Heuristic Search for Unbounded Executions

Matthias Kuntz, Stefan Leue, Christoph Scheben, Wei Wei, and Sen Yang

Department of Computer and Information Science, University of Konstanz
D-78457 Konstanz, Germany

Abstract. We present a heuristic search based approach to finding unbounded executions in software models that can be described using Communicating Finite State Machines (CFSMs). This improves the unboundedness test devised by Jeron and Jard in case certain knowledge about potential sources of unboundedness is available. Such knowledge can be obtained from a boundedness analysis that we designed in precursory work. We evaluate the effectiveness of several different heuristics and search strategies. To show the feasibility of our approach, we compare the performance of the heuristic search algorithms with that of uninformed search algorithms in detecting unbounded executions for a number of case studies. We discuss the applicability of our approach to high level modeling languages for concurrent systems such as Promela.

1 Introduction

Unboundedness of communication buffers has several negative impacts on software designs. First, buffer unboundedness may cause buffer overflow, loss of messages, and other undesired behavior. Second, unbounded buffers render the state space of a software system infinite, which impedes the applicability of software analysis and verification techniques based on exhaustive state space exploration, such as model checking. Jeron and Jard [8] proposed an incomplete unboundedness test based on depth-first search (DFS) on reachability trees of Communicating Finite State Machines (CFSMs). One disadvantage of their test is that the search is uninformed, so it may become inefficient to discover unbounded executions. In precursory work, we devised a boundedness test for CFSMs that returns counterexamples in case boundedness cannot be proved [11]. A counterexample in our setting is a set of simple control flow cycles together with their respective iteration counts. A counterexample indicates which combination of cycles causes unboundedness. In this paper we suggest several informed algorithms to search for unbounded executions, that utilize heuristics derived from counterexamples. These suggested algorithms can also complement our boundedness test to show that a model is truly unbounded.

Related work. (Un-)boundedness is known to be undecidable for CFSMs [1]. We are not aware of any work on testing unboundedness except for [8], on which our approach is based. On the other side, many approaches exist for testing boundedness, such as [1, 14, 9] that check only subclasses of CFSMs, and incomplete

decision procedures [10, 15] that treat the whole class. Scheduling techniques have been applied to assure that only bounded executions can occur during run-time, even if the system is unbounded [17, 12, 2]. Finally, the idea of using heuristics to search for short paths has been extensively used in many areas such as robotic planning [18], model checking [4], among others.

Outline. This paper is further organised as follows. In Sec. 2 we formally define CFSMs and (un-)boundedness. We also explain Jeron and Jard's unboundedness test and our boundedness analysis. In Sec. 3 we briefly introduce the heuristic search alorithms A^* and Best-First-Search. In Sec. 4 we present our heuristic search approach for unbounded executions of CFSMs. We discuss the issues of testing unboundedness for PROMELA models in Sec. 5. In Sec. 6 we report on experimental results, and conclude the paper in Sec. 7.

2 Unboundedness

2.1 CFSMs

The formalism of CFSMs is a common model for asynchronous concurrent systems. Informally, a system of CFSMs is a set of concurrent processes that communicate via the exchange of messages. Processes are autonomous, so they can be executed in any possible interleaving order. The behaviour of a process is captured by a labeled transition system. Messages are stored in First-In-First-Out (FIFO) buffers. After sending a message, the sender process can continue its local execution. A receiver process blocks only if the expected message is not available. Fig. 1 shows an example of CFSMs.

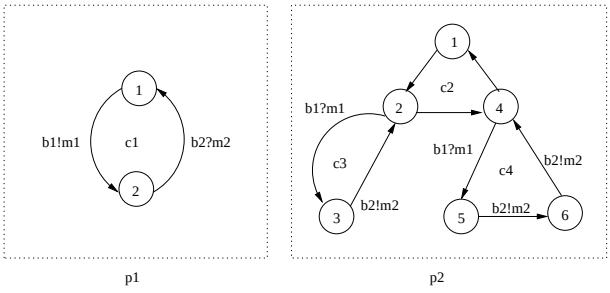

Fig. 1. A CFSM system.

Definition 1 (Communicating Finite State Machines). *A system of* Communicating Finite State Machines *is a triple* $(P, B, succ)$*, where*

- P*: a finite set of processes. Each process* $p_i \in P$ *is a pair* (S^i, s_0^i)*, where* S^i *is a finite set of local states of* p_i*, and* $s_0^i \in S^i$ *is the unique initial state.*

- *B: a finite set of message buffers. Each buffer $b \in B$ is associated with a finite alphabet $\Sigma(b)$ that represents the set of exchangeable messages in b.*
- *succ: a finite set of local transitions (s_1^i, e, s_2^i), where $s_1^i, s_2^i \in S^i$ for some process p_i, and e is either empty, or a message sending event $b!m$, or a message receiving event $b?m$.*

Definition 2 (Global State). *In the above defined system, a* global state *s is a tuple $(s^1, \ldots, s^{|P|}, q^1, \ldots, q^{|B|})$, where each s^i is a local state of process p_i and q^i is a queue of messages in buffer b_i. We define the* initial global state *of the system, denoted by s_0, to be $(s_0^1, \ldots, s_0^{|P|}, \varepsilon, \ldots, \varepsilon)$ where all processes are in their local initial states and all buffers are empty.*

In the rest of the paper, we will use $b(s)$ to denote the content of a buffer b in a global state s, and use $p_i(s)$ to denote the local state of p_i in s. Let q and q' be two sequences, then $q.q'$ denotes the concatenation of the two sequences.

