A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems

Stefan Leue, Alin Ştefănescu, and Wei Wei

Department of Computer and Information Science
University of Konstanz, D-78457 Konstanz, Germany
Email: {Stefan.Leue|Alin.Stefanescu|Wei.Wei}@uni-konstanz.de

Abstract. We describe an incomplete but sound and efficient livelock freedom test for infinite state asynchronous reactive systems. The method abstracts a system into a set of simple control flow cycles labeled with their message passing effects. From these cycles, it constructs a homogeneous integer programming problem (IP) encoding a necessary condition for the existence of livelock runs. Livelock freedom is assured by the infeasibility of the generated homogeneous IP, which can be checked in polynomial time. In the case that livelock freedom cannot be proved, the method proposes a counterexample given as a set of cycles. We apply an automated cycle dependency analysis to counterexamples to check their spuriousness and to refine the abstraction. We illustrate the application of the method to Promela models using our prototype implementation named *aLive*.

1 Introduction

The main characteristic of a concurrent reactive system [17] is that of maintaining an ongoing activity of exchanging and processing information. One salient property that any reactive system must satisfy is deadlock freedom, i.e., the execution of the system is non-blocking. However, a system may be free of deadlock and yet it does no progress in executing its tasks. Such a situation is referred to as livelock. Freedom from livelock is highly desirable as it is important to ensure that the execution of a system is not only continuous but also meaningful.

Explicit state model checking techniques are mostly used to verify livelock freedom for finite state systems [4, 11, 9]. However, these techniques suffer from the state explosion problem especially when applied to asynchronous concurrent systems. Such systems usually possess a large global state space due to the combinatorial interleaving of the executions of local processes. On the contrary, integer programming (IP) based verification techniques does not rely on the enumeration of global states and thus avoid the state explosion problem. However, the existing IP based techniques focus on the analysis of synchronous systems.

In this paper we propose an incomplete analysis method for livelock freedom of asynchronous reactive systems, relying on the observation that control flow cycles play a central rôle in the setting of reactive systems with a "forever run"

behavior. We consider asynchronous message-passing as the underlying communication paradigm of the systems that we analyze. The livelock freedom test is reduced to the solving of a homogeneous integer programming problem, which can be done in polynomial time. In case the incomplete analysis method that we propose cannot establish livelock freedom we use a heuristic abstraction refinement method to improve the accuracy of our analysis. Since the size of the communication channels is not relevant to the analysis, we can assume they are infinite, meaning that our method can verify infinite state systems that cannot be addressed for instance via an explicit state exploration.

The paper is structured as follows. Section 2 introduces the running example described in Promela. Section 3 properly defines the livelock freedom problem, while Section 4 presents the core idea of checking livelock freedom using integer linear programming. Section 5 gives the cycle refinement procedure. We conclude in the last three sections with experimental results, related work, respectively conclusions and future work. Some of the details are placed in the appendix.

2 Promela

We briefly introduce into the *Promela* modeling language for concurrent systems and present the running example of this paper.

Promela is the input language of the *SPIN* explicit state model checker [11]. It has been successfully used for the modeling and analysis of many concurrent systems [12, 4]. The Promela language supports asynchronous communication as well as synchronous rendez-vous communication and synchronization via shared variables. In the scope of this paper, we concentrate on asynchronous communication and exclude the use of any other types of communication that Promela offers.

```
mtype = {req, ack, rel};

chan c_s[2] = [1] of {mtype};
chan s_c[2] = [1] of {mtype};

proctype client(int index) {
  do
  :: c_s[index]!req;
     s_c[index]?ack ->
        // do some computation here
       c_s[index]!rel;

  od;
}
```

```
proctype server() {
  do
  :: c_s[0]?req -> s_c[0]!ack; c_s[0]?rel;
  :: c_s[1]?req -> s_c[1]!ack; c_s[1]?rel;
  od
}

init {
  run server();
  atomic {
    run client(0); run client(1);
  }
}
```

Fig. 1. The running example in Promela.

Figure 1 shows a simple Promela model that will be used throughout the paper as a running example. The model consists of two instances of the *client* process type and one instance of the *server* process type. Each client *client[i]* exchanges messages with the server over two exclusive communication buffers

c_s[i] and *s_c[i]*. The types of exchanged messages are defined as elements of the special enumeration type *mtype*. Each client performs a loop: it first sends a resource request (`req`) to the server; after it receives an acknowledgment (`ack`) from the server, it performs some local computation and then sends back a resource release notification (`rel`). The server chooses nondeterministically a request to handle and grants the resource to one client, using an `ack` message, only after it receives a release notification from the other client.

The choice of the Promela language in the context of this paper is motivated by reasons of convenience. Promela possesses the salient features of most asynchronous concurrent system models, and a large number of models are publicly available. However, Promela was designed to be used for finite state verification and hence possesses some language features to ensure this property, such as limiting data to finite domains and requiring message buffers to have finite capacity. However, our livelock freedom analysis is applicable to both finite and infinite state systems which is why we simply ignore the respective Promela constructs. To facilitate our analysis we also assume that it is known at compile time how many Promela processes of which type will be instantiated at run time. In Section 4.3 we show that the soundness of our analysis does not rely on specifics of the Promela language, which is why we put forward that its application to other modeling and programming languages for asynchronous concurrent systems can easily be accomplished.

