A Scalable Incomplete Test for Message Buffer Overflow in Promela Models

Stefan Leue, Richard Mayr, and Wei Wei

Department of Computer Science
Albert-Ludwigs-University Freiburg
Georges-Koehler-Allee 51, D-79110 Freiburg, Germany
E-mail: [leue|mayrri|wwei]@informatik.uni-freiburg.de

Abstract. In Promela, communication buffers are defined with a fixed length, and buffer overflows can be handled in two different ways: block the send statement or lose the message. Both solutions change the semantics of the system, compared to one with unbounded channels. The question arises, if such buffer overflows can ever occur in a given system and what buffer lengths are sufficient to avoid them. We describe a scalable incomplete boundedness test for the communication buffers in Promela models, which is based on overapproximation and static analysis. We first reduce Promela models to systems of communicating finite state machines (CFSMs) and then apply further abstractions that leave us with a system of linear inequalities. Those represent the message sending and receiving effect that the control flow cycles of every process have on any message buffer. The test tries to establish the existence of a linear combination of the effect vectors so that at least one message can occur an unbounded number of times. If no such linear combination exists then the system is bounded. We discuss the complexity of this test and present experimental results using our implementation in the IBOC system. Scalability of the test is in part due to the fact that it is polynomial for the type of sparse control flow graphs derived from Promela models. Also, the analysis is local, i.e., it avoids the combinatorial state space explosion due to concurrency of the models. We also present a method to derive upper bound estimates for the maximal occupancy of each individual message buffer. Previously, we have applied this approach to UML RT models, while in this paper we focus on the additional problems specific to Promela code: determining the potential message types of any channel, tracking potential contents of variables, channels passed as arguments to processes, channel assignments, channel arrays and parallel process creation.

1 Introduction

In Promela, the input language of the SPIN model checker [7], inter-process communication can be done via shared global variables or by message passing via communication channels that operate as first-in first-out (FIFO) buffers. These buffers are defined with a fixed length, and buffer overflows (i.e., an attempt to send a message to a full buffer) can be handled by SPIN in two different ways: block the send statement or lose the message. Both solutions change the semantics of the system, compared to one with

unbounded channels. The question arises whether such buffer overflows can ever occur in a given system and what buffer lengths are sufficient to avoid them. Our paper presents an automated test for the occurrence of these buffer overflows in Promela.

Of course, possible buffer overflows can be detected by simulation, or by encoding this question into an LTL model checking problem. However, this normally involves fully exploring the state space. Here we propose a type of boundedness analysis that avoids exhaustively checking all the computations of the model. We describe a scalable incomplete boundedness test for the communication buffers in Promela models, which is based on overapproximation and static analysis. For the test, we first interpret all communication buffers in the model as having unbounded length (instead of the fixed length in their definition) and then try to prove their boundedness, i.e., to establish upper bounds on the maximal reachable occupancy of every buffer. To do this, we first reduce Promela models to systems of communicating finite state machines (CFSMs) and then apply further abstractions that leave us with a system of linear inequalities. Those represent the summary message sending and receiving effect that the control flow cycles of every process have on any message buffer. The test tries to establish the existence of a linear combination of the resulting effect vectors so that at least one message can occur an unbounded number of times. If no such linear combination exists then the system is bounded. By similar techniques it is also possible to derive upper bound estimates for the maximal occupancy of each individual message buffer.

Our test is: (i) Incomplete: Since boundedness for systems of CFSMs is undecidable [4] we work with an overapproximation of the Promela model. Hence, not every instance of a bounded system can be detected. (ii) Safe: If our test returns the result 'bounded' for the overapproximation then the original Promela model is also bounded. The computed upper bounds for maximal occupancy of each individual message buffer also carry over to the original Promela model. (iii) Scalable: Scalability of the test is in part due to the fact that it is polynomial for the type of sparse control flow graphs derived from Promela models. Also, the analysis is local, i.e., it avoids the combinatorial state space explosion due to concurrency of the models.

In precursory work [10] we have successfully applied this approach to boundedness checking of communication channels in UML RT [14, 15] models, using our implementation in the IBOC (*IMCOS Boundedness Checker*) tool that we are currently developing. Promela differs from UML RT in a number of important aspects. In UML RT the different parallel processes in the system are represented by so-called *capsules* which communicate with each other only by message passing. These message passing channels are a priori assumed to be unbounded and the topology of the communication structure is defined statically at compile time. The capsule behaviors are defined through hierarchical state machines whose transitions are triggered solely by message-receive events. These transition can also be labeled with arbitrary programming language code (which we abstract from in our UML RT analysis). Promela, on the other hand, is a concurrent programming language with concurrent processes, referred to as *proctypes*. It's control structure is much more flexible and versatile than that of UML RT, although state machines can easily be modeled in Promela. As opposed to UML RT, in Promela communication between proctypes can be via message passing or shared variables, and the communication topology can be dynamically changed.

However, once the static code analysis is completed and the message passing effect vectors have been determined, the boundedness analysis for the Promela case is identical to the analysis in the UML RT case. The focus of this paper is therefore on the specific problems that have to be addressed when analyzing Promela code in order to determine the message passing effect vectors. Issues that we will consider include

- the identification of message types, since in receive statements these can be referred to by variables whose values are not statically known;
- the passing of channel names as formal parameters during proctype instantiation;
- the replication of identical proctype instances;
- the assignment of channel variables;
- the use of channel array data structures where the arrays are indexed by variables not known statically;
- and the impact of unbounded proctype creation.