Definition 3 (Successor). *In the above definition, a global state s_2 is a* successor *of another state s_1 if and only if there exists a process $p_i \in P$ such that the following conditions hold:*

- *There exists a transition $(p_i(s_1), e, p_i(s_2)) \in succ$ such that*
  - *If e is empty, then for every buffer $b \in B$ it holds that $b(s_1) = b(s_2)$.*
  - *If $e = b!m$, then $b(s_2) = b(s_1).m$ and, for any other buffer b', $b'(s_1) = b'(s_2)$.*
  - *If $e = b?m$, then $m.b(s_2) = b(s_1)$ and, for any other buffer b', $b'(s_1) = b'(s_2)$.*
- *For any other process p_j, we have $p_j(s_1) = p_j(s_2)$.*

The semantics of CFSMs can be given as reachability trees.

Definition 4 (Reachability Tree). *Given a system of CFSMs $(P, B, succ)$, its* reachability *tree RT is the smallest tree satisfying the following conditions:*

- *The root of RT is labeled with the initial global state s_0 of the system.*
- *Let n be a node in RT labeled with a global state s. For every successor s' of s, there is a child n' of n labeled with s'.*

Every finite or infinite path in RT is an execution of the system. Note that RT may have multiple nodes, even potentially infinitely many, labeled with the same global state s. For each occurrence of s in RT, it is reached from s_0 on a distinct path. In particular, RT can be infinite even when there are only finitely many reachable global states.

2.2 (Un)boundedness

Definition 5 (Bounded and Unbounded Executions). *Given a CFSM system, an execution r is* bounded *if there exists a natural number k such that, for any buffer b, any global state reached along r contains no more than k messages in b. If no such k exists, then r is* unbounded.

The existence of unbounded executions is undecidable for CFSMs [5].

Example 1. Consider the example CFSM system in Fig. 1. An example of a bounded execution is one in which only the cycles c_1 and c_3 are executed forever. An unbounded execution can be obtained by repeating c_1 and c_4 exclusively.

2.3 The Jeron-Jard-Test

The Jeron-Jard-test [8] finds unbounded executions by performing a depth-first search on the reachability tree of a CFSM system. Every time when a new global state s is encountered during the search, the test checks the satisfaction of a sufficient condition for unboundedness by s and all its ancestors in the reachability tree. If there is an ancestor a such that s and a satisfy the condition, then an unbounded execution $r = r_1.(r_2)^\omega$ is discovered where r_1 is the path from the initial state s_0 to a and r_2 is the path from a to s. We call $r_1.r_2$ the *generating path* of r. In the following we formally define the sufficient unboundedness condition.

Let r be an arbitrary finite path in a reachability tree. We define $out_b(r)$ for a buffer b inductively as follows: (1) In case the length $len(r) = 0$, then $out_b(r) = \varepsilon$ is empty; (2) Suppose $len(r) = k$ where $k > 0$. Let $r = r'.e$ where the edge e connects two global states s_1 and s_2. If s_2 is reached from s_1 by sending a message m to b, then $out_b(r) = out_b(r').m$. Otherwise, $out_b(r) = out_b(r')$. Intuitively, $out_b(r)$ is the sequence of messages ever generated along r.

Given two global states s_1 and s_2 such that s_2 is reached from s_1 by a finite path r, s_1 and s_2 satisfies the unboundedness condition if and only if

1. for every process p, $p(s_1) = p(s_2)$, and
2. for each buffer b, $b(s_1).out_b(r)$ is a prefix of $b(s_2).out_b(r)$, and
3. there exists a buffer b' such that $b'(s_1).out_{b'}(r)$ is a proper prefix of $b'(s_2).out_{b'}(r)$.

As the above condition is only sufficient but not necessary, not all sources of unboundedness can be discovered by the Jeron-Jard-test.

Example 2. Consider the CFSM system in Fig. 1. The reachability tree of the system contains the following path r from the initial state: $s_0 = (1, 1, \varepsilon, \varepsilon)$, $s_1 = (2, 1, m_1, \varepsilon)$, $s_2 = (2, 2, m_1, \varepsilon)$, $s_3 = (2, 3, \varepsilon, \varepsilon)$, $s_4 = (2, 2, \varepsilon, m_2)$, $s_5 = (1, 2, \varepsilon, \varepsilon)$, $s_6 = (1, 4, \varepsilon, \varepsilon)$, $s_7 = (2, 4, m_1, \varepsilon)$, $s_8 = (2, 5, \varepsilon, \varepsilon)$, $s_9 = (2, 6, \varepsilon, m_2)$, $s_{10} = (1, 6, \varepsilon, \varepsilon)$, $s_{11} = (2, 6, m_1, \varepsilon)$, $s_{12} = (2, 1, m_1, m_2)$. The states s_1, s_{12} and the path r' in-between satisfy the unboundedness condition: (1) s_1 and s_{12} agree on local states; (2) $b_1(s_1) = m_1$, $b_1(s_{12}) = m_1$, $out_{b_1}(r') = m_1.m_1$, so $b_1(s_1).out_{b_1}(r')$ is a prefix of $b_1(s_{12}).out_{b_1}(r')$; (3) $b_2(s_1) = \varepsilon$, $b_2(s_{12}) = m_2$, $out_{b_2}(r') = m_2.m_2$, so $b_2(s_1).out_{b_2}(r')$ is a proper prefix of $b_2(s_{12}).out_{b_2}(r')$. Then, we can conclude that $r.(r')^\omega$ is an unbounded execution while r is the generating path.

