Counterexample Refinement for a Boundedness Test for CFSM Languages

Stefan Leue and Wei Wei

Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Email: [Stefan.Leue|wei]@inf.uni-konstanz.de

Abstract. In precursory work we suggested an abstraction-based highly scalable semi-test for the boundedness of Communicating Finite State Machine (CFSM) based modelling and programming languages. We illustrated its application to Promela and UML-RT models. The test is sound with respect to determining boundedness, but may return inconclusive "counterexamples" when boundedness cannot be established. In this paper we turn to the question how to effectively determine the spuriousness of these counterexamples, and how to refine the abstraction based on the analysis. We employ methods from program analysis and illustrate the application of our refinement method to a number of Promela examples.

1 Introduction

For various reasons the unboundedness of the maximal filling of communication channels in Communicating Finite State Machine (CFSM) [2] based modelling and programming languages at runtime is an undesirable property. First, unboundedness of communication buffers per se is an undesirable system feature since it points at a design flaw. Second, the unboundedness of a channel leads to an infinite state space and limits the applicability of finite state verification methods. Finally, in otherwise finite state models the unboundedness of a communication channel may lead to a model that reveals unpredictable behavior, for instance when a process attempts to write to a channel that has reached its specified capacity limit.

In precursory work [8, 9] we have defined an incomplete semi-test for the boundedness of the message buffers in CFSM systems, which is an undecidable problem [2]. We have implemented the test in a tool named IBOC and applied it to CFSM models given in UML RT [8] and in Promela [9], the input language of the model checker SPIN [6]. We have also developed a method to estimate conservative upper bounds for the runtime filling of individual communication buffers in that work.

The boundedness semi-test that we devised is sound with respect to the boundedness of a given CFSM model. However, in case boundedness cannot be established, the test will return a verdict of "UNKNOWN". In that event the test also returns a "counterexample" which consists of a collection of control flow cycles in the state machines that collectively may lead to an unbounded flooding of at least one message buffer in

the system. Our test is also based on a gross abstraction of the original CFSMs – we abstract from transition triggers, transition code, message orders, the activation conditions of cycles, and cycle dependencies. As a consequence, the UNKNOWN verdict may be based on false negatives, i.e., the combinations of cycles that the test believes to be possible candidates for an unbounded flooding of message buffers may not be executable in the concrete model.

It is the objective of this paper to suggest methods that give the user automated support to a) determine, whether a counterexample is spurious or not, and b) to refine the abstraction in case a counterexample was found to be spurious. We will apply our method to models given in Promela, which is a modelling language that is very often used to model CFSM systems, in particular communication protocols. While Promela was a convenient choice, the ideas and concepts that we develop have wider applicability in the realm of concurrent, message based modeling and programming languages.

We employ methods from program analysis, in particular control flow and variable analysis to determine executability conditions for the cycles included in a counterexample. We have implemented our refinement method in the IBOC boundedness analysis tool and have applied our approach to various realistic and real-life Promela models.

Related Work. In our paper we follow the general idea of iterative counterexample guided abstraction refinement that was proposed in [3] and later applied to software model checking [1]. Since we use different code abstraction techniques the abstraction refinements that we propose are not comparable to the refinements proposed in those papers. In the context of linear programming based model checking the authors of [10] propose abstraction refinements based on the analysis of structural characteristics of control flow graphs. However, they do not address the imprecision that is caused by the abstraction of program code in concrete models.

Structure of the Paper. In Section 2 we review our previously published boundedness test and illustrate its application to Promela. In Section 3 we discuss the different sources of imprecision that lead to spurious counterexamples. Section 4 describes the static program analysis that we perform in order to abstract Promela code and illustrates how the abstraction can be refined in the presence of spurious counterexamples. We also employ graph structural criteria on the control flow cycle graphs to determine spuriousness and derive abstraction refinements, which we discuss in Section 5. In Section 6 we consider the complexities of the spuriousness detection and refinement method, and in Section 7 we present our experimental results.

2 Overview of the Boundedness Test

We now review the boundedness test for CFSM models first published in [8] and [9] and motivate the occurrence of spurious counterexamples. We review the test using Promela encodings of the CFSMs.

The objective of the boundedness test is to reduce the message passing behavior of a set of concurrent Promela processes to a number of control flow cycles. Each such cycle is then mapped to an effect vector that captures the number of messages sent

minus the number of messages received for every message type in the system when executing that particular control flow loop. Linear inequality solving is then used on the resulting system of effect vectors to determine whether there is a linear combination of the vectors that leads to an unbounded "blow-up" of the number of messages of at least one message type in the system. If such a linear combination does not exist, we know that the system is bounded (test outcome BOUNDED). If a linear combination can be found, i.e., the linear inequality system has a solution, the system under consideration may be bounded or not (test outcome UNKNOWN). In this case, a counterexample is constructed from the particular solution to the linear inequality system.

```
mtype = {a, b};
chan ch1 = [1] of {mtype};
chan ch2 = [1] of {mtype};
active proctype Left(){
   do
   :: ch2 ? b -> ch1 ! a;
   od
}
```

```
active proctype Right(){
   byte x;
   x = 0;
   do
   :: (x == 0) -> ch2 ! b; x = 1;
   :: (x == 1) -> ch1 ? a; x = 0;
   od
}
```

Fig. 1. A simple Promela model.

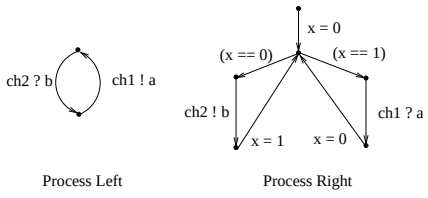

Fig. 2. The state machines of the Promela model in Figure 1.

