of c can continue if and only if B is true. Since any variable $x_i \in var(B)$ is not modified in p after receiving the message, we have that $B[x_i \leftarrow d_i]$ is true. $\square$