The disadvantage of the Jeron-Jard-test is that it performs an uninformed search on the reachability tree of a model. Since it is not aware of the sources

of unboundedness, the test becomes inefficient when an unbounded execution is discovered only after a large number of bounded executions are explored first. For instance, in the above example, the path r executes the cycle c_3 in the process p_2 before executing c_4. If c_4 is chosen to be executed first, then an unbounded execution with a shorter generating path can be found earlier. A shorter generating path is always preferable because it is more comprehensible for diagnosis. We next present a boundedness analysis and then discuss how it may help to guide the search for unbounded executions.

2.4 Buffer Boundedness Check

The boundedness check can be applied to Promela [7] and UML-RT models [10, 11], both of which are first abstracted into CFSMs during the check. In this paper, we present the test on the level of CFSMs.

The intuitive idea of this method is to check whether there is any combination of local cycles in a model that when executed has the potential to blow up some message buffer. The boundedness check applies a series of abstractions to abstract from message orders, from activation conditions of cycles, and from cycle dependencies. The resulting abstract system is a set of simple cycles with their message passing behavior over-approximated as integer vectors. Consider the example in Fig. 1. The abstract cycle system of this model consists of the cycles $c_1, \ldots, c_4$ while their respective message passing effect vectors are $e_1 = (1, -1)$, $e_2 = (0, 0)$, $e_3 = (-1, 1)$, and $e_4 = (-1, 2)$. In these vectors, the first components correspond to messages m_1 and the second components to messages m_2.

Next, a necessary condition for unboundedness is encoded into an integer linear programming (ILP) problem based on the set of cycles and their effects, whose infeasibility implies boundedness. If the resulting ILP problem has solutions, then no verdict about the boundedness can be delivered. In our example, the ILP problem is $x_1 \cdot e_1 + x_2 \cdot e_2 + x_3 \cdot e_3 + x_4 \cdot e_4 > 0$. Every x_i denotes the number of times that the cycle c_i is repeated in a particular combination. Intuitively, any solution to this ILP problem gives a combination of cycle executions that consumes no messages and sends at least one message overall. It is easy to find a solution in which $x_1 = x_4 = 1$ and $x_2 = x_3 = 0$.

The boundedness check is inherently incomplete [11, 1]. But in case boundedness cannot be proved, the check constructs a counterexample from any solution to the ILP problem. Intuitively, a counterexample indicates which cycles should be repeated infinitely often and how their executions should be combined in order to make some message buffers unbounded. We formally define counterexamples as follows.

Definition 6 (Counterexample). *A* counterexample *E is a function from the set of simple cycles in a model to natural numbers. In particular, for some cycle c with $E(c) > 0$, we say that c is in E and $E(c)$ is the* repetition counter *for c.*

A counterexample E can be simply constructed from an ILP solution by assigning the value of x_i in the solution to $E(c_i)$, assuming that x_i corresponds

to the cycle c_i. In our example, a counterexample E is $E(c_1) = E(c_4) = 1$ and $E(c_2) = E(c_3) = 0$.

Definition 7 (Counterexample Conforming Execution). *An execution r* conforms *to a counterexample E if and only if $r = r_1.(r_2)^\omega$ and when the following conditions hold:*

- *r_1 is a finite path.*
- *r_2 is a finite path in which only the cycles in E are executed. In particular, no transitions outside the cycles in E are taken along r_2.*
- *For each cycle c in E, if c is executed in r_2, then all the executions of c are complete, and the number of times that c is executed is no larger than $E(c)$.*

A counterexample is real if there is an unbounded execution that conforms to the counterexample. Otherwise, due to the over-approximating nature of the abstractions that the boundedness check uses the counterexample may also be spurious. In [19] we extended the boundedness check to determine spuriousness of counterexamples automatically based on the discovery of static cycle dependencies. However, the boundedness check is unable to determine whether a counterexample is real.

2.5 Using Counterexamples as Heuristics

Since a counterexample constructed by the boundedness analysis indicates a potential source of unboundedness, we can use this information to guide the search in the reachability tree in order to find unbounded executions more efficiently. The basic idea is to find an unbounded execution that conforms to the given counterexample. In case such a conforming execution is found, we can conclude that the counterexample is real, which complements the incompleteness of the boundedness check.

Consider the model in Fig. 1. Since we obtain a counterexample E where $E(c_1) = E(c_4) = 1$ and $E(c_2) = E(c_3) = 0$, we look for a conforming execution $r = r_1.(r_2)^\omega$ in which r_2 executes only the cycles c_1 and c_4 and nothing else. Therefore, we should move the execution of the two processes p_1 and p_2 as quickly as possible to the cycles c_1 and c_4. Once the executions of these cycles start, we should stick with them until their executions are completed. This is the intuitive idea behind the design of the heuristic functions with respect to counterexample, which we will discuss in greater detail in Sec. 4.

3 Heuristic Search Algorithms

We consider to use A^* [6] because it may give (sub-)optimal solutions, i.e., shorter unbounded executions in our case.

The A^* algorithm. As an *informed* search algorithm, A^* attempts to find the path to a target node (or a goal state) that induces the minimal cost, e.g., the shortest path. This algorithm computes heuristic values for all reached nodes. A heuristic value $f(s) = g(s) + h(s)$ of a node includes (1) the accumulated cost $g(s)$ to reach the node so far and (2) an estimate of the remaining cost $h(s)$ to reach the target node. A^* maintains an open list of nodes from which it always chooses a node with the smallest heuristic value to expand. Initially, the initial node is inserted to the open list. Each time when a node s is expanded, s is removed from the open list, and every successor s' is treated as follows: (1) If s' has not yet been visited, then it is added to the open list; (2) If s' has been visited and the new heuristic value of s' is smaller than its old heuristic value, then s' appears in the open list with the new heuristic value. This is the concept of "reopening"; (3) Otherwise, s' is simply discarded. A^* checks if a target node is reached each time before a node is expanded. This is necessary to ensure optimality. The search terminates either if the target node has been found, or if the open list becomes empty. In the latter case no solution exists.

