Deadlock Detection in Algebraic Specifications for Concurrent Systems

Marcelo Sihman[1] and Mario R. F. Benevides[2]

[1] Department of Computer Science
Technion – Israel Institute of Technology
Technion City, Haifa 32000, Israel
sihman@cs.technion.ac.il
[2] Departamento de Ciencia da Computacao
Instituto de Matematica
Universidade Federal do Rio de Janeiro
Rio de Janeiro 21945-970, Brazil
mario@cos.ufrj.br

Abstract. In this work, we present an algorithm for deadlock detection in algebraic specifications for concurrent systems. The algorithm acts on a composed agent, specified in CCS (Calculus for Communicating Systems) or π-Calculus, which mostly possess some ports that are restricted to internal communication among its components. The algorithm simulates its execution step by step, where all the execution paths are tested. If there is some possibility that a component or all of them reach a deadlocked state, then this is detected by the algorithm. An advantage of this technique is that it determines all types of deadlock and their locations in the composed agent. Thus, when one executes a specification he will already be assured that it is deadlock free.

1 Introduction

In this work, we present an algorithm for deadlock detection in algebraic specifications for concurrent systems. A specification is simulated and if some kind of deadlock may occur during its execution, then this is detected. The algorithm acts on a composed agent in CCS or π-Calculus, which mostly possess some ports that are restricted to internal communication among its component agents. Its execution is simulated step by step, where all the execution paths are tested. In case there is some possibility that a component or all of them deadlock, then this is detected by the algorithm. An advantage of this technique is that it determines all the types of deadlock and their locations in the composed agent. Thus, when one executes its specification he will already be assured that it is deadlock free.

Sections 2 and 3 contain brief introductions to CCS and π-Calculus respectively. We present the algorithm and its application in two sections. In Section 4, we present the solution for specifications in CCS; initially explaining the algorithm and then demonstrating its application in two examples. In Section 5, we present the solution for specifications in π-Calculus; we explain the differences between the π-Calculus and CCS solutions, showing that the deadlock detection algorithm itself is basically

the same. In these sections, we present theorems that demonstrate that the presented algorithm always detects deadlock when this exists, and always finishes execution normally in the cases where there is no deadlock, signaling the absence of this property.

2 Calculus for Communicating System - CCS

Calculus for Communicating Systems (CCS) is a well-known formalism, proposed by Robin Milner [4], for specification of concurrent systems. It deals with interaction between agents through individual actions of communication. A CCS specification is a description of the behavior perceived by a system, in function of the communication events that occur. The expressions describing the agents' behavior can be built using the following operations:

Let P, Q be agents, x and y ports and L be a set of ports.
Prefix (.): Denote sequence.
Summation (+): P + Q means that the agent will behave as P or as Q.
Composition (|): P | Q is a system where P and Q can communicate through the complementary ports.
Restriction (/): P/L, means that the ports in L cannot communicate externally.
Relabeling ([ ]): P[x/y] means the agent obtained from P replacing all occurrence arbitrarily of y by x.

CCS is the inspiration of some algebraic based formalism like π-Calculus, and also of languages for formal specification like LOTOS. The reader is referred to [2] for an example of an application that makes use of CCS. Finally, the algorithm presented here could be easily adapted for most of these specification formalism and tools.

3 π-Calculus

π-Calculus is a processes calculus that aims to describe and analyze systems that consist of agents that interact concurrently with each other, and whose structure is in continuous process of evolution. These structure changes of a process can be naturally expressed using this calculus. The component agents of a system can be arbitrarily connected, but a communication between neighbors can also carry information with the power to change connections.

The development of π-Calculus was based on the studies of Engberg and Nielsen [3], who successively extended CCS to include mobility, while preserving its algebraic properties. The proposal of this calculus - presented in two joint articles by Milner, Parrow and Walker in [5], [6] - is to keep the essence of the work of Engberg and Nielsen, but to reduce its complexity and to fortify its elementary theory. In addition to the agent expressions seen in Section 2 for CCS, the following expressions are also used to describe an agent's behavior in π-Calculus:

Let P, Q, R be agents, x, y and z ports and L be a set of ports.

Negative prefix: $\bar{x}y.P$, $\bar{x}$ is an output port that transmits the name y and then behaves like P.

Positive prefix: x(z).P, x is an input port that receives some name y and then behaves like P{z/y}.

Restriction: (L)P, means that the ports in L cannot communicate externally, just as in CCS, but the operator is different.

Equality test: [x=y]P, behaves like P if x and y are identical, and like 0 (inert agent - can do nothing) otherwise.

We have a special interest in π-Calculus's scope extrusion law, which may be illustrated by the following example:

$$(y)(\bar{x}y.P \mid Q) \mid x(z).R \rightarrow^{\tau} (y)(P \mid Q \mid R\{y/z\})$$

where initially P has a private link y to Q, restricted to their internal communication. However, P passes y along its link x to R, where R possesses no y-link. R receives the message and replaces all occurrences of z by y, and now R is also connected to P via link y, and the y-link is private to all agents P, Q and R. When the private y-link is exported to R the scope of the restriction is extended, extruding the scope of the private y-link also to R.