3 Livelock Freedom

Livelock has been defined variously in different contexts [10]. For concurrent systems, livelock often means "individual starvation": a process is prevented from performing some particular actions [17]. These actions are normally intended to make progress, deliver outputs, or respond to the environment and other peer processes. We call such an action a *progress action*. In the running example, a progress action of a client is to do the local computation after it receives an acknowledgment. In this paper we follow this meaning of livelock and give its definition in the setting of reactive systems.

We define that a *livelock* for a reactive system is an infinite run in which only non-progress actions are executed after a certain point of the run, i.e., all the progress actions are repeated only a finite number of times. If a reactive system has no livelock runs, then it is *livelock free*.

Both livelock and deadlock result in a lack of progress in the system. They are sometimes not distinguishable from a practical point of view. However, these two concepts are used to refer to two different sources of non-progress. Furthermore, from a formal point of view, they belong to two different types of properties: deadlock freedom is a safety property while livelock freedom is a liveness property. As a consequence, the techniques used to check these two properties are radically different. That is why we make a clear distinction of deadlock and livelock in our definition: a finite run, in particular a deadlocked run, is not a livelocked run. In our analysis, we focus on checking the absence of livelock and

ignore the existence of deadlocks: if a system is proved to be livelock free using our method, it may still have deadlocks.

The SPIN model checker distinguishes in a similar fashion between deadlock and livelock [11]. In Promela models, "progress" labels are attached to progress actions. SPIN then checks livelock freedom by checking the absence of non-progress global cycles by a nested depth first search in the global state space. However, such a state enumeration approach suffers from the state explosion problem and can only deal with finite state systems. In Appendix A we prove that livelock freedom is undecidable for infinite state systems with unbounded communication buffers.

4 Livelock Freedom Analysis

We propose an incomplete but sound method to prove livelock freedom for asynchronous reactive systems based on integer programming solving. The incompleteness is a consequence of the undecidability of the livelock freedom problem.

We outline the method as follows. Given a reactive system and a set of progress actions, we first carry out a series of abstractions that transforms the system into a set of independent control flow cycles labeled with their message passing effects. A cycle is a *progress cycle* if it contains one of the progress actions, and we identify the set of all the progress cycles. We give a *necessary condition* which ensures the existence of a livelock run, i.e., an infinite run in which all the progress cycles are repeated only a finite number of times. We translate this condition into a homogeneous *integer programming problem* (IP). If the resulting IP problem has no solution then the necessary condition cannot hold, which implies livelock freedom. On the other hand, if the resulting IP has solutions then the system may or may not be livelock free, which corresponds to the incomplete side of our test.

4.1 Abstraction

In asynchronous reactive systems, concurrent processes coordinate their actions via exchanging messages. Thus, the message passing behavior is a major factor to decide how cycles in the control flow are executed. This observation underlies the conservative abstraction approach sketched below for our livelock freedom analysis. The same abstraction steps were also used in our previous work on buffer boundedness analysis, which are detailed in [14, 13]. In particular [13] deals with specifics of abstracting Promela models.

Code abstraction. Given the program code of a reactive system we first abstract from variables, operations on data, the testing of conditions, etc., and retain only the finite control structure and the message passing behaviour of all processes. The resulting system is a system of *communicating finite state machines* (CFSM) [22, 1].

Message orders. In the next step, we abstract from the order of messages in any communication buffer. We use an integer vector to represent how many messages of a certain type are currently stored in each buffer. Consider the running example. An integer vector $< 1, 0, 3, 2, 4, 6 >$ denotes that there is 1 *req* message in the buffer *c_s[0]*, no *ack* message in *s_c[0]*, 3 *rel* messages in *c_s[0]*, 2 *req* messages in *c_s[1]*, 4 *ack* messages in *s_c[1]*, and 6 *rel* messages in *c_s[1]*. We also use an integer vector, called an *effect vector*, to denote the message passing effect of a transition. A positive component in an effect vector corresponds to message sending, and a negative component corresponds to message consumption.

Activation conditions and dependencies of control flow cycles. In this step, we assume that (1) any control flow cycle can be reached from the initial configuration of the system and that (2) these cycles are totally independent from one another. We detect all the local control flow cycles in each process of the system. We consider only simple cycles, i.e., cycles that cannot be decomposed into smaller cycles. For each cycle, we compute the sum of the effect vectors of all the transitions along the cycle. The resulting system is a set of independent control flow cycles with their effect vectors. In the running example, there are 4 cycles: one from the process *client[0]*, one from *client[1]*, and two from *server* given as the two nondeterministic choices within the `do` loop. Their effect vectors are, respectively, $< -1, 1, -1, 0, 0, 0 >$, $< 0, 0, 0, -1, 1, -1 >$, $< 1, -1, 1, 0, 0, 0 >$, and $< 0, 0, 0, 1, -1, 1 >$.