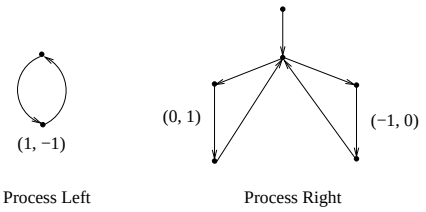

Fig. 3. The state machines with effect vectors of the Promela model in Figure 1.

Consider the simple Promela model in Figure 1. The abstraction we perform in our test will first produce an extended CFSM model as in Figure 2. By ignoring variables and transition code this will be further abstracted into a system of state machines with effect vectors as illustrated in Figure 3. The three cycles entail the effect vectors (1, -1), (0, 1) and (-1, 0) which gives rise to the following system of integer linear inequalities: (1) $-x_1 + x_2 \geq 0$, (2) $x_1 - x_3 \geq 0$, and (3) $x_2 - x_3 > 0$. Note that the third equation is used to ensure that at least one component in the summary effect vector of each linear combination of cycle effect vectors attains a truly positive value.

The test will return a solution, in fact a linear combination with the values $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$, and hence an outcome of UNKNOWN. Obviously, in the abstraction the left cycle of the process `Right` alone can cause the message buffer to blow up, and this is what the solution indicates. However, as it is easy to see, in the original Promela model an unbounded execution of this cycle without intervention of

the other cycles is not possible, which is ensured by the use of the (abstracted) program variable x.

Note that each cycle in the abstract system has a corresponding variable in the integer linear programming (ILP) problem into which the above system of inequalities is translated. This variable represents the number of executions of the respective cycle in a solution of the ILP problem if the ILP problem has a solution, i.e., if boundedness cannot be established. We call a set of cycles whose corresponding variables receive positive values in a solution to the ILP problem a *counterexample*. A counterexample represents a behavior of the system in which only the cycles in the counterexample are repeated infinitely often. Any other cycle in the system is either repeated only a finite number of times or not executed at all. A counterexample is said to be *spurious* if the behavior that it denotes is not a valid execution of the original model. In the above example, the test has found a spurious counterexample. It is the objective of the work in this paper to automatically detect spurious counterexamples generated by the boundedness test, and to refine the Promela model abstraction to exclude the spurious counterexamples that have been detected.

3 Sources of Imprecision

In this section we study the causes for the introduction of spurious counterexamples in our boundedness test. We also examine to what extent each cause affects the precision of the boundedness test.

3.1 Counterexamples and Spuriousness

The introduction of spurious counterexamples is a consequence of the conservative abstraction steps that we perform in the course of our boundedness test. We reconsider each of these abstraction steps to examine which information is removed from models during the step and how significant it affects the precision of the boundedness test. Note that these abstraction steps are conceptual and do not correspond to the concrete abstraction steps that the IBOC tool performs.

Step 1: Code Abstraction. In this step the program code in a model is abstracted away. The resulting CFSM system retains only the finite control structure and the message passing behavior of the model. We lose all the information about how the behavior of the model is constrained by the conditions on variables that are imposed by the program code. Losing such information is very significant because it often depends on the runtime value of a variable whether to send or receive a message, which message to send or receive, where messages are to be sent or from where messages are to be received. We will therefore consider this source of imprecision in more detail.

Step 2: Abstraction from Message Orders. In this step we neglect all information regarding the order of messages in message buffers. In particular, we assume that a message is always available to trigger a transition wherever it is in the buffer. This can be too coarse an overapproximation for a model that employs strict first-in-first-out message

buffers. However, models in practice usually have a message deferral/recall mechanism that stores an arriving message which cannot immediately be processed by the system into a special buffer so that it can be recalled when it is later needed. This is consistent with the semantics of our abstraction. In other words, this abstraction step does not introduce imprecision in most practical situations and we will therefore not address it in this paper.

Step 3: Abstraction from Activation Conditions. In this step the activation conditions of control flow cycles are abstracted away. We assume that there are always enough messages of the right type available for a cycle to be reachable from the initial configuration of the model. This assumption is reasonable in practice since an unreachable cycle has no influence on the system behavior and usually indicates a design error. We will therefore not consider this source of imprecision either.

Step 4: Abstraction from Cycle Dependencies. In this step we abstract from implicit dependencies between control flow cycles. We consider different types of such dependencies: exclusion dependencies or inclusion dependencies, global dependencies or local dependencies. An exclusion dependency forbids a set of cycles to be jointly repeated infinitely often, while an inclusion dependency stipulates the need of some cycles being repeated infinitely often to enable other cycles to be repeated infinitely often. A global dependency specifies a dependency among cycles in multiple processes, while a local dependency relates cycles in one process. Cycle dependencies are imposed either by the program code executed while the program goes through a cycle, or by structural characteristics of the control flow graphs. Disregarding cycle dependencies means that arbitrary cyclic executions can be combined to form a potentially spurious counterexample, which is why we will further address this source of imprecision.

3.2 Boolean Conditions on Cycles

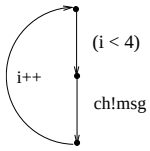

Fig. 4. A simple cycle.

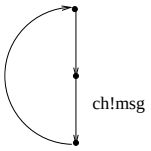

Fig. 5. The abstract cycle.

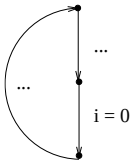

Fig. 6. A cycle resetting i.