Paper Outline. For the sake of self-containedness of this paper we review the principle of our boundedness test in Section 2. In Section 3 we describe the solutions to the specific issues in the application of the analysis to Promela. Experimental results are discussed in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Boundedness Analysis

For the sake of self-containedness of this paper we now summarize the general principle of our boundedness analysis. A more detailed description can be found in [10].

First, we consider a sequence of conceptual abstractions for Promela models. In every step we obtain a coarser overapproximation of the previous model, for which the boundedness problem is easier to solve. All behavior of the original system is also possible in the overapproximations, i.e., they are monotonous w.r.t. simulation preorder. Furthermore, the abstractions preserve the (upper bounds on the) number of messages in every communication channel (buffer) of the Promela model. In practice, in the IBOC tools, all these abstractions are done in a single step.

Level 0: Promela code. We start with the original system model described in Promela, except that we a priori assume that buffers have arbitrary length. For this model (Promela with arbitrary length buffers) boundedness is, of course, undecidable, since the buffers could be used to simulate a Turing-machine tape.

Level 1: CFSMs. First, we abstract from the general program code in the model, i.e., variables, arithmetic, etc. We retain only the finite control structure of the program and the message passing behavior via unbounded buffers representing the communication channels. We obtain a system of communicating finite-state machines (CFSMs), sometimes also called FIFO-channel systems [1]. For the CFSM model boundedness is also undecidable [4].

Level 2: Parallel-Composition-VASS. In the next step we abstract from the order of the messages in the buffers and consider only the number of messages of any given type. For example, the buffer with contents `abbacb` would be represented by the integer vector $(2, 3, 1)$, representing 2 messages of type `a`, 3 messages of type `b` and 1 message of type `c`. Also we abstract from the ability to test explicitly whether a given buffer is empty. We so obtain a vector addition system with states (VASS) [3]. More exactly, we obtain a *parallel-composition-VASS*. This is a VASS whose finite-control is the parallel composition of several finite automata. Each part of this parallel composition corresponds to the finite control of some part of CFSM of level 1, and to the finite control of a process in the original Promela model. (Parallel-composition-VASS are as expressive, but more succinct than normal VASS.) The boundedness problem for parallel-composition-VASS is polynomially equivalent to the boundedness problem for Petri nets, which is $EXPSPACE$-complete [17].

Level 3: Parallel-Composition-VASS with Arbitrary Input. We now abstract from activation conditions of cycles in the control-graph of the VASS and assume instead that there are always enough messages, represented by tokens, present to start the cycle. As far as boundedness is concerned, we replace the problem 'Is the system bounded if starting at the given initial configuration?' by the problem 'Is the system bounded for any finite initial configuration?', also referred to as the *structural boundedness problem*. It has been shown in [10] that this structural boundedness problem for parallel-composition-VASS is co-$\mathcal{NP}$-complete, unlike for standard Petri nets where it is polynomial [12, 6].

Level 4: Independent Cycle System. Finally, we abstract from the fact that certain cycles in the control graph depend on each other. Instead we assume that all cycles are independent and any combination of them is executable infinitely often, provided that the combined effect of this combination on all places is non-negative. The *un*boundedness problem for this abstracted model then becomes the following question: Is there any linear combination (with non-negative integer coefficients) of the effects of simple cycles in the control graph, such that the combined effect is non-negative on all places and strictly positive on at least one place? Since we consider an overapproximation, the original Promela model is surely bounded if the answer to this question is 'no'. Since these effects of simple cycles can be represented by integer vectors, we get the following problem. Given a set of integer vectors, does there exist a linear combination (with non-negative integer coefficients) of them, such that the result is non-negative in every component and strictly positive in at least one. This problem can be solved in time polynomial in the number of vectors by using linear programming techniques.

However, the important aspect is that the time required is only polynomial in the number of simple cycles, unlike at level 3, where the problem is co-$\mathcal{NP}$-hard even for a linear number of simple cycles. This is very significant, since for instances derived from typical Promela models, the number of simple cycles is usually small. This is because the typical control-flow graphs of Promela code are (like in programming languages) sparse and often very local. (This is also the general reason why caching works.) Thus the number of different simple cycles derived from this code is typically polynomial rather than (in the worst case) exponential.

Overall Boundedness Test. From every simple cycle found in the control structure, a vector can be derived which describes its effect on the unbounded system part. Here, for the Promela model, the vector describes how many messages were altogether added to the buffer. For every buffer and every message type there is one component in each of the effect vectors. The component can be negative if in the cycle more messages of this type were removed from a buffer than added to it. The resulting semilinear system is unbounded if and only if there exists a linear combination with non-negative coefficients of the effect-vectors that is non-negative in every component and strictly positive in at least one component. Formally, this can be described as follows: Let $v_1, \ldots, v_n \in \mathbb{Z}^k$ be the effect-vectors of all simple cycles and let v^j be the j-th component of the vector v. The question then is

$$\exists x_1, \ldots, x_n \in \mathbb{N}_0.\ \sum_{i=1}^{n} x_i v_i \geq \mathbf{0} \ \wedge\ \exists j.\ \left(\sum_{i=1}^{n} x_i v_i\right)^j > 0.$$

This can easily be transformed into a system of linear inequations and solved by standard linear programming tools. If this condition is true then our overapproximation is unbounded, but not necessarily also the Promela model. The unboundedness could simply be due to the coarseness of the overapproximation. On the other hand, if the condition above is false, then our overapproximation is bounded, and thus our original Promela model is also bounded. Thus, this test yields an answer of the form "BOUNDED" in case no linear combination of the effect vectors satisfying the above constraint can be found, and "UNKNOWN" when such a linear combination exists.