4.2 Determining Livelock Freedom

A reactive system is livelock free if at least one progress cycle can be repeated infinitely often in any infinite run. Let $C_1, \ldots, C_n$ be the set of control flow cycles that we collect from the system, and $C_{j_1}, \ldots, C_{j_m}$ $(j_1, \ldots, j_m \in \{1, \ldots, n\})$ be the set of progress cycles. We use c_i to denote the effect vector of a cycle C_i. We use the following IP problem to characterize a necessary condition for the existence of a livelock run, i.e., an infinite run in which any progress cycle can be repeated only a finite number of times.

$$x_1 c_1 + \ldots + x_n c_n \geq \bar{0} \tag{1}$$
$$x_1 + \ldots + x_n > 0 \tag{2}$$
$$x_{j_1} + \ldots + x_{j_m} = 0 \tag{3}$$
$$x_i \geq 0 \quad \text{for all } i \tag{4}$$

In the above inequalities, we assign an integer variable x_i to each cycle C_i to denote the number of times that it is repeated in a finite segment of a run. These variables may have only non-negative values as imposed by the inequalities 4. A particular assignment to all x_i's represents a *linear combination of cycle executions*. The inequality 1 states that the *overall* effect of a linear combination of cycle executions does not consume any messages. Thus, an infinite exclusive

repetition of such a linear combination is possible since it does not run out of any type of messages. The inequality 2 excludes a trivial combination in which no cycle is executed at all. The inequalities 1 and 2 give a necessary condition for the existence of infinite runs. The inequality 3 then excludes any progress cycle C_{j_i} from a linear combination. Consequently, this condition excludes any progress cycle from being repeated infinitely often in any infinite run. The arguments in Section 4.3 ensure that the IP problem defined by the inequalities 1–4 gives indeed a necessary condition for the existence of livelock runs.

If the IP problem has no solutions, then the necessary condition cannot hold. In such a case, at least one progress cycle C_{j_i} has to be repeated infinitely often in any infinite run. This proves livelock freedom for the system. On the other hand, if the IP problem has solutions, then we do not know whether the system is livelock free since the IP problem gives a necessary but not sufficient livelock existence condition.

Consider the running example. Let the only progress action be the local computation of one client, say *client[0]*. We use x_1 to correspond to the cycle in *client[0]*, x_2 to the cycle in *client[1]*, and x_3 and x_4 to the two cycles given as the two nondeterminstic choices within the `do` loop in *server*. The resulting livelock freedom determination IP problem is given as below.

$$-x_1 + x_3 \geq 0 \quad (5) \qquad x_2 - x_4 \geq 0 \quad (9)$$
$$x_1 - x_3 \geq 0 \quad (6) \qquad -x_2 + x_4 \geq 0 \quad (10)$$
$$-x_1 + x_3 \geq 0 \quad (7) \qquad x_1 + x_2 + x_3 + x_4 > 0 \quad (11)$$
$$-x_2 + x_4 \geq 0 \quad (8) \qquad x_1 = 0 \quad (12)$$
$$x_1, x_2, x_3, x_4 \geq 0 \quad (13)$$

The inequalities 5–10 restrict the aggregate effect vector of a linear combination to be positive[1]. The inequality 11 excludes an all-zero combination. The inequality 12 excludes the only progress cycle in *client[0]*. There is one solution satisfying these inequalities: $x_2 = x_4 = 1$ while assigning 0 to all other variables. As a consequence we cannot prove livelock freedom for the running example. However, we can construct a *counterexample* from the above obtained solution as a collection of cycles whose variable receives a nonzero value in the solution. A manual check of the counterexample reveals a real livelock scenario in which the server decides to accept only requests from *client[1]*. Note that due to the overapproximating abstractions that we use, a counterexample corresponds not always to a valid execution of the system. In such a case, the counterexample is called *spurious*. An automated method to determine spurious counterexamples will be discussed in depth in Section 5.

To eliminate the source of livelock that we uncovered above, we modify the model by removing the nondeterministic behavior of the server. We fix an order in which the server alternatively handles requests from the two clients as follows:

[1] A vector is positive if all its components are non-negative.

```
proctype server() {
  do
  :: c_s[0]?req -> s_c[0]!ack; c_s[0]?rel;
     c_s[1]?req -> s_c[1]!ack; c_s[1]?rel;
  od
}
```

The resulting IP problem for the revised model, given below, has no solutions, which implies livelock freedom.

$$-x_1 + x_3 \geq 0 \quad (14)$$
$$x_1 - x_3 \geq 0 \quad (15)$$
$$-x_1 + x_3 \geq 0 \quad (16)$$
$$-x_2 + x_3 \geq 0 \quad (17)$$
$$x_2 - x_3 \geq 0 \quad (18)$$
$$-x_2 + x_3 \geq 0 \quad (19)$$
$$x_1 + x_2 + x_3 \geq 1 \quad (20)$$
$$x_1 = 0 \quad (21)$$
$$x_1, x_2, x_3 \geq 0 \quad (22)$$

Complexity of the livelock freedom test. Given a reactive system, the size of the constructed IP problem is linear in the number of message types and in the number of simple local control flow cycles. The number of simple cycles may be exponential in the size of the control flow graph. However, in practice the control flow graph extracted from the Promela code of a process is sparse, and we observed that the number of simple cycles is usually polynomial.