We now consider the impact of the abstraction of boolean executability conditions during code abstraction. As an example, consider the cycle shown in Figure 4[1]. The cycle guard checks whether the runtime value of a local integer variable i is less than 4. If the condition is satisfied, then a message `msg` is sent to the channel `ch`, and i is

[1] All program code in examples will be given in Promela syntax.

incremented. After the code abstraction, the resulting abstract cycle in Figure 5 retains only the message sending statement, and all the information about i has been removed. The abstract cycle will inevitably by itself form a counterexample in which a message `msg` is sent to `ch` without any constraint each time the cycle is executed. Apparently, this counterexample is spurious since the cycle can be repeated without interruption at most 4 times, if i is initialized to 0. If we want the cycle to be repeated later, then some other cycle in the control flow graph of the same process in which the value of i will be changed back to be less than 4 needs to be executed. Figure 6 shows such a cycle in which the value of i is reset to 0. If this cycle is the only other cycle that modifies i, then the boolean condition on the cycle in Figure 4 imposes an *inclusion* dependency between these two cycles. In other words, if the cycle in Figure 4 appears in a counterexample, then the cycle in Figure 6 must be included in the same counterexample.

3.3 Graph-Structural Dependencies

Graph-structural dependencies are another source of cycle dependencies that may help to reveal spurious counterexamples. Consider two cycles in the control flow graph of one same process that reside in two different strongly connected components. Since at least one cycle is not reachable from the other, they cannot be jointly repeated infinitely often. Strongly connected components induce *exclusion* dependencies.

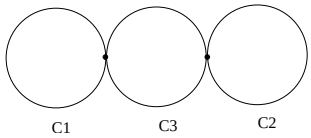

Fig. 7. Three cycles.

There is another kind of graph-structural cycle dependencies. Consider the control flow graph of a process as shown in Figure 7. Assume that a counterexample contains the cycles `C1` and `C2` and does not contain the cycle `C3`. Note that `C1` and `C2` do not share a common state, which implies that it is impossible to repeat `C1` and `C2` infinitely often without repeating `C3` infinitely often. Since `C3` is not included in the above counterexample it is spurious. The resulting dependency is an *inclusion* dependency.

Summary. We recognize the following two types of information as being crucial to the precision of our unboundedness test: (1) cycle dependencies imposed by the boolean conditions on cycles, and (2) cycle dependencies imposed by structural characteristics of control flow graphs. In the next two sections, we will propose several automated refinement methods based on counterexample spuriousness analyses with respect to these two types of information.

4 Cycle Code Analysis

It is generally impossible to precisely determine cycle inclusion dependencies. Our method is therefore incomplete and it overapproximates the actual inclusion dependencies. As a consequence, (1) a counterexample can still be spurious even if it does not violate the determined dependency, and (2) some spurious counterexamples may never be excluded.

A cycle in a control flow graph corresponds to a loop in the program code of the respective process. The task of a loop is either to continuously react to stimuli from the environment, or to perform some local information processing task. As mentioned above, we also distinguish global and local dependencies. Local dependencies are determined by cycle conditions in which all the variables are locally modified. This excludes the use of local variables to store the contents of messages since this would imply global dependencies. In this paper we will focus solely on local dependencies and leave the treatment of global dependencies for future work.

Our approximative method to determine inclusion dependencies is only applicable to loop code that follows a certain syntactic pattern:

- There exists at least one branch statement or loop statement in the loop with a guarding boolean expression in which all the variables are only locally modified within the loop. We call a variable occurring in the guard of a branch statement or a loop statement a *control variable*.
- The computation on control variables involves only linear expressions over integers, such as an incrementation or a decrementation. The guards of branch statements and loop statements are boolean conditions that involve only comparisons of linear expressions over control variables.

We notice that this is not overly constraining since code following this pattern is commonly used in real models. We further assume that all the program statements are side-effect free, and that all function calls have been inlined.

4.1 Constraints on Repeated Cycle Executions

A condition statement in a cycle functions as the guard on the executability of a statement within a loop. Since program code can have nested loops, a cycle can have several condition statements that guard the loop statements at different depths. We study how the repeated executions of a cycle are constrained by each of the condition statements in the cycle.

We denote a condition statement by a boolean expression enclosed in a pair of parentheses. Consider a condition statement `(`B`)` in a cycle `C1`. `(`B`)` is executable if and only if B evaluates to `true` under the current valuation of variables. We are interested in determining max_B, the maximal number of times that `(`B`)` can be repeatedly executed if all variables in B are only modified within `C1`. If max_B exists, then we can draw the following conclusions: (1) `C1` cannot be consecutively repeated more than max_B times. (2) For every max_B times that `C1` is repeated, at least one of the *neighboring* cycles of `C1` has to be executed. A cycle is a neighboring cycle of another cycle

if they share common states. (3) For every max_B times that `C1` is repeated, at least one of the *supplementary* cycles with respect to B has to be executed. A supplementary cycle of `C1` with respect to B is a cycle that modifies at least one of the variables in B in a way that renders B satisfied. The conclusions (2) and (3) impose an inclusion cycle dependency that can be used to determine counterexample spuriousness and to refine the abstract system of the original model. However, it is generally impossible to precisely determine max_B and we will hence resort to computing an upper bound for max_B. It is easy to see that this is a safe approximation. For any number n, the restriction that other cycles have to be executed every n times that `C1` is executed does not exclude the possibility that other cycles are executed every m times that `C1` is repeated for any $m < n$.

We notice that a boolean expression can be converted into its negation free disjunctive normal form, where each disjunct is a conjunction of one or more positive atomic propositions. Because the boolean conditions that we consider involve only comparisons of linear expressions, an atomic proposition is a linear (in)equality. In the sequel we assume that any boolean expression is given in this type of normal form.