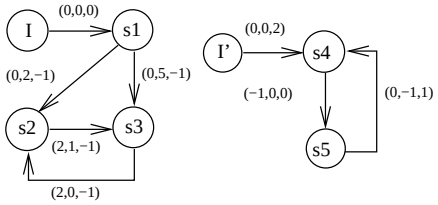

Fig. 1. Effect Graphs of a 2-Process Model

Example. Figure 1 describes effect graphs obtained from two communicating processes. The left process sends messages 'a' or 'b' to the right one, and the right process sends messages 'c' to the left one. The three components of the vector describe how many messages 'a', 'b' or 'c' are written (positive values) or read (negative values) in a step. For example, in the step from s_2 to s_3 two messages 'a' and one message 'b' are written and one message 'c' is read. From this graph we obtain the effect vectors $v_1 = (4, 1, -2)$ and $v_2 = (-1, -1, 1)$ for the simple cycles. To represent the > 0 condition in the linear inequation system we add a constraint $3x_1 - x_2 \geq 1$. The linear inequation solver returns infeasibility of this system of inequations, and we thus conclude a result of "BOUNDED".

Computing Bounds for Individual Buffers. A more refined problem is to compute upper bounds on the reachable lengths of individual buffers in the system. In particular, some buffers might be bounded even if the whole system is unbounded. Since normally not all buffers can reach maximal length simultaneously, the analysis is done individually for each buffer B. This can be done by solving a linear programming problem that maximizes a linear target function f_B (length of B) on an abstraction level 4 system description. The basic idea is the following. Let p be a path from the initial configuration to a configuration where B has maximal length. Then p can be decomposed into a cyclic part p_c and an acyclic part p_a. Since at abstraction level 4 we assume total independence of simple cycles, the effect of p_c can be described by a linear combination of the vectors describing the effects of simple cycles. It thus suffices to maximize f_B on this linear combination. Determining the maximal contribution of the acyclic part p_a is harder since one has to consider all possible combinations of acyclic paths in all parallel processes. (This is generally exponential in the number of parallel processes.) Therefore we only compute an upper bound on the effect of p_a as follows: Let n be the number of parallel processes and p_a^i the part of p_a in the i-th process. Let $E(x)$ be the effect vector of path x. Then $E(p_a) := \sum_{i=1}^{n} E(p_a^i)$. Now we compute vectors r_i which are upper bounds on $E(p_a^i)$, i.e., $\forall p_a^i.\ r_i \geq E(p_a^i)$. The r_i are computed (in polynomial time) by maximizing individually every component of the possible effect of paths p_a^i. (For example, if in process i there are two acyclic paths with effects $(3,1)$ and $(1,2)$ then $r_i = (3,2)$.) It follows that $E(p_a) = \sum_{i=1}^{n} E(p_a^i) \leq \sum_{i=1}^{n} r_i$ and thus $E(p) = E(p_c) + E(p_a) \leq E(p_c) + \sum_{i=1}^{n} r_i$. It only the remains to solve the linear optimization problem of f_B on $E(p)$ over $E(p_c)$ as explained above.

Example. Having established boundedness of the example of Figure 1, we now compute the estimated upper bound for each buffer. First we compute the effect vectors for all non-cyclic paths. They are listed in Table 1 where $init$ and $init'$ are the initial states of the state machines. Then we take the maxima of the individual components from those effect vectors and construct the overapproximated maximal effect vectors for process `Left` as $r_1 = (2,5,0)$ and for `Right` as $r_2 = (0,0,2)$. Thus the sum is $\sum_{i=1}^{n} r_i = (2,5,2)$. We obtain the following two optimization problems (1-4 and 5-8) for the two buffers left-to-right and right-to-left:

$$max : 2 - 2x_1 + x_2 \quad (1) \qquad max : 7 + 5x_1 - 2x_2 \quad (5)$$
$$2 + 4x_1 - x_2 \geq 0 \quad (2) \qquad 2 + 4x_1 - x_2 \geq 0 \quad (6)$$
$$5 + x_1 - x_2 \geq 0 \quad (3) \qquad 5 + x_1 - x_2 \geq 0 \quad (7)$$
$$2 - 2x_1 + x_2 \geq 0. \quad (4) \qquad 2 - 2x_1 + x_2 \geq 0. \quad (8)$$

Linear Programming returns a value of 6 for the objective function (1) and a value of 18 for the objective function (5). These values represent the estimated bounds for the communication buffers 1 and 2, respectively.

3 Promela-specific Issues

Given a Promela model the first step in the model analysis is to extract from it a system of CFSMs that consists of all state machines of all potentially executing processes.

The non-cyclic path	The effect vectors	The non-cyclic path	The effect vectors
$< init, s1 >$	(0,0,0)	$< init, s1, s2 >$	(0,2,-1)
$< init, s1, s2, s3 >$	(2,3,-2)	$< init, s1, s3 >$	(0,5,-1)
$< init, s1, s3, s2 >$	(2,5,-2)	$< init', s4 >$	(0,0,2)
		$< init', s4, s5 >$	(-1,0,2)

Table 1. The Effect Vectors for all Non-Cyclic Paths in Figure 1

SPIN can automatically generate a state machine representation for every proctype definition in the model [1]. The state machine of each actually instantiated process is then obtained from the state machine of its process type by replacing all formal arguments by the corresponding actual arguments. Figure 2 shows a simple Promela model and the corresponding system of CFSMs. There are two process instances at run time, one of process type P and the other of process type Q. Each transition in the state machine representation corresponds to a basic statement in the Promela code [2] of its process type. The source state of the transition denotes the entry point of the corresponding statement and the target state denotes the exit point of the statement. Every transition has a guard which is determined by the implicit executability condition of the corresponding statement. The executability of a statement describes under which condition the statement is executable. For instance, the transition from the state 4 to the state 3 is labeled by its corresponding statement $C?msg1$. The statement is executable if and only if there is a message $msg1$ available in the channel C.