Furthermore, the IP problem that our method constructs is homogeneous, i.e., the right hand side of each inequality in the problem is 0. This homogeneous IP problem can be solved in polynomial time as follows. We solve the linear programming version of the IP problem to obtain a rational solution. This can be done in polynomial time [19]. If we obtain a rational solution, we can easily construct an integer solution by multiplying each component in the rational solution by the least common denominator of all the components.

4.3 Soundness Proof

In the soundness proof we use the following proposition, whose proof can be found in Appendix B.

Proposition 1. *Let S be a CFSM system and C a subset of control flow cycles in S. Suppose that there exists no positive linear combination of effect vectors of cycles in C. Then, for any infinite execution in which only cycles in C are executed, the number of messages in all the communication buffers is always bounded.*

Theorem 1 (Soundness). *If we prove livelock freedom for a reactive system using the method described in Subsection 4.2, then the system is indeed livelock free.*

Proof. Consider a reactive system for which we use our method to prove livelock freedom. The first abstraction step constructs a CFSM from the original system

in a conservative way in that it preserves the existence of livelock runs. Thus, if the CFSM is livelock free, then the original system is also livelock free.

Assume that the reactive system is proved to be livelock free. Then, in the corresponding CFSM there exists no positive linear combination of effect vectors of non-progress cycles (as there is no solution to the corresponding IP problem described by the inequalities 1–4). By Proposition 1, taking C to be the set of all non-progress cycles, we obtain that the number of messages in each communication buffer is bounded if only non-progress cycles are executed.

We prove that the CFSM is livelock free, which implies livelock freedom for the original system. We assume *by contradiction* that the CFSM has a livelock run r. In r all the progress cycles are repeated only a finite number of times. Then, there exists a particular point of time t in r after which only non-progress cycles are executed. As discussed above, following Proposition 1, the number of messages in each communication buffer must always be bounded after t in r. Note that any state machine in the CFSM has only finitely many local states. Thus, there will be only finitely many reachable configurations of the CFSM after t in r. Furthermore, because r is an infinite run, there must be two distinct points of time t' and t'' after t at which the CFSM reaches one same configuration. The finite segment of execution between t' and t'' can be represented as a linear combination of executions of non-progress cycles. The aggregate effect vector of this segment is however an all-zero vector. This contradicts the previous claim that no linear combination of effect vectors of non-progress cycles is positive. □

Note that the above proof does not use any assumption about buffer lengths. Consequently, if a system with unbounded buffers is proved to be livelock free, then the same system with bounded buffers of predefined lengths is also livelock free.

5 Counterexample-based Refinements

The abstractions described in the previous section reduce the accuracy of the analysis, and our method may propose spurious counterexamples. We observed that the introducing of spurious counterexamples is often caused by the abstraction from those conditional statements that determine the repeatability of control flow cycles. As we will show later, such conditionals enforce dependencies among cycles that have been lost during the abstractions. In [15] we have proposed a counterexample-based refinement technique based on re-discovering local dependencies among the cycles of a same process. This technique can be adopted to the livelock freedom analysis in this paper and will be illustrated on a simple example. We will also present an improvement to the determination of cycle dependencies in [15] that is more efficient and precise in practice. We mention that all the techniques used in the refinement procedure are conservative with respect to livelock freedom.

Cycle Dependency Analysis. The details of the cycle dependency analysis can be found in [15]. Here we only illustrate the basic idea of the technique on a

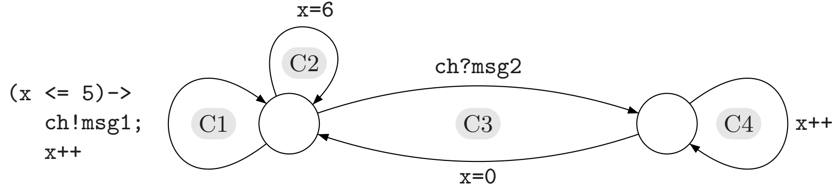

Fig. 2. Running example of Section 5

simple example. Figure 2 shows the control flow graph of a process in a reactive system. It contains four cycles C_1, C_2, C_3 and C_4. Suppose that none of them is a progress cycle and that the integer variable x is local. C_1 is enabled when the value of x is no larger than 5. When executed, it sends a message `msg1` and increments the value of x by 1. C_1 leads to a spurious counterexample because the condition $x \leq 5$ is omitted during the abstraction, i.e., in the abstraction it is assumed that C_1 can be executed forever without interruption. To exclude this spurious counterexample, we perform the following cycle dependency analysis.