For each disjunct d of B, we denote by $max_{B,d}$ the maximal number of times that d is satisfied when `(`B`)` is repeatedly executed without the variables in B being modified outside `C1`. If we can determine $max_{B,d}$ values, then max_B is bounded by the sum of all the $max_{B,d}$ values. For each (in)equality l of d, we denote by $max_{B,d,l}$ the maximal number of times that l is satisfied when `(`B`)` is repeatedly executed without the variables in B being modified outside `C1`. We can further reduce the computation of $max_{B,d}$ to the computations of $max_{B,d,l}$ values. $max_{B,d}$ is the minimum of all the $max_{B,d,l}$ values, because if any (in)equality l of d is not satisfied then d is not satisfied.

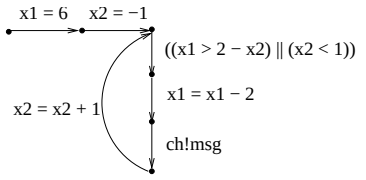

Fig. 8. Cycle `C`.

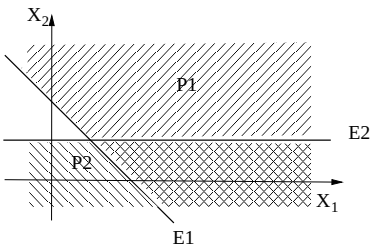

Fig. 9. Convex polyhedra.

Consider as an example the cycle `C` in Figure 8. x_1 and x_2 are integer variables. Before `C` is entered, x_1 is initialized with 6 and x_2 is initialized with -1. During each cycle execution, the value of x_1 is decremented by 2 and the value of x_2 is incremented by 1. Let $B = (x_1 > 2 - x_2) \vee (x_2 < 1)$, $d_1 = l_1 = (x_1 > 2 - x_2)$, and $d_2 = l_2 = (x_2 < 1)$. It can be manually determined that $max_{B,d_1,l_1} = 3$, $max_{B,d_1} = 3$, $max_{B,d_2,l_2} = 2$, $max_{B,d_2} = 2$, and $max_B = 3$. An upper bound of max_B is 5 $(3+2)$, which is larger than 3 because d_1 and d_2 can be both satisfied at same time during cycle executions. We conservatively take 5 to overapproximate the actual max_B value 3.

Before we explain the method to compute $max_{B,d,l}$, we give a geometric illustration of the problem. In the example from Figure 8 each of B's disjuncts is a system of linear (in)equalities that defines a convex polyhedron in the 2-dimensional Euclidean space as shown in Figure 9. The polyhedron `P1` is defined by the disjunct d_1 and the polyhedron `P2` is defined by the disjunct d_2. B is the union of all the polyhedra defined by B's disjuncts. The (in)equalities of a disjunct define the edges of the polyhedron defined by the disjunct. For instance, the edge `E1` is defined by the inequality $l_1 = (x_1 > 2 - x_2)$. `E1` can be represented by the equation $x_1 + x_2 = 2$, in which no variables occur in the right-hand side. We call $x_1 + x_2$ the *control expression* of l_1, and 2 the *boundary value* of l_1.

Let (x_1, x_2) denote the point defined by x_1 and x_2 in the Euclidean space. During repeated executions of `C`, (x_1, x_2) moves in the space while x_1 and x_2 are modified. For a disjunct d of B, $max_{B,d}$ is the same value as the maximal number of computation steps that (x_1, x_2) can stay in the polyhedron defined by d. $max_{B,d}$ is therefore bounded if and only if the following *exiting property* is satisfied: at some point of time (x_1, x_2) will exit the polyhedron and never return. In the sequel we show a sufficient but not necessary condition of the exiting property.

During repeated execution of `C` we obtain a sequence of vectors (x_1^0, x_2^0), (x_1^1, x_2^1), ... that denotes the values of x_1 and x_2 at the end of each execution of `C`. Given an edge `E` of a polyhedron `P`, we assume that `E` is defined by an (in)equality l. Let $ce(l)$ denote the control expression of l and $bv(l)$ denote the boundary value of l. Let $ce(l)^i$ denote the value of $ce(l)$ when $x_1 = x_1^i$ and $x_2 = x_2^i$, where i is a nonnegative integer. Assume that the sequence $ce(l)^0$, $ce(l)^1$, ... is either strictly increasing or strictly decreasing, i.e., $ce(l)$ is modified monotonically. Let $d^i = |ce(l)^i - bv(l)|$ denote the distance between $ce(l)^i$ and the boundary value of l. If the sequence d^0, d^1, ... is strictly decreasing, then the point (x_1, x_2) is always moving closer to the edge `E`. Furthermore, the sequence d^0, d^1, ... must be finite. The sequence (x_1^0, x_2^0), (x_1^1, x_2^1), ... is then also finite, which implies that (x_1, x_2) can only stay in `P` for a finite number of computation steps. Note that max_{B,d_1,l_1} is identical to the length of the sequence of distances d^i. We conclude that under the above assumption the exiting property holds for `P` and we call `E` an *exit border* of `P`. Obviously, both `E1` and `E2` are exit borders of the respective polyhedron. The existence of an exit border is a sufficient condition for the exiting property to hold.

4.2 Computing $max_{B,d,l}$ Values

Given a cycle `C1`, let `(`B`)` denote one of the condition statements in `C1`, d a disjunct of B and l an (in)equality of d. We now address the question how to compute $max_{B,d,l}$.