A^* terminates and returns an optimal path if the heuristic function f is monotone: First, f never overestimates the actual costs to reach the goal state; Second, for each node s and each successor s' of s it holds that $h(s) \leq c(s, s') + h(s')$ where $c(s, s')$ denotes the actual cost to reach s' from s.

Normally, A^* operates on a graph and searches for a target node in the graph. However, in our context we look for paths representing potentially unbounded executions. Whether a path is a generating path for an unbounded execution depends entirely on the path itself rather than on some predefined goal states. Therefore, we need to apply A^* on reachability trees. This allows us to cover all possible acyclic paths leading to a state. Note that the Jeron-Jard-test indeed works on reachability trees. While searching using our modified version of A^*, a state s is always inserted to the open list with its current heuristic value whenever it is encountered, unless s is reached by a path p from s_0 such that s occurs in p, in which case a global loop has been encountered. As a consequence a state s may have multiple occurrences in the open list. We disregard global loops because (1) a loop does not contribute to unboundedness, and (2) we avoid unbounded unfolding of a loop. The modified algorithm may not terminate on infinite reachability trees since there is no predefined goal state. All other properties of the standard A^* are preserved by our modification because the open list used in the modified version always contains more information, and no less, than the open list in the standard version.

Best-First Search. By removing the g part of the heuristic function, A^* becomes Best-First Search (`BEST`), which disregards the cost accumulated so far to reach a state.

4 Heuristics-Based Unboundedness Test

Let M be a system of CFSMs, P be the set of all processes in M, and E be a counterexample throughout this section. We first assume that E is real, and consider

the following problem under this assumption: "Find an unbounded execution r that conforms to E." This problem is unsolvable. The proof of unsolvability can be easily obtained by a reduction from the well-known undecidable problem of the executability of a message reception in CFSMs [1]. We omit the proof due to limited space.

In Sec. 4.1, we present an incomplete solution to the above problem. In Sec. 4.2 we discuss some alternative heuristic functions and strategies in search for unbounded executions when we drop the assumption that E is real.

4.1 Finding Conforming Unbounded Executions

We derive heuristics from E to guide the search in the reachability tree of M for unbounded executions that conform to E. Intuitively, the derived heuristics should work as follows.

Remember that an execution r conforming to E can be decomposed into two parts $r = r_1.(r_2)^\omega$ where the exclusive repetition of the cycles in E starts at the end of r_1. As mentioned previously, we are interested in such a prefix r_1 with the shortest length. Furthermore, the exclusive repetition of r_2 can possibly begin only if, for each process p in M that has cycles in E, the local state of p is already in one of the cycles in M. So, before r_2 is reached, the heuristics should guide the search in such a way to move the local state of each process as close as possible to those cycles in M. After the execution of r_2 starts, the heuristics should guide the search to stick to r_2 until a conforming execution is found.

The algorithm (see Appendix A) works as follows: Let s be one occurrence of an arbitrary global state of M in the reachability tree. The heuristic function $f(s) = g(s) + h_1(s) + h_2(s)$ consists of three parts. The value of $g(s)$ is the distance from the initial state s_0 to s in the reachability tree.

The function $h_1(s) = \sum_{p \in P} dist_p(s)$ denotes the sum of the shortest local distance of each process to the cycles in E, which can be easily computed using Dijkstra's algorithm [3]. In particular, for any $p \in P$ that has no cycles in E, we define that $dist_p(s) = 0$. A state s may be the starting state of r_2 only if $h_1(s) = 0$.

The value of $h_2(s)$ estimates the number of remaining steps to finish r_2, i.e., the execution of E. Its computation rests on a set of counters $\{ct_i(s)\}$, each corresponding to a transition t_i contained in some cycles in E. Each counter $ct_i(s)$ remembers how many times the transition t_i has yet to be taken from s until the execution of r_2 is finished. The value of $h_2(s)$ is simply the sum of all counters at s. We use ct_i^0 to denote the initial value of $ct_i(s)$. Let t_i be contained in the cycles $\{c_1, ..., c_j\} \subseteq E$. Then, $ct_i^0 = \sum_{k=1}^{j} E(c_k)$. In the following, we explain how the value of $ct_i(s)$ is computed from the counter values in the predecessor v of s. Suppose that s is reached from v by taking the transition t_i in the reachability tree. Moreover, let T denote the set of transitions contained in the cycles in E. When s is reached during the search, we calculate counter values depending on the following cases:

1. The condition $h_1(v) > 0$ implies that the execution of E cannot have been started. In this case all counters receive their respective initial values. Particularly, if $h_1(s) = 0$, then s may be the starting state of E's execution.
2. The condition $h_1(v) = 0$ indicates that E's execution may have begun. In this case we need to distinguish the following cases in regard to t_i:
   (a) If $t \in T$ and $ct_i(v) > 0$, then it may still be on the way to finish the execution of E. In this case $ct_i(s) = ct_i(v) - 1$, and $ct_j(s) = ct_j(v)$ for any other transition t_j.
   (b) If $t \in T$ but $ct_i(v) = 0$, then we have run out of the number of times to execute t_i. In this case we need to adjust counter values to change the potential starting state of E's execution. This is easily achieved by a backward traversal of the reachability tree from s as implemented in the function `adjustCounters` (see Alg. 2 in App. A).
   (c) If $t \notin T$, then the execution of E should not have been started. We set all counters to their respective initial values.