– We first determine that C_1 can be repeated without interruption at most 5 times before the condition $x \leq 5$ turns false. However, the determination of the maximal number of times that a cycle iterates relies on a termination decision which is undecidable. We recently proposed an incomplete automated termination proving technique [16] that can be easily extended to estimate cycle iteration counts, complementing the approach described in [15].
– We determine two sets of cycles that C_1 depends on. One set consists of all C_1's *neighbors*, i.e., cycles that share some common states with C_1. In our example the neighbors are C_2 and C_3. Another set of cycles that we determine consists of all the so-called *supplementary cycles* that, intuitively speaking, exert a positive effect to enable the execution of C_1 again, i.e., to render the condition $x \leq 5$ to become true. As can be easily seen, C_1 has only one such supplementary cycle which is C_3. However, in general it is hard to determine the exact set of supplementary cycles for a given cycle. Later in this section we will propose a so-called “next door” strategy that can be used to determine supplementary cycles more efficiently and precisely in practice.
– The following cycle dependencies can be determined from the above analysis: every 5 times that C_1 is repeated, (1) one of its neighbors has to be executed at least once; and (2) one of its supplementary cycles has to be executed at least once.

Refinement. We can easily express the two above determined dependencies using two linear inequalities. Let c_i be the variable corresponding to C_i in the respective livelock freedom determination IP problem. The inequality $c_1 \leq 5(c_2 + c_3)$ describes the first dependency regarding neighbors, and the inequality $c_1 \leq 5c_3$ describes the second one regarding supplementary cycles. These two inequalities are then added to the livelock freedom determination IP problem, which refines

the abstraction by imposing the discovered cycle dependencies and thus ruling out the spurious counterexample consisting of only C_1.

In [15] we also consider other sources of cycle dependencies than those imposed by conditional statements in cycles. We leave out the discussion here due to limited space.

Next door strategy. In [15] two alternative methods were proposed to determine supplementary cycles. These methods are either relatively coarse or costly, and we give an improvement using what we call the "next door" strategy.

Note that, in the example in Figure 2, the incrementation of x in C_4 does not influence the satisfaction of the condition in C_1. This is because x is re-assigned with a constant by C_3, a neighbor of C_1, on the way back to C_1 from C_4. Thus, C_4 is not a supplementary cycle of C_1. In fact we can see that C_1 is *isolated* by all its neighbors in that, upon re-entering C_1, x is always reset to some constant by one of its neighbors. In such a case, the satisfaction of the condition in C_1 is totally decided by its neighbors, and thus no cycle other than a neighbor is supplementary to C_1.

Given a cycle whose supplementary cycles are to be determined, the next door strategy will first check whether the given cycle is isolated by all its neighbors. When this is the case, we can safely restrict the search for supplementary cycles to the set of its neighbors.

6 Case Studies

We implemented the livelock freedom proving method in a prototype system named "aLive", and carried out a few case studies with realistic Promela models on a Pentium IV 1.60GHz machine with 1GB memory. We also compared the performance of aLive and the SPIN model checker on each model.

GARP. The *Group Address Registration Protocol* (*GARP*) is a network protocol allowing users to dynamically register to and detach themselves from a multicast group. A progress action is either for a user to join or leave a multicast group, or for the system to remove all the users from a group. The Promela implementation of GARP [18] consists of 7 concurrent processes with 131 local states, 212 local transitions, and 10 communication buffers. SPIN proved livelock freedom for the model within 56 seconds and visited 5×10^6 global states during the check. aLive used only 8 seconds to return the same result after 7 abstraction refinement steps. We contend that this seven fold speedup compared to SPIN is possible because aLive does not need to visit all reachable global states and thus avoids the combinatorial state space explosion caused by concurrency that the verification algorithm of SPIN is subject to. During the verification aLive identified 29 message types, collected 86 local control flow cycles, and generated altogether 21 IP problems. During the analysis 7 counterexamples were suggested and aLive automatically determined all of these to be spurious. One of these spurious counterexamples suggests the following scenario: While no other

process moves on, one process keeps executing a cycle in which it sends a *join* message to inform a service process of some user's decision to join a multicast group. However, after the message is sent, the user is included in the group, and the process cannot send another *join* message. This cycle can be repeated only if another cycle of the same process has been executed in which the process receives a message through which the user announces that he is leaving the group. aLive successfully detected this dependency between these two cycles and refined the abstraction accordingly.

GSM Handover. We also checked livelock freedom for a model of the *Handover* procedure in the GSM protocol. The model is included as an example in the latest SPIN 4.26 distribution. In this case a progress action is to hand over the control of communication from one base region to another one. We carefully revised the original model to remove the use of sending data objects corresponding to communication buffers from one process to another, which is a Promela language feature that we currently cannot handle. However, the revision does not change the behavior of the original model. The revised model consists of 6 processes with 49 local states, 62 local transitions, and 7 communication buffers. For the revised model, SPIN immediately reported an error trail with a length of 36 steps. aLive also found one counterexample in the first checking iteration and returned `UNKNOWN` after it failed to determine spuriousness for the counterexample. The counterexample consists of 6 control flow cycles and indicates the situation in which a base station is continuously forwarding messages between a mobile user and the system without handing over the control to another base region. Guided by this counterexample, we replayed the indicated scenario by a manual simulation of the *original* model within exactly 36 steps. Thus, the counterexample that aLive found is a real counterexample.