Remember that l defines an edge E of the polyhedron that d defines. E can be represented as the equation $ce(l) = bv(l)$. In case that $ce(l) = bv(l)$ is an exit border, an upper bound of the maximal number of computation steps that are needed for the value of $ce(l)$ to reach $bv(l)$ can be determined, (1) if we can determine the possible differences of the runtime values of $ce(l)$, i.e., the step values of $ce(l)$, before and after each execution of `C1`, and (2) if we can determine the initial values of $ce(l)$ before `C1` is entered and therefore the longest distance to the boundary value. We determine whether $ce(l) = bv(l)$ is an exit border by determining and analyzing the step values of $ce(l)$.

Determining Step Values. The cycle code imposes a relation constraining the values of variables before and after an execution of the cycle. Determining bounds for the step values of the control expression $ce(l)$ can be seen as an optimization problem. We generate a set of ILP problems from the cycle code, whose constraints reflect how the runtime values of variables are changed and constrained by the cycle code. The objective functions of these ILP problems are either the maximum or the minimum of the step values of $ce(l)$. The generation of the constraints of an ILP problem for determining step values involves the transformation of each program statement in the cycle code into one or more (in)equations. Since not all the statements in the cycle code are relevant to the computation on the control variables in l, we compute a slice [12] of the cycle code with the slice criterion being $< (B), \bar{x} >$, where $\bar{x}$ are all the variables in l. Since we have the assumption that all the program statements are side-effect free, the transformation of program statements is straightforward. As an example, the assignment statement `x := x + 1` is transformed to the equation $x_{i+1} = x_i + 1$. The variable x_i denotes the runtime value of the variable x before the execution of the assignment, and the variable x_{i+1} denotes the runtime value of x after the execution. When a receive statement is executed, a variable x in the statement receives a random value that depends on the content of the received message. In the corresponding equation, we must represent the new value of x by a newly introduced variable. Moreover, when an array member is affected by an assignment statement or a receive statement, we take the conservative assumption that any member of the array may be affected.

```
max: x1_1 + x2_1 - x1_0 - x2_0;

x1_0 > 2 - x2_0;
x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;
```

```
min: x1_1 + x2_1 - x1_0 - x2_0;

x1_0 > 2 - x2_0;
x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;
```

```
max: x1_1 + x2_1 - x1_0 - x2_0;

x2_0 < 1;
x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;
```

```
min: x1_1 + x2_1 - x1_0 - x2_0;

x2_0 < 1;
x1_1 = x1_0 - 2;
x2_1 = x2_0 + 1;
```

Fig. 10. The generated integer programming problems.

Consider the example in Figure 8. The cycle code is abstracted into the four ILP problems in Figure 10 for determining step values for the control expression $x_1 + x_2$ of the inequality $x_1 > 2 - x_2$. The top two ILP problems are used to determine the bounds on step values under the condition $(x_1 > 2 - x_2)$ as the first disjunct of the guard $(x_1 > 2 - x_2) \vee (x_2 < 1)$, while the other two ILP problems are used to determine the bounds under the condition $(x_2 < 1)$ as the second disjunct of the same guard.

Analyzing Step Values. The solutions to the objective functions of the ILP problems for determining the step values of $ce(l)$ set an upper bound and a lower bound on the actual step values. If the two bounds have different signs, then the value of $ce(l)$ may not be modified monotonically. If one of the two determined bounds is 0, then the value of $ce(l)$ may be unchanged forever. In these two cases, we fail to determine whether

$ce(l) = bv(l)$ is an exit border, and conservatively set $max_{B,d,l}$ to be unbounded. If the upper bound and the lower bound are both positive, then the value of $ce(l)$ is always increased. We conservatively take the lower bound as the constant step value of $ce(l)$ that denotes the slowest move of $ce(l)$. Similarly, if the upper bound and the lower bound are both negative, then we take the upper bound to be the constant step value. For the ILP problems in Figure 10, all the solutions are -1, which is then both the upper bound and the lower bound determined for the step values of the control expression $x_1 + x_2$. The constant step value determined for $x_1 + x_2$ is then -1.

Let $op(l)$ denote the *comparison operator* in l. We determine whether $ce(l) = bv(l)$ is an exit border according to $op(l)$ and the determined constant step value of $ce(l)$. If $op(l)$ is $=$, then $ce(l) = bv(l)$ is an exit border whatever the constant step value is, because any nonzero step value annuls the satisfaction of l. In this case we directly set $max_{B,d,l}$ to 1. Let $\bar{x}$ be the vector of the variables in B. If the constant step value of $ce(l)$ is positive, and if op is $<$ or $\leq$, then in the polyhedron defined by d the point defined by $\bar{x}$ is moving closer to the edge $ce(l) = bv(l)$. $ce(l) = bv(l)$ is then an exit border. Similarly, if the constant step value is negative, then $ce(l) = bv(l)$ is an exit border when op is $>$ or $\geq$. In all other cases, $ce(l) = bv(l)$ is not an exit border and we set $max_{b,d,l}$ to be unbounded. In the example in Figure 8, we have determined the constant step value of $x_1 + x_2$ to be -1. The comparison operator of the corresponding inequality is $>$. The edge $x_1 + x_2 = 2$ is therefore an exit border.

Determining Initial Values. If we determine $ce(l) = bv(l)$ to be an exit border, and if $op(l)$ is not $=$, then we need to determine the initial values of $ce(l)$ in order to know the longest distance from the initial values of $ce(l)$ to the boundary value. We use a simple solution that employs backward depth first searches to check each acyclic path leading to one of the entry locations of `C1` whether $ce(l)$ receives a constant value on that path before `C1` is entered. Our solution to determining initial values is certainly incomplete, but also more efficient than other approaches, such as interval analysis [4]. This is because our solution does not require the analysis of the whole control flow graph of the respective process. An interval analysis will cost even more when any variable in $ce(l)$ is affected somewhere outside `C1` by a received message sent from some other process. The sending process of that message has then to be analyzed even if the receiving of the message will not affect the initial value of $ce(l)$. On the other hand, we will show later that, even in the absence of the determined initial values of $ce(l)$, we may still be able to determine an inclusion dependency.