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
active proctype P(){
        C!msg0;
        do
        :: C?msg1 -> C!msg0
        od}
active proctype Q(){
        C!msg1;
        do
        :: C?msg0 -> C!msg1
        od}
```

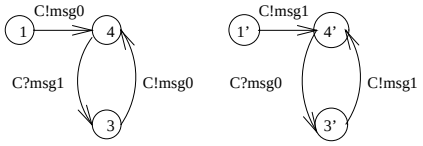

Fig. 2. A simple Promela model and the corresponding state machines.

We apply code abstraction to the resulting system of CFSMs. We replace all the statements in the state machines by their message passing effects. The resulting system is called the system of effect graphs. The remaining steps of the analysis, including cycle detection and translating the summary effects of all the cycles into a linear programming problem, are not different from the corresponding steps for UML RT models as described in [10]. In the remainder of this section we discuss several issues that have to be addressed during the Promela code abstraction in order to extract effect vectors

[1] This is accomplished by invoking the `spin -d` option.

[2] A basic statement is defined in the Promela language as an indivisible statement such as an assignment statement or a receive statement.

from the Promela code that ensure an overapproximation of the system. The resolution of these issues greatly influences the resulting systems of effect graphs, in particular how coarse the overapproximation is.

3.1 Identifying Message Types

Each component of an effect vector corresponds to a distinct message type. To construct effect vectors we must determine all distinct message types occurring in the model. Before we discuss this problem, we first review the syntactic structure of messages and the message passing semantics in Promela. The structures of the messages are defined in the declarations of the channels that store them. Consider the following channel declarations:

```
chan C1 = [2] of { mtype, int, bool }; chan C2 = [2] of { int, bool };
```

Messages can contain any finite number of fields. Each field can be of any well-formed type except arrays. In particular, the channel type $chan$ is allowed. There is a special type called $mtype$ that contains user-defined symbolic constants. In the Promela model in figure 2 $mtype$ is declared as containing two constants: $msg0$ and $msg1$. In any model, there can be at most one $mtype$ declaration. A $mtype$ field is not necessarily contained in every message. For instance, the messages in the channel $C2$ only contain an integer field and a boolean field.

Consider a send statement of the form $Ch!e_1, e_2, ..., e_n$ where each e_i ($1 \leq i \leq n$) can be an arbitrary Promela expression. The statement is executable if the channel Ch is not full. Nevertheless, under our a priori assumption of unboundedness of channels, the send statement is never blocked. The send statement sends the message $(val(e_1), val(e_2), ..., val(e_n))$ to the channel, where $val(e_i)$ is the run-time value of e_i.

For a receive statement $Ch?e_1, e_2, ..., e_n$[3], each e_i ($1 \leq i \leq n$) can be a variable, the evaluation $eval(v)$ of some variable v, or a constant. $eval(v)$ can be regarded as a constant. But its value is unknown statically. The statement is executable if all the constant expressions match the relative fields in the available message. Otherwise, it is blocked. If e_i is some variable v then, if the statement is executable, the corresponding field of the received message is stored into v.

The constants in a receive statement distinguish among messages containing different values of the relative fields. This allows users to associate with each message a proper piece of code for manipulation. We observe that these constants are often of type $mtype$. The variables in a receive statement can never block the execution. They are used to retrieve the relative fields from the received message. Consider the Promela model in Figure 3. The two receive statements in the model use constants of type $mtype$ to discriminate between the messages of type *exact* and *inexact*. The variable x stores the data upon receiving, no matter whether the received message type is *exact* or *inexact*, and no matter which integer is transmitted with the message.

[3] Promela knows additional forms of the send and receive primitives, denoted by *!!* and *??*, respectively. They are variants of the basic send and receive statement and only differ in the order in which they add and remove messages from the communication channels. Since our analysis abstracts from the order of messages in the buffer anyway, we do not distinguish these primitives here and map them to their respective base form.

```
mtype = {exact, inexact}
chan C = [2] of {mtype, int}
active proctype P(){
   int x;
   do
   :: C?exact, x -> keep(x)
   :: C?inexact, x -> dump(x)
   od
}
```

Fig. 3. Receive statements

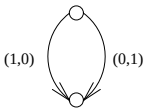

Fig. 4. Nondeterministic effect graph