CORBA GIOP. Our analysis of the CORBA GIOP [12] protocol revealed a limitation of the current aLive approach that is rooted in the unavailability of suitable static analysis methods for global cycle dependencies in the abstraction refinement loop. aLive found 8 counterexamples during the analysis and determined spuriousness for all but the last one. The failure on the last counterexample results exactly from the existence of a global cycle dependency that we cannot currently handle. A manual inspection easily proves the spuriousness of this counterexample. On the contrary, SPIN proved livelock freedom for the GIOP model very efficiently.

Analysis of parametric and infinite state models. Note that aLive actually proves livelock freedom for a class of Promela models that can be parameterized with *arbitrary* finite communication buffer capacities. SPIN, on the other hand, verifies only a given model with a fixed finite buffer length setting. As a consequence, if the buffer lengths specified in a Promela model are increased, SPIN may run out of memory due to an exponential growth of the size of the global state space and hence be unable to prove livelock freedom, while aLive is insensitive to the size

of the buffers bounds. Even more, if we assume that the (syntactically inadmissible) omission of buffer bounds in Promela channel declarations is interpreted as buffers with unbounded capacity, then our aLive analysis extends to the class of infinite state Promela models.

7 Related Work

Integer programming (IP) based techniques were previously used in the verification of concurrent systems [2, 5, 3, 6, 20]. INCA [2] relies on IP to provide an incomplete but sound method of verifying safety and liveness properties. However, INCA currently handles only synchronous rendezvous-like communication, although the theoretical framework is extensible to asynchronous communication. Furthermore, the analysis in INCA is restricted to control flow structures whereas our method also takes data into account. Also, the refinements of the control flow constraints proposed in [20] for INCA are different than the ones proposed in this paper. The work described in [5] uses a notion of T-invariants described in constraint programming (a more powerful framework than IP) to give an NP-complete semi-decision test ("yes" or "unknown") for LTL liveness properties on 1-safe Petri nets.

Livelock analysis was also studied in the context of process algebras. Tools explicitly verifying livelock freedom in the synchronous communication model of CSP are [7, 21]. Note that our analysis focusses on the asynchronous communication model and is therefore fundamentally different.

In explicit state model checking, the verification of livelock freedom reduces to the detection of non-progress cycles using nested [11] or even simple [9] depth first state space traversals. In fact, any CTL or LTL model-checker is able to address livelock freedom checking [4], since livelock freedom can be expressed in both CTL and LTL temporal logics. SPIN [11] checks livelock-freedom of Promela models (attaching 'progress' labels to actions of interest and searching for non-progress cycles). Verisoft [8] addresses also the livelock freedom issue, but with a very restricted definition of livelock that is only applicable to finite executions.

8 Conclusion

In this paper we have presented an incomplete analysis method for the detection of livelock freedom for asynchronous infinite-state reactive systems. The method is based on a property conserving abstraction that reduces these systems to a system of numerical effect vectors. The livelock problem is then encoded into an integer linear programming problem over these effect vectors. The solvability of this IP problem answers the question, whether the program is livelock free, or whether livelock freeness cannot be proven. In the latter case the analysis returns a counterexample. We have devised automated heuristics to determine spuriousness of a given counterexample and to refine the abstraction, when applicable. We have evaluated the analysis using a number of real-life Promela models that

we subjected to our prototype analysis tool aLive. The analysis together with the automated refinement in aLive have produced meaningful results. In one instance our automated counterexample refinement failed, which points at necessary improvements in the underlying static analysis. We have also compared our analysis with finite-state verification, in particular the SPIN model checker, and found that aLive performs very favorably. As we have argued, the soundness of our analyis does not hinge upon finiteness of the underlying model.

Further research aims at investigating how to encode other types of liveness properties, such as response properties, using effect vector analysis. Furthermore, we plan to use the counterexamples (linear combination of cycles) produced by our analysis to guide SPIN in its search for non-progress cycles (livelocks) using, for instance, as heuristic metric function the distance to a cycle in the counterexample. We also plan to improve our static analysis and extend it to global cycle dependencies so that impediments to automated abstraction refinement such as they occurred in GIOP will be eliminated.

Acknowledgments : This work was supported by the DFG-funded research project IMCOS (Grant No. LE 1342/1-/2). We thank George Avrunin, Javier Esparza and Keijo Heljanko for their pointers to relevant literature.

References

1. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
2. 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.
3. S. Dellacherie, S. Devulder, and J.-L. Lambert. Software verification based on linear programming. In *Proc. of FM'99*, volume 1709 of *LNCS*, pages 1147–1165. Springer, 1999.
4. 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.
5. J. Esparza and S. Melzer. Model checking LTL using constraint programming. In *Proc. of ICATPN'97*, volume 1248 of *LNCS*, pages 1–20. Springer, 1997.
6. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16(2):159–189, 2000.
7. FDR2 tool. Formal Systems (Europe) Ltd. http://www.fsel.com.
8. P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.
9. H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. *ENTCS*, 66(2), 2002.
10. A. Ho, S. Smith, and S. Hand. On deadlock, livelock, and forward progress. Technical Report UCAM-CL-TR-633, Cambridge University, Computer Laboratory, 2005. 8 pp. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-633.pdf.
11. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.