Alg. 1 is a modified version of A^* to search on a reachability tree using the above described heuristic function. Unlike Jeron and Jard's test based on DFS, our algorithm does not need to check the unboundedness condition at every state with each of its ancestors. Since we are only looking for an unbounded conforming execution, it suffices to check the unboundedness condition at those states where the h_1 value is 0. The ancestor check in Jeron and Jard is also dispensable. Suppose that $h_1(s) = 0$. Let $d = \sum_{t_i \in T} ct_i^0 - h_2(s)$. The value d denotes the number of steps that may have been taken in executing E. Our algorithm only needs to check the unboundedness condition at s with all its ancestors within the distance d. If there is an ancestor a such that a and s satisfy the condition, then we have found an unbounded conforming execution $r = r_1.(r_2)^\omega$ where r_1 is the path from s_0 to a in the reachability tree, and r_2 is the path from a to s.

Example 3. In Sec. 2.4 the boundedness analysis returns a counterexample E for the model in Fig. 1, where $E(c_1) = E(c_4) = 1$ and $E(c_2) = E(c_3) = 0$. Figure 2 shows the portion of the reachability tree constructed for the model. Every global state is represented as a rounded rectangle. States in solid lines are expanded, and those in dotted lines are only visited but not expanded. Our algorithm starts with the initial state s_0. There are 5 transitions in the cycles in E, and each transition needs to be executed only once. So, $h_2(s_0) = 5$. First, s_0 is expanded, and we obtain two successors s_1 and s_2. To reach s_2, the process p_2 is executed to move the local state one step closer to the cycle c_4. Therefore, s_2 has a smaller heuristic value than s_1 and is chosen to be expanded. A similar argument holds for the fact that s_4 is expanded instead of s_3. The execution of E may start at s_4 because $h_1(s_4) = 0$. The two successors of s_7 have equal heuristic values, so any of them can be chosen for expansion. We randomly choose s_8. The state s_8 has two successors. State s_{10} is reached by taking the transition in c_1 to send a message m_1. However, the cycle c_1 had been completely executed once before reaching s_8. Therefore, the function `adjustCounters` is called at s_{10}, resulting in $h_2(s_{10}) = 1$. This leads to a greater heuristic value than that of s_{11}, which

is taken out of the `open` list first. At this point, the unboundedness condition is checked for s_{11} with its ancestors within 5 steps. It is easy to see that s_{11} and s_4 satisfy the condition. We find an unbounded execution whose generating path is only 7 steps long. Note that before s_{11}, the algorithm also checks the unboundedness condition at the states $s_4, \ldots, s_8$ because their h_1 values are zero, but none of these states satisfies the condition with their ancestors.

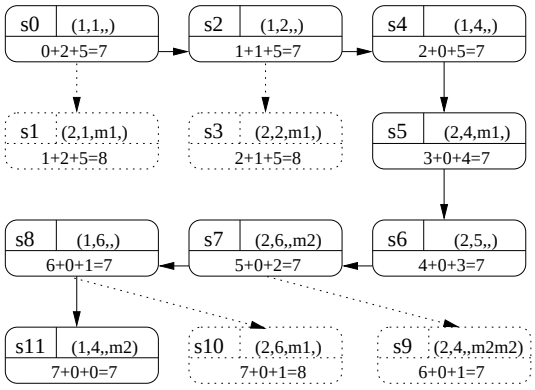

Fig. 2. The portion of the reachability tree constructed by Alg. 1 for Ex. 1. Each state is represented as a rectangle consisting of three regions: (1) the upper-left region is the name of the state; (2) the vector (s^1, s^2, q_1, q_2) in the upper-right region denotes the local states s^1 and s^2 of p_1 and p_2, and the contents of the buffers b_1 and b_2; (3) the lower region is the heuristic value in form $g + h_1 + h_2$.

Soundness, Completeness, and Optimality. The soundness of the Alg. 1 can be easily seen from the definition of counterexample conformance and the soundness of the unboundedness condition. The algorithm is incomplete because unboundedness is undecidable. However, it is easy to see that the incompleteness solely comes from the sufficient unboundedness condition. The proof is sketched as follows. Suppose the execution $r = r_1.(r_2)^\omega$ is the only unbounded conforming execution where the execution of E starts at the end of r_1. We can safely claim that r is free of global loops. Then, at some point during the search, the path $r_1.r_2$ will be explored. At the end of r_2, we will check the unboundedness condition for the starting state and the last state of r_2. We will miss this unbounded execution only if the checking of the condition fails. For two reasons our algorithm is not optimal. First, this is due to the unboundedness condition. Second, the heuristic function is defined to estimate the cost to completely execute E, which means that every cycle c in E is executed exactly $E(c)$ times. But an unbounded conforming execution may be found before E is completely executed. Consequently, a heuristic value may be over-estimating. We can, however, prove the following:

Theorem 1. *Let $r = r_1.(r_2)^\omega$ be the unbounded conforming execution found by Alg. 1. Suppose that there exists another unbounded conforming execution $r' = r'_1.(r'_2)^\omega$ such that (1) r' can be determined by the unboundedness condition; and (2) the length l' of $r'_1.r'_2$ is smaller than the length l of $r_1.r_2$. Let $|E|$ denote the number of steps to complete the execution of E. Then, we have that $l < l' + |E|$.*