Computing $max_{B,d,l}$. Let $step$ denote the determined constant step value of $ce(l)$ and $distance$ the longest distance between $ce(l)$ and $bv(l)$. $\lceil \cdot \rceil$ is the ceiling function that returns the smallest integer greater than the input real number. $\lceil distance/step \rceil$ sets an upper bound on $max_{B,d,l}$ that is used to safely approximate the actual $max_{B,d,l}$ value. In the example in Figure 8, we determine $max_{(x_1>2-x_2)\vee(x2<1),x_1>2-x_2,x_1>2-x_2}$ as follows. The initial value of $x_1 + x_2$ is 5. $step = -1$ and $distance = 2 - 5 = -3$. The determined upper bound is then 3 ($-3 \div -1$).

4.3 Abstraction Refinement

For a condition statement (`(`B`)`) in a cycle `C1`, if we can determine an approximative max_B value, then we know that one of the neighboring cycles of `C1` and one of the supplementary cycles of `C1` with respect to B have to be executed every max_B times `C1` is executed. A counterexample containing `C1` is determined to be spurious, if (1) no neighboring cycle of `C1` is included in the counterexample, or if (2) no supplementary cycle of `C1` with respect to B is included in the counterexample.

However, it is generally impossible to determine the exact set of supplementary cycles with respect to B, since it is generally impossible to determine whether the execution of a cycle has the effect to render B satisfied. A simple overapproximation is to consider a cycle a supplementary cycle if it resides in the same process as `C1` and if it modifies at least one of the variables in B.

Refinement with max_B. We use max_B, the set of neighboring cycles, and the set of supplementary cycles to refine the abstract system of the original model. We first add to the boundedness test ILP problem one additional constraint which enforces that every max_B times that `C1` is executed least one of its neighbors will be executed. Assume that the set of neighboring cycles of `C1` is NC. The added constraint is: $c_1 \leq max_B \times \sum_{C_i \in NC} c_i$, where c_1 is the variable corresponding to `C1` and c_i is the variable corresponding to `C`$_i$. A similar constraint is added to the boundedness test ILP problem. It requires at least one supplementary cycle to be executed each max_B times `C1` is executed.

Refinement without max_B. Let $\bar{x}$ denote the vector of the variables in B. If all disjuncts of B contain an (in)equality that defines an exit border, then we know that the number of computation steps for which the point defined by $\bar{x}$ can stay in the polyhedron of each disjunct is always finite. In this case, even if max_B cannot be determined, we can still determine an inclusion dependency. After a finite but unknown number of times `C1` is executed, `C1` will be exited, and one of its neighboring cycles as well as one of its supplementary cycles have to be executed. Let SC be the set of supplementary cycles of `C1` with respect to B. In the refinement without max_B, the boundedness test ILP problem is replaced by two ILP problems, each augmenting the original ILP problem with one of the following constraints: (1) $c_1 = 0$, or (2) $c_1 > 0 \wedge \sum_{C_i \in NC} c_i > 0 \wedge \sum_{C_i \in SC} c_i > 0$, where c_1 is the variable corresponding to `C1` and c_i is the variable corresponding to `C`$_i$. These two constraints stipulate that (1) either `C1` is not repeated infinitely often, or (2) at least one of the neighboring cycles and at least one of the supplementary cycles have to be repeated infinitely often. The two newly generated ILP problems partition the behavior of the refined abstract system into two disjoint subsets. The original model can be determined to be bounded if both of the two ILP problems are infeasible.

During experiments we found that this approximative way of determining supplementary cycles is rather coarse for some models in that it prevents many spurious counterexamples from being excluded. Consider the cycle in Figure 8. After the condition $(x_1 > 2 - x_2) \vee (x_2 < 1)$ is not satisfied and `C` is exited, assume that some supplementary cycles are going to be executed to make the condition satisfied again. Assume

a neighboring cycle C′ of C, in which no message is sent or received. If each execution of C′ decreases the value of x_1 and increases the value of x_2, then C′ has the same effect on x_1 and x_2 as C does. In particular, it cannot render $(x_1 > 2 - x_2) \vee (x_2 < 1)$ satisfied. However, C′ is regarded a supplementary cycle of C. The refinement of the abstract system with the determined set of supplementary cycles including C′ does not exclude the spurious counterexample consisting of C and C′.

We adopt a finer solution if, for every (in)equality l in B, $ce(l)$ is modified monotonically within C1. In this case, we regard a cycle C2 a supplementary cycle with respect to B, if C2 satisfies the following condition. There exists an (in)equality l in B that defines an exit border. The value of $ce(l)$ is increased within C2 if it is decreased within C1. The value of $ce(l)$ is decreased within C2 if it is increased within C1. This solution is more expensive since it involves code analysis for each cycle that modifies one or more variables in B, but is more precise in determining spuriousness.

5 Graph-Structural Dependency Analysis

In this section we consider two types of cycle dependencies, namely those imposed by strongly connected components and those imposed by direct connectedness of cycles, and propose appropriate abstraction refinements.

Given a counterexample in which two cycles in one same process do not share common states, some other cycles in the same process have to be included in the counterexample to "bridge" them. We introduce the concept of *self-connected cycle set*. A set of cycles is self-connected if any two cycles in the set are reachable from each other by traversing through only the cycles in the set. A counterexample is spurious if, for some process P, the set of all the cycles of P in the counterexample is not self-connected.