```
mtype = {msg0, msg1};
chan C = [2] of {mtype};
chan D = [2] of {mtype};
proctype P(chan X; chan Y){
   do
   :: X?msg0; Y!msg0; Y!msg0
   od}
init{   run P(C, D);
        run P(D, D)}
```

Fig. 5. Channel parameters.

For the purpose of our analysis one can regard two messages which have the same structure but disagree on at least one field as being of different types. However, the number of the message types recognized in this way can be very large. It is also not necessary based on the following observation. In the previous example, the messages $(exact, 5)$ and $(inexact, 5)$ must be of different types because the model discriminates between them. On the contrary, the messages $(exact, 5)$ and $(exact, 7)$ need not to be of different types since the model treats them in exactly the same way. Only the first field of the messages is used to indicate the message types.

Based on the discussion so far, we propose the following solution to identify message types. For any channel Ch that stores messages with $mtype$ fields, we identify the types of the messages in Ch as pairs $(Ch, mconst)$ where $mconst$ is an $mtype$ constant. We include the channel name into message types because more than one channel can store messages with the same structure, and messages exchanged in different channels must be distinguished. For any channel Ch that stores messages without $mtype$ field, there is only one type, denoted as (Ch), for all the messages in Ch. In this way there are two message types identified for the model in Figure 3: $(C, exact)$ and $(C, inexact)$. This solution simply abstracts away all the non-$mtype$ fields of messages.

This abstraction approach becomes coarse if, in any model, some constants in some receive statements are of other types than $mtype$. For instance, if we have a receive statement $C?(exact, 5)$ in the model in Figure 3, the model discriminates between the messages containing the integer 5 and the messages containing other integers. Then the previously identified types are not sufficient to distinguish messages such as $(exact, 5)$ and $(exact, 7)$. Thus, for any channel *Ch* and receive statement $Ch?e_1, e_2, ..., e_i, ..., e_n$, where e_i is a constant c_i, those messages containing c_i in their i-th fields and those messages containing other constants in their i-th fields must be of different message types.

We propose a finer abstraction as follows. For any channel Ch that is declared as $chanCh = [k]$ of $\{type_1, type_2, ..., type_n\}$, the message types are those $(Ch, < d_1, d_2, ..., d_n >)$ such that

- D_i is the domain of the type $type_i$.
- $P(D_i)$ is a partition over D_i and $P(D_i) = \{\{const_1\}, \{const_2\}, ..., \{const_m\}, D_i - \bigcup_{j=1}^{j=m}\{const_j\}\}$ such that each constant $const_j \in D_i$ $(1 \leq i \leq m)$ is the i-th expression in some receive statement $Ch?e_1, e_2, ..., const_j, ..., e_n$.
- $d_i \in P(D_i)$.

For the example in Figure 3 augmented with the receive statement $C?exact, 5$, assuming I as the integer domain, we then have four message types as follows: $(C, < exact, \{5\} >)$, $(C, < exact, I - \{5\} >)$, $(C, < inexact, \{5\} >)$ and $(C, < inexact, I - \{5\} >)$.

3.2 Tracking *mtype* Variables

If an expression in a send or receive statement is a variable or the evaluation of a variable, its run-time value is not known statically. This affects our abstraction when the relative field of messages is used to identify the message types. For instance, assume that we have a send statement $C!x, y$ in the model in Figure 2 where x is a $mtype$ variable and y is an integer variable. We identify all the message types as $(C, exact)$ and $(C, inexact)$. In other words, the integer field of messages is abstracted away. Which type the sent message has depends only on the variable x. Since we can not generally determine the run-time values for x, we model the statement as sending nondeterministically any message whose $mtype$ field is either $exact$ or $inexact$. Therefore, in the resulting effect graph, there are two transitions. Both are leaving from the state corresponding to the entry point of the statement and lead to the state corresponding to the exit point, as shown in Figure 4. The left transition is labeled by the effect $(1, 0)$ denoting that a message of the type $(C, exact)$ is sent. The right transition is labeled by the effect $(0, 1)$, denoting that a message of the type $(C, inexact)$ is sent. Thus one obtains a nondeterministic choice between two transitions, as shown in Figure 4.

As mentioned before, there is at most one $mtype$ declaration in a model. All constants of type $mtype$ can be used by any send or receive operation on any channel. However, most channels usually use only a small portion of all $mtype$ constants. If we can determine for each channel the range of the $mtype$ constants it uses, we only need to consider those constants in the range for the nondeterministic modeling of message passing. So we could obtain a finer overapproximation. Our approach works by statically tracking possible values of $mtype$ variables. $mtype$ constants are numerical symbols. The Promela language allows for arithmetic operations on $mtype$ constants or variables. If $mtype = \{msg0, msg1\}$, $msg0$ and $msg1$ are internally represented by the compiler as integers 2 and 1, respectively. The statement $v = msg0 + msg1$ is syntactically valid and the $mtype$ variable v is assigned the value 3. Note that this value is outside the range of the integers for representing a $mtype$ constant in this example. However, Spin does not report such a range error. Arithmetic operations over the $mtype$ domain make it extremely hard to track $mtype$ variables. Due to these reasons, we exclude the usage of arithmetic operations over $mtype$ from our analysis. Hence there are only three ways to change the value of a $mtype$ variable: through an assignment, through a receive statement, or through argument passing.

We propose a solution to determine the ranges for the channels for the coarser approach of identifying message types. That means all non-$mtype$ fields of messages are abstracted away. The rules for updating the ranges for $mtype$ variables and channels are given as follows, where v_1 and v_2 are $mtype$ variables, $mconst$ is a $mtype$ constant, and Ch is a channel:

- Initially all the $mtype$ variables and channels have the empty set as the ranges for them.

- For the assignment $v_1 = mconst$, add $mconst$ to the range of v_1.
- For the assignment $v_1 = v_2$, add all the constants in the range of v_2 to the range of v_1.
- For the send statement $Ch!e_1, ..., v_1, ..., e_n$, add all the constants in the range of v_1 to the range of Ch.
- For the send statement $Ch!e_1, ..., mconst..., e_n$, add $mconst$ to the range of Ch.
- For the receive statement $Ch?e_1, ..., v_1, ..., e_n$, add all the constants in the range of Ch to the range of v_1.
- Assume a process type defined as $proctype\ P(...; mtype\ v_1; ...)$. For $run\ P(..., v_2, ...)$, add all the constants in the range of v_2 to the range of v_1.
- Assume a process type defined as $proctype\ P(...; mtype\ v_1, ...)$. For $run\ P(..., mconst, ...)$, add $mconst$ to the range of v_1.

After determining the range of a channel Ch, we may reduce the number of distinct message types. For instance, if the domain of the $mtype$ constants is D and the range of Ch is $D_{ch} \subset D$, then any message type $(Ch, mconst)$ can be discarded if $mconst \in D - D_{ch}$.

3.3 Channel Arguments

In Promela models, process types can be parameterized. For any process type that has formal arguments as channels, its instances with different instantiations of the channel arguments have different message passing behaviors. Consider the Promela model in Figure 5, where two running instances of process P are created. We refer to them as p_1 and p_2. The process p_1, instantiated as $P(C, D)$, accepts two different channels as the actual arguments. The process p_2 instantiated as $P(D, D)$ accepts the same channel D for both the formal arguments X and Y. We can easily observe that p_1 alone does not cause any unboundedness, while p_2 floods the channel D by messages $msg0$.

3.4 Replication of Proctypes

As shown above, different instances of some procedure with different channel arguments differ w.r.t. the boundedness of the channels. However, several parallel instances of some proctype with the same channel arguments do not contribute more to a potential unboundedness than just one, as far as our analysis is concerned. This is because in our abstraction level 4 (see Section 2) we assume all control-flow cycles to be independent. Thus two parallel copies of a procedure do not contribute more different control-flow cycles than just one.

3.5 Channel Assignments

The channel names specified in a Promela model are actually variables of the type $chan$. At run time, Spin maintains a set of actual channels called queues, and each channel variable keeps a pointer to a specific queue. The queue pointed to by a channel variable can be changed through channel assignments. Consider the Promela model in Figure 6. The channel C and D initially point to two separate queues. After the assignment

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
chan D = [2] of { mtype };
active proctype P(){
     C = D;
     do
     :: C?msg0; D!msg0; D!msg0
     od}
```