Proof. The proof makes use of the fact that heuristic values never decrease along a path. We leave out the proof of this fact for limited space. Let s be the last state of r_2. Then, we have $g(s) = l$ and $h_1(s) = 0$. Let s' be the last state to be expanded along $r'_1.r'_2$. Based on the above fact, $f(s') \leq l' + k$ for some natural number $k < |E|$, where $l' + k$ would be the heuristic value of the last state of r'_2 if the algorithm reached the end of r'_2. Because s is taken out of the `open` list prior to s', we have (a) $f(s) = l + h_2(s) \leq f(s') \leq l' + k$. Assume by contradiction that (b) $l \geq l' + |E|$. Combining (a) and (b), we have $h_2(s) \leq k - |E| < 0$. This is a contradiction, which implies $l < l' + |E|$.

4.2 Alternative Heuristics and Search Algorithms

Until now we have made the assumption that any input counterexample to the Alg. 1 is real. However, it is not always easy to obtain real counterexamples from a model. In particular, our boundedness analysis may return a spurious counterexample. In such case there cannot be any unbounded execution conforming to this spurious counterexample.

Suppose that the spuriousness of the counterexample E is unknown, if we also want to find some unbounded execution in case E is actually spurious, then we need to modify the Alg. 1 to allow the unboundedness condition to be checked for every state, before it is expanded, with every ancestor of the state. The heuristic function still makes it preferable to find an unbounded execution conforming to E.

In the Alg. 1 we maintain a set of counters to remember how many number of steps possibly remain to finish the execution of the input counterexample E. This can be time and resource consuming especially when we do not know whether E is real or not and when we want to find *any* unbounded executions. We propose a different heuristic function in which we discard the h_2 part and consequently the counters and the needed calculations. So, given any occurrence of a global state s in the reachability tree, now the heuristic function becomes $f(s) = g(s) + h_1(s)$ where g and h_1 are defined as for the Alg. 1.

The presence of the h_1 part in the above heuristic function implies that we still prefer to find an unbounded execution in which only the cycles in E are repeated infinitely often. So, h_1 is supposed to stick to the executions of the cycles in E. However, because of the lack of the h_2 part, h_1 fails to force the execution to stay with the executions of the cycles in E in the following situation: Consider a global state s that has two successors s_1 and s_2, where s_1 is reached by executing some transition in a cycle in E, and s_2 is reached by executing some process that contains no cycles in E. Apparently, $f(s_1) = g(s_1) + h_1(s_1) = g(s) + 1 + h_1(s)$ and $f(s_2) = g(s_2) + h_1(s_2) = g(s) + 1 + h_1(s)$. This is because $dist_p(s) = 0$ for any

process p that contains no cycles in E. Since s_1 and s_2 have the same heuristic value, it is possible that s_2 is chosen to be expanded before s_1. A straightforward solution is to add a h_3 part to the heuristic function where $h_3(s)$ is defined as follows: $h_3(s) = 1$ if s is reached by executing a process containing no cycles in E, otherwise $h_3(s) = 0$.

Finally, we may further discard g, and the search algorithm becomes a Best-First-Search based algorithm with the heuristic function $f(s) = h_1(s) + h_3(s)$.

In Sec. 6 we will evaluate the effectiveness of the above mentioned different heuristic functions and search algorithms on a number of case studies.

5 Heuristic Unboundedness Test for PROMELA Models

PROMELA is the input modeling language for the explicit state moder checker SPIN [7]. It possesses most salient features for modeling both synchronous and asynchronous concurrent systems, and is broadly used to model and analyze real-life software systems [13]. We discuss some important issues when applying the previously described unboundedness tests to PROMELA models. These issues are also present when applying the tests to other high-level modeling languages such as UML-RT. The root cause of these issues is that the sufficient unboundedness condition by Jeron and Jard is given on the level of CFSMs. It therefore does not specify the impact that message formats, the use of variables, and multiple instantiations of process classes as well as other high-level features of PROMELA have on the test.

Compound Messages, and Variables in Message Exchanging Statements. A message may consist of several components. As an example, consider the model in Fig. 3. Messages exchanged in the buffer `b1` carry a message body containing an integer variable along with the message type `req`. When the statement `b1?req(x)` is executed, it will remove the topmost message in `b1`, and store the value of the message body in the variable `x`. The statement is executable no matter which value the component has. Although it is easy to extend the unboundedness condition to handle compound messages, the presence of variables in message exchanging statements may affect the precision of the unboundedness tests. To illustrate this point, consider a finite execution of the model in which the `do` loop in each process is executed once. Before the loops are executed, the content of the buffer `b1` is `req(0)`. After the loops are executed, the content becomes `req(1)`. The sequence of sent messages $out_{\texttt{b1}}$ is `req(1)`. Apparently, `req(0).req(1)` is not a prefix of `req(1).req(1)`. So, the checking of the unboundedness condition fails. However, it is easy to see that the model is truly unbounded. The problem is that messages `req(0)` and `req(1)` should not be perceived as being different since they are not distinguished by the model anyway. One potential solution is to identify equivalence classes for messages exchanged in the model. An over-approximation of such equivalence relation can be obtained from message receiving statements. In our example, we can have

$\{\texttt{req(x)}|\texttt{x} \in \mathcal{Z}\}$ as an equivalence class. While checking the unboundedness condition, we consider `req(0)` and `req(1)` to be equivalent, which results in the satisfaction of the condition.

```
// defines the types of messages
mtype = {req, reply, log}
active proctype server(){
  do // loop
  :: b1?req(x); b2!reply(x);
  od;
}
```

```
active proctype client(){
  int x = 0;
  b1!req(x);
  do
  :: b2?reply(x) -> x++; b3!log; b1!req(x);
  od;
}
```

Fig. 3. A client-server PROMELA model.