We propose the following refinement. Given a counterexample, if there are two cycles C1 and C2 of one same process P in the counterexample that are not reachable from each other by traversing through only the cycles in the counterexample, then we determine all the self-connected sets of cycles of P that contain both C1 and C2. If no such set exists, then C1 and C2 are in different strongly connected components. Let c_1 be the variable corresponding to C1 and c_2 be the variable corresponding to C2. We replace the boundedness test ILP problem with three ILP problems, each of which augments the original ILP problem with one of the following three constraints: (1) $c_1 = 0 \wedge c_2 = 0$; (2) $c_1 = 0 \wedge c_2 > 0$; (3) $c_1 > 0 \wedge c_2 = 0$. These three constraints prevent C1 and C2 from being both repeated infinitely often.

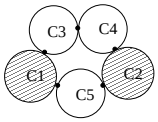

Fig. 11. A control-flow graph.

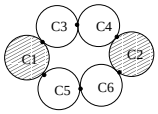

Fig. 12. A control-flow graph.

If there exists at least one self-connected set of cycles containing both C1 and C2, then C1 and C2 are in the same strongly connected component. Let n denote the number of determined self-connected cycle sets. To refine the abstract system, we generate $n+3$ new ILP problems to replace the original boundedness test ILP problem. Three of them are the same ILP problems as generated in case C1 and C2 belong to different strongly connected components. The other n ILP problems stipulate that, if C1 and C2 are both repeated infinitely often, then all the cycles in one of the determined self-connected cycle sets have to be repeated infinitely often. For each determined self-connected cycle set S_i ($1 \leq i \leq n$), a new ILP problem is generated by augmenting the original ILP problem with the following constraint: $c_1 > 0 \wedge c_2 > 0 \wedge \bigwedge_{C \in S_i} c > 0$, where c is the variable corresponding to the cycle C. As an example, we assume that the control-flow graph of P is the same as shown in Figure 11. All the self-connected sets of cycles containing C1 and C2 are $S_1 = \{$C1, C3, C4, C2$\}$ and $S_2 = \{$C1, C5, C2$\}$. Let c_i ($3 \leq i \leq 5$) denote the variable corresponding to the cycle Ci. Each of the 5 newly generated ILP problems has one of the following constraints: (1)$c_1 = 0 \wedge c_2 = 0$; (2) $c_1 = 0 \wedge c_2 > 0$; (3) $c_1 > 0 \wedge c_2 = 0$; (4) $c_1 > 0 \wedge c_2 > 0 \wedge c_3 > 0 \wedge c_4 > 0$; (5) $c_1 > 0 \wedge c_2 > 0 \wedge c_5 > 0$.

A caveat of this method is that the number of newly generated ILP problems can be exponential in the number of the cycles of P. We propose an alternative method that only generates 4 ILP problems each time the abstract system is refined. Consider the control-flow graph in Figure 11. To reach C2 from C1, one of the neighboring cycles of C1 must be entered. Similarly, one of the neighboring cycles of C2 must be entered in order to reach C1 from C2. Instead of generating the two ILP problems corresponding to the two self-connected cycle sets containing C1 and C2, we can build only one ILP problem by augmenting the original boundedness test ILP problem with the following constraint: $c_1 > 0 \wedge c_2 > 0 \wedge c_3 + c_5 > 0 \wedge c_4 + c_5 > 0$. To generalize this idea we assume that two cycles C and C′ in a counterexample are not reachable from each other by traversing only the cycles in the counterexample. We compute a sequence of pairs of cycle sets: $< N_C^0, N_{C'}^0 >$, $< N_C^1, N_{C'}^1 >$, ..., $< N_C^n, N_{C'}^n >$. N_C^0 is the set of neighboring cycles of C. N_C^{i+1} contains all the cycles that neighbor some cycle in N_C^i and that are not in any N_C^j if $j \leq i$. $N_{C'}^i$'s ($0 \leq i \leq n$) are defined likewise. The computation of the sequence $< N_C^0, N_{C'}^0 >$, ..., $< N_C^n, N_{C'}^n >$ terminates if, for some number n, either (1) N_C^n or $N_{C'}^n$ is empty, or (2) there is a cycle in N_C^n and a cycle in $N_{C'}^n$ such that these two cycles are neighbors. If the first condition is true, then C and C′ must be in different strongly connected components. The treatment in this case is the same as in the previous refinement method. If the second condition is true, then we know that, to reach C′ from C, one cycle from each set in the sequence N_C^0, N_C^1, ..., N_C^n, $N_{C'}^n$, $N_{C'}^{n-1}$, ..., $N_{C'}^0$ has to be entered. A similar requirement is for reaching C from C′. To refine the abstract system, we replace the boundedness test ILP problem with 4 newly generated ILP problems. Three of them augment the original ILP problems with the constraints that prevent C and C′ from being both repeated infinitely often. The last ILP problem augments the original ILP problem with the constraint which stipulates that, if C and C′ are both repeated infinitely often, then at least one cycle from each set in the sequence N_C^0, N_C^n, ..., $N_{C'}^1$, ..., $N_{C'}^n$ has to be also repeated infinitely often. Note that, although this method generates fewer ILP problems, it is coarser in that

some spurious counterexample may not be excluded. Consider the control flow graph in Figure 12. Let c_i ($1 \leq i \leq 6$) be the variable corresponding to the cycle `Ci`. If a counterexample contains `C1` and `C2` and does not contain any of other cycles in the same control flow graph, then a constraint is added to one of the newly generated ILP problems as following: $c_1 > 0 \wedge c_2 > 0 \wedge c_3 + c_5 > 0 \wedge c_4 + c_6 > 0$. This constraint cannot exclude a potentially spurious counterexample that contains `C1`. `C2`, `C3`, `C6`, and not others in the same control flow graph.