Fig. 6. A Promela model

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
chan D = [2] of { mtype };
active proctype P(){
     do
     :: C?msg0; D!msg0; D!msg0
     od;
     C = D;}
```

Fig. 7. A Promela model

$C = D$, C points to the same queue as pointed by D. It's easy to see that this queue is flooded by messages $msg0$.

A simple abstraction works as follows. Wherever we find a channel assignment $Ch_1 = Ch_2$ in the model, we merge the channels Ch_1 and Ch_2 into a single channel. That means one does not discriminate between the messages in Ch_1 and the messages in Ch_2. This is an overapproximation since we abstract from message orders. This solution is relatively coarse because a channel assignment does not necessarily affect all parts of the model. Consider the Promela model in Figure 7. Apparently the channel assignment $C = D$ does not affect the loop in the process type P where C and D still point to separate queues.

We propose a finer overapproximation based on the notion of strongly connected components (SCCs). A SCC in a directed graph is a subgraph in which any vertex is reachable from any other vertex. If we collapse all the vertices in the same SCC into a single vertex, we obtain a directed acyclic graph (DAG). In the DAG each vertex denotes a SCC in the original graph. Each transition from the state SCC_1 to the state SCC_2 corresponds to a transition in the original graph from one of the states in SCC_1 to one of the states in SCC_2. We derive the DAGs from the state machines of the running processes that contain channel assignments. It's obvious that a channel assignment in some SCC can only affect those SCCs reachable from it in the DAG of SCCs. For parallel processes, a channel assignment in one process can affect every part of every other process running in parallel. In this setting the effect vectors are constructed with separate components for different channels. However, at program locations where two channels are possibly identical, messages are nondeterministically sent to either channel. This encoding has the same effect as the unification of channels described above.

3.6 Channel Arrays

A set of channels can be declared as an array, e.g., *chan C[3] = [5] of* { *mtype*}. The channel array C consists of three channels indexed by integers between 0 and 2. For instance, the statement $C[1]!msg$ sends a message to the channel indexed at 1. When an index is a variable, its value is generally not known statically. For instance, the statement $C[i]!msg$ uses an integer variable i to index the array element. A simple solution is to model the statement as nondeterministically sending the message to any element of C, assuming that the run-time values of i always fall inside the range of the channel array indices. A finer approach is to statically track the index variable i to determine its range in a similar way as tracking $mtype$ variables. However we cannot exclude arithmetic

operations over integers. Whenever an arithmetic expression is met in an assignment, or in a receive statement, or in an argument passing, we have to set the range of the affected variable to the range of the channel array indices.

3.7 Unbounded Process Creations

The SPIN model checker limits the number of parallel processes to an implementation dependent constant which is in most installations 255. If one takes such a limit for granted then process creation alone could not lead to channel unboundedness. However, the Promela language could just as well be interpreted without this limitation. Here we show how unbounded parallel process creation could lead to channel-unboundedness (even in the absence of cycles in the control-flow graphs), and how our method could handle this problem.

Unbounded process creations can result in unbounded channels. There are two kinds of unbounded process creations. One kind is through local loops as demonstrated by the Promela model in Figure 8. The instance p of the process type P repeatedly creates instances of the process type Q. There is an execution where every new instance of Q immediately sends a message $msg0$ to the channel C after p creates it, and then stops there for p to create another new instance of Q. This floods C with messages $msg0$.

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
proctype Q(){
        C!msg0;
        C?msg1;}
active proctype P(){
        do
        :: run Q()
        od;}
```

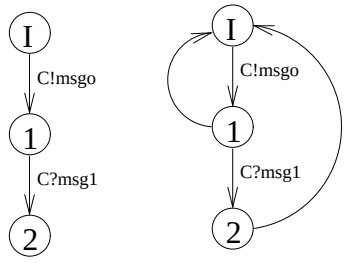

Fig. 8. A Promela model (left), the state machine of a running process of Q (middle), and the modified state machine with replication transitions (right).

The unboundedness of C can not be detected from the state machine of any instance of Q as shown in the middle of the figure. A straightforward solution is to add an extra backward transition from each state in the state machine to the initial state. These transitions are called replication transitions. The modified state machine is on the right in the figure. Now we can determine the loop $(I, 1, I)$ as the cause of the flooding of channel C.