Condition Statements. The presence of condition statements such as `x > 0` may spoil the soundness of the unboundedness tests. Consider a path r that satisfies the unboundedness condition. Suppose that r contains `x > 0`, and the value of `x` is always decreased by executing r. Apparently, r cannot be repeated exclusively forever. One solution is to modify the unboundedness condition such that, for any two global states to satisfy the condition, they must also agree on the values of all variables. However, this may be too coarse a solution. In the above example, if r always increases the value of `x`, then r can be repeated infinitely often.

Multiple Instantiations. In PROMELA a class of processes may have multiple instances at run time. Alg. 1 uses a set of counters for the transitions in the input counterexample to remember the number of times that each transition needs to be taken to complete the execution of the counterexample. When there are multiple instances of a same process class, the counter of a transition has no knowledge which particular instance executes the transition. This does not render the unboundedness tests unsound or lead to missomg any unbounded executions, but it results in a more conservative estimate of the cost to reach the goal. This in turn increases the number of times that the unboundedness condition is checked during the search.

6 Experimental Evaluation

We have built a prototype implementation of the discussed unboundedness tests, including the Jeron-Jard-test, for Promela models. Our implementation is based on the heuristics guided model checker HSF-SPIN [4]. We restricted our experimentation to the models listed in Table 1. These models do not reveal the Promela language features that would impede applicability of the unboundedness tests, c.f. Sec. 5. Since all of these models are of modest size, all search algorithms terminate almost in no time. We therefore compare mainly the memory consumption and the number of visited/expanded states.

The A^*_{conf} algorithm outperforms DFS for all models, except for GooDB and GARP, by expanding fewer states and finding unbounded executions with

Model	Alg.	#expanded	#stored	len. of gen. path	mem
Fig. 1	DFS	9	9	9	1885 KB
	BSF	31	43	7	1885 KB
	A^*_{conf}	7	11	7	2021 KB
	A^*_{any}	7	11	7	2021 KB
	$A^*_{any-h_2}$	30	46	7	2021 KB
	BEST	7	11	7	1885 KB
Jeron/Jard	DFS	27	27	8	1885 KB
(Fig. 1 in [8])	BSF	64	117	8	1885 KB
(with a real	A^*_{conf}	8	15	8	2053 KB
counterexample)	A^*_{any}	8	15	8	2053 KB
	$A^*_{any-h_2}$	64	117	8	2049 KB
	BEST	16	26	12	1885 KB
Jeron/Jard	DFS	27	27	8	1885 KB
(with a spurious	BSF	64	117	8	1885 KB
counterexample)	A^*_{conf}	499910	1124385	Out Of Memory	512 MB
	A^*_{any}	318	489	8	2049 KB
	$A^*_{any-h_2}$	162	293	8	2017 KB
	BEST	743	834	13	2149 KB
GooDB	DFS	35	35	37	1885 KB
(A model for an	BSF	250	305	36	2009 KB
Internet-based	A^*_{conf}	37	55	40	2017 KB
personal train	A^*_{any}	33	48	36	2017 KB
travel information	$A^*_{any-h_2}$	250	305	36	2017 KB
management application)	BEST	34	49	42	2017 KB
DTP	DFS	23	23	10	1885 KB
(available in	BSF	14	15	10	1885 KB
SPIN distribution)	A^*_{conf}	10	11	10	2021 KB
	A^*_{any}	10	11	10	2021 KB
	$A^*_{any-h_2}$	10	11	10	2021 KB
	BEST	10	11	10	1885 KB
GARP [16]	DFS	4	4	13	1885 KB
	BSF	14	72	13	1885 KB
	A^*_{conf}	4	14	13	2041 KB
	A^*_{any}	4	14	13	2041 KB
	$A^*_{any-h_2}$	9	43	13	2041 KB
	BEST	9	43	13	2041 KB

Table 1. The statistic data comparing six search algorithms: DFS (i.e., Jeron-Jard-test), Breadth-First-Search (BFS), Alg. 1 (A^*_{conf}), A^* to find any unbounded path with h_2 (A^*_{any}) and without h_2 ($A^*_{any-h_2}$), and Best-First Search (BEST).

shorter generating paths. For GooDB, even though A^*_{conf} returns an unbounded execution with a longer generating path, the found execution offers useful diagnostic information since it conforms to the input counterexample. How DFS performs depends on the ways in which DFS is implemented and and how the models are structured syntactically. For instance, when we switch the orders in which the definitions of the two processes appear in the textual representation of the model for Figure 1, DFS needs to expand more states and finds an execution with a 12 steps long generating path. A^*_{conf} always visits and expands fewer states than BFS for all models for which it finds an unbounded execution.

We intentionally provide a spurious counterexample as input to the checking of the Jeron/Jard model. As expected, A^*_{conf} is not able to find a conforming execution. Other heuristic algorithms searching for any unbounded execution are much more inefficient than DFS and BFS since they are misled by a spurious counterexample. BEST performs the worst in this case since the negative effect entailed by the spurious counterexamples is not compensated by the accumulated cost that has been spent due to the lack of the g part in the heuristic function.