6 Complexity

It is an undecidable problem to determine whether a counterexample generated in the boundedness test is spurious or not. The proposed refinement method in this paper is incomplete, and inevitably has a high theoretical complexity due to the following facts. (1) The number of generated ILP problems for determining the step values of a control expression is exponential both in the size of the respective cycle and in the size of the guard boolean expression of each condition statement in the cycle. (2) Solving an ILP problem is NP-complete.

Despite a high theoretical complexity, the proposed refinement method is efficient in practice. The reasons are the following. (1) A cycle usually has a small portion relevant to the computation of the control variables. (2) A guard boolean expression of a condition statement is usually not complex such that its negation free disjunctive normal form contains only a small number of disjuncts. (3) A generated ILP problem for determining step values is usually very small.

7 Experimental Results

We implemented the spuriousness determination and counterexample refinement methods that we propose in the IBOC system . We report the experimental results of analyzing the following models using IBOC on a Pentium IV 3.20GHz machine with 2GB memory: the *Sort* model included in the SPIN [6] 4.22 distribution package, a model of the *General Inter-ORB Protocol* (GIOP) [7], and a model of the *Model View and Concurrency Control Protocol* (MVCC) [11].

The *sort* model consists of 8 running processes that collaboratively sort 7 numbers by exchanging messages through 7 buffers. IBOC used 0.718 second to report a counterexample. The counterexample consists of only one cycle from the `Left` process that sends one of the 7 numbers to be sorted to the buffer `q[0]` in their initial order each time the cycle is executed. The cycle is guarded by the condition $counter < 7$ in which the variable `counter` receives the value 0 before the cycle is entered. Each execution of the cycle increments `counter` by 1. IBOC used another 0.688 second to determine that the cycle cannot be repeated consecutively more than 7 times, and that the cycle has neither neighboring cycles nor supplementary cycles to modify `counter`. IBOC therefore determined the counterexample to be spurious, and excluded the cycle from the abstract system. IBOC found no more counterexamples. The *sort* model was determined to be bounded.

The GIOP protocol supports message exchange and server object migration between object request brokers (ORBs) in the CORBA architecture. The Promela implementation [7] that we took is a real life model with considerable size and complexity. IBOC reported 5 counterexamples within 14.015 seconds. The first counterexample was determined to be spurious. IBOC failed to determine spuriousness for the second, the third, and the fourth counterexample because there is a cycle in each of them guarded by a boolean condition that involves a variable used to store the content of received messages. These conditions induce global cycle dependencies, which the proposed refinement method in this paper cannot handle. The last reported counterexample was manually determined to be spurious. The reason that IBOC failed on it is the following. When IBOC determines a cycle to be supplementary to an analyzed cycle, it does not consider how many times the supplementary cycle has to be executed in order to enable the analyzed cycle. However, considering such information may lead to an exponential number of new boundedness test ILP problems to be generated with respect to the number of supplementary cycles. Each of the ILP problems corresponds to one potential combinatory effect of supplementary cycles.

The MVCC protocol is one of the underlying protocols of the *Clock* toolkit [5] for the development of groupware applications. It supports multi-user server-client communication and the synchronization of concurrent updates of information. We took the Promela implementation in Appendix A of [11] that allows 2 clients. The model consists of 8 concurrently running processes. IBOC visited 70 states and 83 transitions in the state machines of the processes, and constructed 46 simple cycles and identified 16 types of messages. Within 4.281 seconds, IBOC found 4 counterexamples and determined the first 3 of them to be spurious. The last counterexample contains only one cycle in a `User` process (as a client) that sends a message without any constraints. It is a real counterexample.

8 Conclusion

In this paper we have presented abstraction refinement techniques for an incomplete communication channel boundedness test for Promela. Our approach allows one a) to determine spuriousness of counterexamples, and b) to refine the previous abstraction in order to exclude these spurious counterexamples. In order to determine spuriousness, we statically compute executability conditions for cycles and we analyse the cycles in the control flow graphs of the system to determine inclusion and exclusion dependencies. We implemented our spuriousness determination and the refinement method in IBOC. We presented experimental results that show that our method scales to systems of realistic size and is capable of returning meaningful results.

Further work includes the analysis of global cycle dependencies - the analysis of the GIOP protocol showed that a number of spurious counterexamples can not yet be detected due to the unavailability of this global analysis. A further goal is to apply our method to UML RT and UML 2.0 models that include state machine transitons attributed with Java code.

Acknowledgements. We thank Richard Mayr for initial discussions on the subject of this paper.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*. Springer Verlag, 2002.
2. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
4. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
5. T. C. Nicholas Graham, Tore Urnes, and Roy Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 1–10, New York, NY, USA, 1996. ACM Press.
6. G.J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.
7. M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using Promela and Spin. *Software Tools for Technology Transfer*, 2:394–409, 2000.
8. S. Leue, R. Mayr, and W. Wei. A scalable incomplete boundedness test for UML RT models. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2004*, Lecture Notes in Computer Science. Springer Verlag, 2004.
9. S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for buffer overflow of Promela models. In *Proc. of the International SPIN Workshop on Model Checking of Software SPIN 2004*, volume 2989 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
10. S. F. Siegel and G. S. Avrunin. Improving the precision of inca by eliminating solutions with spurious cycles. *IEEE Trans. Softw. Eng.*, 28(2):115–128, 2002.
11. M. H. ter Beek, M. Massink, D. Latella, and S. Gnesi. Model checking groupware protocols. In *COOP*, pages 179–194, 2004.
12. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.