The other kind of unbounded process creations is through self-creations or mutual creations. An example for self-creation is the Promela model in Figure 9. The channel C is unbounded because any new instance of P is self-created and every instance sends a message to the channel. The unboundedness is not detected from the state machine of any running process of P as shown in the middle of the figure. Similarly we use

replication transitions to detect self-creations. But we only need to add a replication transition for those states corresponding to the entry points of self-creations. So in the modified state machine on the right of the figure, there is no backward transition from state 2. The execution of any instance of P can never reach there before it creates a new instance.

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
active proctype P(){
        C!msg0;
        run P()}
```

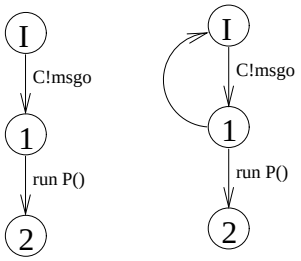

Fig. 9. A Promela model (left), the state machine of a running process of P (middle), and the modified state machine with replication transitions (right).

Now consider the situation that the unbounded process creations are through *mutual* invocation, as demonstrated by the Promela model in Figure 10. An instance of P creates an instance of Q that creates an instance of R. This instance of R creates in turn another new instance of P. The channel C is flooded with messages $msg0$ at run time.

One way of dealing with this is to use replication transitions again to detect the unboundedness caused by mutual creations. But for any process, the replication transition would not target its own initial state since it is not self-created. Instead, for instance, the replication transition from the state 1 of the state machine of a P instance transfers the control to the initial state of the state machine of a Q instance. In this way, several previously independent state machines are united to one much larger state machine. Thereby our boundedness analysis would no longer be local to individual processes, but had to consider the complete system, which would significantly increase its complexity. In fact, unlike before, the time required would then be exponential in the number of parallel components, even if each individual component contains only a polynomial number of simple cycles.

A second solution would be to add local replication transitions, just like in Figure 9, in every process which can, directly or indirectly, call itself. This is a safe, but coarser, overapproximation of the solution in Figure 10. The advantage is that one avoids interprocess transitions and thus keeps the boundedness analysis local and efficient.

4 Experimental Results

In this section we present experimental results using a prototype implementation of our analysis algorithms in the IBOC tool. As case studies we use the *2-Proctype* model that is given in Figure 11, a Promela model of the *Alternating Bit Protocol*, and a Promela

```
mtype = { msg0, msg1 };
chan C = [2] of { mtype };
proctype P(){
        C!msg0;
        run Q()}
proctype Q(){
        C?msg1;
        run R()}
proctype R(){
        C!msg1;
        run P()}
```

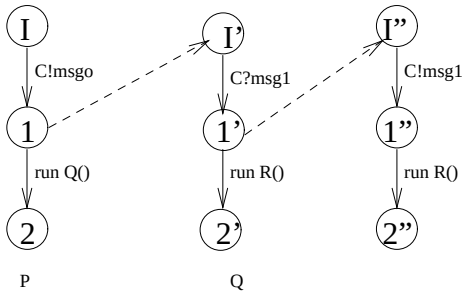

Fig. 10. A Promela model (left), the united state machines with replication transitions (right).

model of the CORBA *General Inter-ORB Protocol* (GIOP) [9][4]. IBOC uses to LP-SOLVE tool for the linear programming tasks. All experiments were performed on a two processor 1GHz Pentium III PC with 2 GB of memory. Table 2 gives some statistics regarding the complexity of these models as well as the computational effort for the analysis with IBOC.

```
mtype = { c, b, a };
chan AB = [25] of { mtype };
chan BA = [25] of { mtype };
active proctype A(){
  s1:   if
        :: BA?c -> AB!b; AB!b; goto s2
        :: BA?c -> AB!b; AB!b; AB!b; AB!b; AB!b; goto s3
        fi;
  s2:   if
        :: BA?c -> AB!a; AB!a; AB!b; goto s3
        fi;
  s3:   if
        :: BA?c -> AB!a; AB!a; goto s2
        fi}
```