12. 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.
13. S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for message buffer overflow in Promela models. In *Proc. of SPIN'04*, volume 2989 of *LNCS*, pages 216–233. Springer, 2004.
14. 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.
15. 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.
16. S. Leue and W. Wei. A region graph based approach to termination proofs. In *Proc. of TACAS'06*, volume 3920 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2006.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer Verlag, 1992.
18. T. Nakatani. Verification of group address registration protocol using PROMELA and SPIN. In *Proc. of SPIN'97*, 1997. http://spinroot.com/spin/Workshops/ws97/nakatani.pdf.
19. C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity.* Prentice Hall, 1982.
20. 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.
21. SLAP tool (version 0.1): A static livelock analyzer for CSP processes. Webpage: http://web.comlab.ox.ac.uk/oucl/work/joel.ouaknine/software/slap.html.
22. G. von Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.

A Undecidability of livelock freedom in our setting

Livelock freedom is proved to be in general undecidable by a simple reduction from the following problem proved to be undecidable in [1]:

Executability of a message reception in a CFSM with unbounded buffers:

INSTANCE: A CFSM M and a local state s of M having an outgoing edge labeled by the receive action '$?a$'

QUESTION: Does there exists a run of M such that the message reception '$?a$' is executed in state s?

Given M, s, and $?a$ as above, we construct another CFSM M' in such a way that after the reception '$?a$', M' may to enter in a livelock. More precisely, M' is obtained from M by replacing the outgoing edge from s labeled by $?a$, by another edge also labeled by $?a$ but going to a new local state s_L that has a self-loop sending a special message $!a_L$. Moreover, we add also a new component with a single state s'_L with a self-loop labeled by $?a_L$ (receiving the a_L messages). In the new M' we set as progress actions, all actions except $!a_L$ and $?a_L$.

It is now easy to see that the reception $?a$ is executed in state s of M if and only if M' has a livelock run (involving a_L): For the direct implication, let r be a finite run of M where $?a$ in state s is eventually executed (we consider the run r to contain only one such occurrence of $?a$, and namely on the last position). Then, we can simulate the same run r in M' and reach the local state s_L. At this point we obtain a livelock by infinitely executing alternations of sends and receptions $!a_L$ and $?a_L$ (the only non-progress actions of M'). For the reverse implication, if M' contains a livelock, this necessarily involves the a_L message, but this is possible only if state s_L is reached, which implies that the reception $?a$ is executable in s in M'. From the construction of M' upon M, we will also be able to find a run in M such that $?a$ is executed in s. □

B Proof of Proposition 1

In the following we denote by $[i..j]$ the set $\{i, \ldots, j\}$ (for $i \leq j$) and by $\bar{x}$ the n-dimensional integer vector $(x_1, \ldots, x_n) \in \mathbb{Z}^n$. For two n-dimensional vectors, we have $\bar{x} \leq \bar{y}$ iff $x_i \leq y_i$, for all $i \in [1..n]$. Moreover, $\bar{x} < \bar{y}$ iff $\bar{x} \leq \bar{y}$ and there exists $i \in [1..n]$ with $x_i < y_i$.

Lemma 1. *Let $\{\bar{c}_0, \ldots, \bar{c}_n\}$ be $n+1$ vectors of dimension m (with $n, m \geq 1$), i.e., $\bar{c}_i := (c_{i1}, \ldots, c_{im})$ for all $i \in [0..n]$. Then, if the following system of linear inequations has no integer solutions*

$$x_1\bar{c}_1 + \ldots + x_n\bar{c}_n \geq \bar{0} \tag{23}$$

$$x_1 + \ldots + x_n > 0 \tag{24}$$

$$x_i \geq 0 \quad \textit{for all } i \tag{25}$$

then, there exists an upper bound B such that for all integer solutions of

$$\bar{c}_0 + x_1\bar{c}_1 + \ldots + x_n\bar{c}_n \geq \bar{0} \tag{26}$$

$$x_1 + \ldots + x_n > 0 \tag{27}$$

$$x_i \geq 0 \quad \textit{for all } i \tag{28}$$

and any $k \in [1..m]$,

$$c_{0k} + x_1c_{1k} + \ldots + x_nc_{nk} \leq B.$$

Proof. Defining for each $k \in [1..m]$ a function $f_k : \mathbb{Z}^n \to \mathbb{Z}$ as $f_k(x_1, \ldots, x_n) := x_1c_{1k} + \ldots + x_nc_{nk}$, we will prove that f_k is bounded for any $k \in [1..m]$ on the domain of integer solutions of (26)–(28).

By contradiction, assume that there exists a $k \in [1..m]$ such that f_k is unbounded. This implies that there exists an infinite sequence $\{\bar{x}^i\}_{i\geq 1}$ of integer solutions of (26)–(28) such that $\lim_{i\to\infty} f_k(\bar{x}^i) = +\infty$ (the limit cannot be $-\infty$ because of (26)).