Last, in case a real counterexample is given, A^*_{conf} and A^*_{any} perform comparably since A^*_{any} only checks the unboundedness condition more often. In particular, for GooDB, A^*_{any} is able to find an unbounded execution with a shorter generating path by expanding much fewer states than BFS.

7 Conclusions

We have presented a heuristic approach to the unboundedness test for CFSM systems based on the sufficient unboundedness condition suggested by Jeron and Jard [8].

In future work we plan to improve the applicability of the test by addressing the issues that currently impede its applicability to more general models. It may also be interesting to apply bounded scheduling techniques to CFSMs and other more general modeling formalisms, and to compare the results with the ones we have obtained. Finally, it seems worthwhile to investigate the application of the goal-path-searching variants of heuristic algorithms in testing, where the search for execution paths also is an important issue.

References

1. D. Brand and P. Zafiropoulo. On Communicating Finite State Machines. *Journal of the ACM*, 2(30):323–342, 1983.
2. P. Darondeau, B. Genest, and P. S. T. S. Yang. Quasi-Static Scheduling of Communicating Tasks. In *Proceedings of CONCUR '08*, volume LNCS 5201, pages 310–324, Berlin, Heidelberg, 2008. Springer.
3. E. Dijkstra. A Note on two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.
4. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.
5. A. Finkel and L. Rosier. A Survey on the Decidability Questions for Classes of FIFO Nets. In *Advances in Petri Nets 1988*, volume LNCS 340, pages 106–132, London, UK, 1988. Springer.
6. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, pages 100–107, 1968.
7. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
8. T. Jeron and C. Jard. Testing for Unboundedness of FIFO Channels. *Theoretical Computer Science*, 113(1):93–117, 1993.
9. L.E. Rosier and H.C. Yen. Boundedness, Empty Channel Detection, and Synchronisation for Communicating Finite Automata. *Theoretical Computer Science*, 44:69–105, 1986.
10. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for Message Buffer Overflows in Promela Models. In *Proceedings of 11th International SPIN Workshop*, volume LNCS 2989, pages 216–233, 2004.
11. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for the Boundedness of UML RT Models. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume LNCS 2988, pages 327–341, 2004.

12. C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, and A. Sangiovanni-Vincentelli. Schedulability Analysis of Petri Nets Based on Structural Properties. In *IEEE International Conference on Application of Concurrency to System Design*, pages 69–78. IEEE Computer Society, June 2006.
13. A. Lluch-Lafuente. Database of Promela models. `http://www.albertolluch.com/research/promelamodels`, checked 07.05.2009.
14. M. Gouda and E.M. Gurari and T.H. Lai and L.E. Rosier. On Deadlock Detection in Systems of Communicating Finite State Machines. *Computers and Artificial Intelligence*, 6(3):209–228, 1987.
15. O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In *Proceedings of CoopIS/DOA/ODBASE 2004*, volume LNCS 3291, pages 1502–1519. Springer, 2004.
16. T. Nakatani. Verification of group address registration protocol using PROMELA and SPIN. In *Proceedings of 3rd SPIN Workshop*, 1997. Available at `http://spinroot.com/spin/Workshops/ws97/nakatani.pdf`.
17. T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.
18. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.
19. W. Wei. *Incomplete Property Checking for Asynchronous Reactive Systems*. PhD thesis, University of Konstanz, 2008.

A Pseudocode for A^*_{conf} and `adjustCounters`

Algorithm 1: A^*_{conf} search algorithm.

```
   Data: M as a system of CFSMs, E as a counterexample
   Result: An unbounded execution of M that conforms to E
 1 insert (s_0, f(s_0)) to open ;                    /* g(s_0) = 0, h_2(s_0) = Σ_{t_j∈T} ct_j^0 */
 2 while open ≠ ∅ do
 3     open = open\{(v, f(v))} ;                     /* v has the smallest heuristic value */
 4     if h_1 = 0 then
 5         d = Σ_{t_j∈T} ct_j^0 − h_2(v);
 6         a = v;
 7         while d > 0 do
 8             d = d − 1;
 9             a = predecessor(a);
10             if unboundednessCondition(a, v) then
11                 return the found unbounded execution;
12             end
13         end
14     end
15     for each successor s of v do
           /* Suppose that s is reached from v by taking the transition t_i          */
16         if there is no global loop through s then
17             if h_1(v) > 0 then
                   /* All counters are unchanged                                     */
18             else
19                 if t_i ∈ T and ct_i(v) > 0 then
20                     ct_i(s) = ct_i(v) − 1;
21                     ct_j(s) = ct_j(v) ;          /* For each t_j ∈ T where j ≠ i */
22                 else if t_i ∈ T and ct_i(v) = 0 then
23                     adjustCounters(s);
24                 else if t_i ∉ T then
25                     ct_j(s) = ct_j^0 ;                    /* For each t_j ∈ T */
26                 end
27             end
28             g(s) = g(v) + 1;
29             h_1(s) = Σ_{p∈P} dist_p(s) ;        /* P is the set of processes in M */
30             h_2(s) = Σ_{t_j∈T} ct_j(s);
31             insert (s, f(s)) to open;
32         end
33     end
34 end
```

Algorithm 2: The function `adjustCounters`.

```
 1 ct_j(s) = ct_j^0 ;                                          /* for each t_j ∈ T */
 2 s' = s;
 3 while s' has a predecessor do
       /* Suppose that s' is reached from s'' by taking the transition t_k       */
 4     s'' = predecessor(s');
 5     if t_k ∉ T or ct_k(s) = 0 then
 6         break;
 7     else
 8         ct_k(s) = ct_k(s) − 1
 9     end
10     s' = s'';
11 end
```