```
active proctype B(){
        BA!c;
        BA!c;
        goto s4;
  s4:   if
        :: AB?a -> goto s5
        fi;
  s5:   if
        :: AB?b -> BA!c; goto s4
        fi;}
```

Fig. 11. The 2-Proctype Promela Model.

In IBOC, the full range of abstractions as discussed in Section 3 is not yet implemented. We use the finer abstraction proposed in Section 3.1 to identify message types. We are tracking *mtype* variables as discussed in Section 3.2. Since we can not exclude arithmetic operations over the integer domain, whenever we encounter an arithmetic expression in an assignment to a variable, we set the integer domain as the range of the variable. We collect all process creation statements to record channel argument passing for each running process. We adopt the coarser of the abstractions proposed in Section 3.5 to deal with channel assignments. We use some tracking of integer variables to narrow down channel array index values. We don't consider the unbounded process creation problem at all since SPIN allows no more than 255 concurrent proctype instances.

IBOC returned a result of "BOUNDED" for the 2-Proctypes model. The estimated bounds were 20 for the channel AB and 6 for the channel BA which allows us to reduce the channel size compared to the values in the original model. This entails a significant

[4] The Promela sources for the IBOC model that we use are freely available from URL http://tele.informatik.uni-freiburg.de/leue/sources/giop/giop-sttt.tar.

	2-Proctype	Alternating Bit	CORBA GIOP
Processes	2	3	5
States	20	12	135
Transitions	21	22	163
Message types	3	8	108
Channels	2	4	11
Reported cycles	3	13	2638
Generated vectors	2	13	752
Runtime for cycle detection [sec.]	0.062	0.218	7.187
Runtime for boundedness check [sec.]	0.000	0.031	0.098
Runtime for computing bounds [sec.]	0.032	-	1.265

Table 2. Model Complexity Statistics and Computational Effort for Analysis

state space reduction. Note that neither the boundedness result nor the estimated bounds could easily be derived from the model by manual inspection.

”UNKNOWN” was returned for the model of the Alternating Bit Protocol. When such a verdict is returned, IBOC indicates control flow cycles that possibly contribute to the unbounded growth of a channel. For this model IBOC identified the cycle in which the sender sends messages to the system as a potential source of unboundedness. This is quite plausible since in the sender can flood the Alternating Bit protocol in an unconstrained fashion. In the sequel we will refer to the potentially unbounded cycles that IBOC identifies as *counterexamples*.

The GIOP model is a real-life communication protocol with significant complexity. It amongst other features it supports server object migration between different Object Request Brokers (ORBs). IBOC returned an ”UNKNOWN” result for the GIOP model and provided two counterexamples within a very reasonable runtime. One counterexample is the cycle where a GIOP client ORB forwards a user request message to the GIOP agent ORB. The execution of this cycle can cause unboundedness if there are an unbounded number of user requests. The other counterexample is the cycle where server objects register their migration from one ORB to another. If we allow migrations to happen at any time, the system is flooded by an unbounded number of register messages. We eliminated these two sources of unboundedness from the GIOP system and applied IBOC again to the modified model. We obtained a result ”BOUNDED” which indicates that the two counterexamples were indeed the only sources of unboundedness in the system. While some of the buffer bound estimates were larger than the ones assumed in [9], there were also some channels with smaller estimates. For instance, the size of the channel $toServer$ in [9] is 3 while its estimate is 1.

5 Related Work

There is a long history of work on the handling of infinite communication buffers in automated system analysis. An overapproximation using the assumption that buffers may loose messages is proposed in [1]. Sufficient syntactic conditions for the *un*boundedness of communication channels in CFSM systems are proposed in [8]. There is a history of

checking properties of Petri-Nets using linear programming techniques (c.f. [11, 5]) but these approaches do not encompass boundedness tests. We are not aware of any work prior to ours that addresses buffer capacity estimation for CFSM-type models. Various attempts have been made to define formal operational semantics for Promela [13, 2, 16]. Note that our analysis largely relies on the recognition of statically analyzable features of Promela models, such as the control flow cycles, and many of the semantic subtleties of Promela were abstracted away. As a consequence our work does not depend on the availability of a completely specified and unanimously agreed semantics definition for Promela.

6 Conclusion

We presented an incomplete test for buffer overflows in Promela models as well as a conservative estimate for the maximal occupancy of Promela message channels. The experimental results we presented indicate that the analysis method scales well to problems of realistic size. We also illustrated that the analysis produces useful results, in particular the maximal occupancy estimates help in finding smaller models.

In comparison to the UML RT analysis presented in [10] the analysis of Promela code imposes more challenges on the employed code abstraction techniques. Due to its syntactically more constrained nature, the determination of message types and the identification of communication channels is not an issue.

Current research focuses on the improvement of counterexample handling and the identification of sources of unboundedness. As we discussed above, IBOC attempts to point the user to potential sources of unboundedness. Determining the non-spuriousness of a counterexample and the ensuing abstraction refinement are currently entirely handcraft, and we are currently working towards more automated support at this end.

Acknowledgements. The third author was supported through the DFG funded project IMCOS (grant number LE 1342/1). We thank all involved students, in particular Quang Minh Bui, for their effort in developing IBOC.

References

1. P. Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. In *LICS'93*. IEEE, 1993.
2. William R. Bevier. Towards an operational semantics of promela in acl2. In *Proceedings of the Third SPIN Workshop, SPIN97*, 1997.
3. A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In *Proc. of STACS'99*, volume 1563 of *LNCS*. Springer Verlag, 1999.
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.
5. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16:159–189, 2000.
6. J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Informatics, Processing and Cybernetics*, 30(3):143–160, 1994.

7. Gerard J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.
8. T. Jeron and C. Jard. Testing for unboundedness of fifo channels. *Theoretical Computer Science*, (113):93–117, 1993.
9. M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Software Tools for Technology Transfer (STTT)*, volume 2, pages 394–409, 2000.
10. S. Leue, R. Mayr, and W. Wei. A scalable incomplete test for the boundedness of UML RT models. In *Proceedings of TACAS 2004*, LNCS. Springer Verlag, 2004.
11. S. Melzer and J. Esparza. Checking system properties via integer programming. In H.R. Nielson, editor, *Proc. of ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 250–264. Springer Verlag, 1996.
12. G. Memmi and G. Roucairol. Linear algebra in net theory. In *Net Theory and Applications*, volume 84 of *LNCS*, pages 213–223, 1980.
13. V. Natarajan and G.J. Holzmann. Outline for an operational semantics of promela. In Jean-Charles Grgoire, Gerard J. Holzmann, and Doron A. Peled, editors, *The SPIN Verification System. Proceedings of the Second SPIN Workshop 1996.*, volume 32 of *DIMACS*. AMS, 1997.
14. B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.
15. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. http://www.rational.com/media/whitepapers/umlrt.pdf, March 1998.
16. Carsten Weise. An incremental formal semantics for promela. In *Proceedings of the Third SPIN Workshop, SPIN97*, 1997.
17. H. Yen. A unified approach for deciding the existence of certain Petri net paths. *Information and Computation*, 96(1):119–137, 1992.