We first show that without loss of generality, we can assume that the sequence $\{\bar{x}^i\}_{i\geq 1}$ has the property that

$$\text{for any } i < j : \quad \bar{x}^i < \bar{x}^j \text{ and } f_k(\bar{x}^i) < f_k(\bar{x}^j) \tag{29}$$

This can be proved using standard mathematical analysis techniques as follows. Since $\lim_{i\to\infty} f_k(\bar{x}^i) = +\infty$, we can select an infinite subsequence $\{\bar{y}^i\}_{i\geq 1}$ of $\{\bar{x}^i\}_{i\geq 1}$ such that $\{f_k(\bar{y}^i)\}_{i\geq 1}$ is strictly increasing. Moreover, we can select $\{\bar{y}^i\}_{i\geq 1}$ to be also strictly increasing. This is possible because $\{\bar{y}^i\}_{i\geq 1}$ is on one hand bounded from below by $\bar{0}$ following (28), while on the other hand is an infinite sequence taking f_k to $+\infty$. In the following, we replace $\{\bar{x}^i\}_{i\geq 1}$ by $\{\bar{y}^i\}_{i\geq 1}$ (for the sake of consistency with the notation in (29)).

Next, we observe the behavior of the increasing sequence $\{\bar{x}^i\}_{i\geq 1}$ on the other functions $f_{k'}$, for $k' \neq k$. We have the following two possibilities for each $k' \in [1..m] \setminus \{k\}$:

- $\{f_{k'}(\bar{x}^i)\}_{i\geq 1}$ is bounded: In this case, since $\{f_{k'}(\bar{x}^i)\}_{i\geq 1}$ is also infinite, there exists an infinite increasing subsequence $\{\bar{y}^i\}_{i\geq 1}$ of $\{\bar{x}^i\}_{i\geq 1}$ such that $f_{k'}(\bar{y}^i) = f_{k'}(\bar{y}^j)$, for any $i, j \geq 1$.
- $\{f_{k'}(\bar{x}^i)\}_{i\geq 1}$ is unbounded: Because of (26), $\{f_{k'}(\bar{x}^i)\}_{i\geq 1}$ is bounded from below (by $-c_{0k}$), so necessarily there exists an infinite increasing subsequence $\{\bar{y}^i\}_{i\geq 1}$ of $\{\bar{x}^i\}_{i\geq 1}$ such that $f_{k'}(\bar{y}^i) < f_{k'}(\bar{y}^j)$, for any $i < j$ (similar to (29)).

From (29) and the above case analysis (applied stepwise for each k'), it is easy to see that there exists an infinite strictly increasing sequence $\{\bar{y}^i\}_{i\geq 1}$ (whose elements are solutions of (26)–(28)) such that $f_k(\bar{y}^i) \leq f_k(\bar{y}^j)$, for any $k \in [1..m]$, and $i < j$.

Finally, let us fix two indexes $i < j$ (from $[1..m]$). Then, for all $k \in [1..m]$ we have $f_k(\bar{y}^i) \leq f_k(\bar{y}^j)$, which implies $f_k(\bar{y}^j) - f_k(\bar{y}^i) \geq 0$. But since all the functions f_k are linear, we have that $f_k(\bar{y}^j - \bar{y}^i) \geq 0$ (*). Moreover, since $\{\bar{y}^i\}_{i\geq 1}$ is strictly increasing, $\bar{y}^j - \bar{y}^i > \bar{0}$ (**). Slightly rewriting (*) and (**), we obtain that $\bar{y}^j - \bar{y}^i$ is a solution to the system of inequations (23)–(25), which is a contradiction with the hypothesis of the lemma. □

We prove now Proposition 1 using Lemma 1. Given a CFSM and a subset of control flow cycles $C = \{C_1, \ldots, C_n\}$, we consider $\bar{c}_i$ as the effect vector of cycle C_i for each $i \in [1..n]$. Moreover, let $\bar{c}_0$ be an upper bound for all the effect vectors of the all acyclic paths of the CFSM.

First, since there exists *no positive* linear combination of effect vectors of cycles in C (from the hypothesis of Proposition 1), the hypothesis of Lemma 1 is satisfied, which implies that there is a global upper bound for $\bar{c}_0 + x_1\bar{c}_1 + \ldots + x_n\bar{c}_n$ for any fixed $\bar{c}_0$ and $\bar{x} := (x_1, \ldots, x_n)$. This means that there is an upper bound B on all the message buffers for all executions consisting of an acyclic path followed by a *linear combination* of simple cycles.

Secondly, suppose now by contradiction that there exists a run of the CFSM that strictly exceeds the bound B in one of the buffers and let us denote by r a finite run that increases the number of messages in one of the buffers to $B + 1$. Since r is necessarily composed of an acyclic path and a finite number of simple cycles (seen as a linear combination of cycles), the effect of r on the message buffers is bounded by B (according to the above application of Lemma 1), but this contradicts the previous assumption on r being able to fill $B + 1$ messages on one of the buffers. □