4 Specifications in CCS

The algorithm operates over a CCS specification (a composed agent) with ports restricted to internal communication among its components, i.e., closed to the external environment. These are the actions that interest us, because messages that are expected in ports open to the environment may always arrive, since any external agent may communicate through an open port, then there is no way a deadlock may occur in this case. The same holds for silent actions, as the algorithm ignores these actions, not simulating them, since they can be executed and do not cause deadlock. However, note that a silent action can modify fairness and liveness properties of a computation, since it can become the only action possible of being executed, but this does not characterize a deadlock.

Concerning ports closed to the environment, if a message a component is waiting for cannot be sent by any component, then this will be in deadlock, and if all components are in this situation, then we have a global deadlock; i.e., all the components are waiting for messages that will never arrive.

The algorithm goes over the composed agent and simulates it from its initial state. The ports that are not restricted in the composition are not considered, and are ignored by the algorithm, which advances the simulation to the next restricted port. Therefore, the current action being checked in a component is always a restricted port.

The algorithm simulates the full execution of the composed agent, all its possible paths. When a deadlock is detected it finishes and signals this property; in case no deadlock is detected it signals the absence of this property. In the simulation, we use a transition diagram to represent the global states of the agent's execution. This diagram contains all the global states that may appear during the agent's execution. The simulation finishes when there are no more global states to visit (meaning all the possible execution paths have been tested successfully) or a deadlock is detected.

Starting from the initial global state, we have only one vertex in the diagram, which represents this global state. We check all the possible communications among the current ports (the first restricted ports that we encounter) of each component. For each possible state transition of the composed agent we add a corresponding edge - in the diagram – from the vertex representing the current state of the simulation to the vertex representing the next state, in case this transition is executed. This edge is labeled with the name of the simulated action. In the beginning, when there is only one vertex in the diagram, there is only one process running. For each new edge leaving this vertex we associate process; this is possible when performing a concurrent simulation, since we can simulate each branch in a separate process, checking each of its states sequentially. For each component, we test if that will succeed to communicate its current action or not. We use a wait graph to check if this state is in deadlock or not. There is a vertex for each component; initially there is no edge connecting them. One component is tested by time, each one has its corresponding wait graph. Considering a given component, we look for the complement of its current port in all other components.

For each complementary port found in the other components we add an oriented edge in the graph - labeled with the simulated action - between the two associated vertices (representing the component being tested and the other with the complementary port). If this complementary port is the current action in the associated component, then no other edge is added from this vertex. Otherwise, we recursively repeat this same process of searching the complements of the current action being checked and adding labeled edges in the graph between the associated vertices, until we get a cycle in the graph after the last edge addition performed (backtracking to a previously visited vertex); or when the complementary port is the current action in the associated component. We get to the final graph of a component when no more edges may be added from any of the vertices with incident edges, meaning all the paths (sequence of edges that are incrementally added) have finished. As already written, that means we got to the final graph, either when in the end of each path the complementary port that interests us is the current action of the associated component, or when this path turned to be a cycle.

It is important to emphasize that if - during the simulation - we come across a component whose current state is a summation (for example, A1 = a.A2 + b.A3 + c.A4, instead of something like A1 = a.A2), then when we build the graph for this component we create a vertex for each option of the summation (three vertices for the example seen) and connect each new vertex to the existing one with an unlabeled edge incident to the vertex. Thus, each new vertex will have the possibility to communicate only one action (and not three as in the example), there will be no summation with various options.

This will be the case if the option of the summation being treated is not a summation itself, as in C1 = (a + b).C2 + c.C3 + d.C4, since in this case the vertex representing component C is split in three new vertices, and the first vertex is split in two. After no more splits are possible to the current vertex, we try to communicate the only possible action, and then the edge is labeled as usual with the current action (in A1, for example, the edges that leave each one of the new vertices are respectively labeled with a, b, c).

If we get to a component (A) when we are treating another component (B) and building B's graph (e.g. when the current states of B and A are respectively B1 = $\bar{d}$.B2 and A1 = a.A2 + b.A3 + c.A4, where A2 = d.A5, the search for the complementary port of the B's current action $\bar{d}$ will return A), then we must add new vertices representing these options of the summation of A1 and connect them to the vertex that originated them (A1) with incident edges to the new vertices. In the example seen, a labeled edge from B1 to A1 is created in the graph, then we create three vertices, one for each option of the summation, and connect A1 (with an incident unlabeled edge) to each one of these; and then we connect each one of these new vertices (through a labeled edge) to the vertices where their complements were found. The labels for the edges leaving the new vertices are "a" for the first vertex created, "b" for the second and "c" for the third. The graph building then follows separately in each one of the branches that represent the options of the summation (in the example seen, in the three vertices created).

When finishing the graph building - for a component in some global state - we want to detect if this component will succeed to communicate its current action with another one in the sequel of the simulation; i.e., if this component is in deadlock or not. For this purpose we use a knot detection algorithm [7]. If the graph presents a knot then the associated component is in deadlock, and this simulation finishes. This happens because we known that when the wait graph for a resource (message) presents a knot, then the component will never have access to this resource, what characterizes a perpetual locking (deadlock).

In case that no knot is found in any of the graphs built for each component, then the simulation goes on from the current global state. For each possible transition from this state (each possible communication between components) an edge is added in the diagram connecting this state to the next (relative to each transition). If the next state already exists, then we only add the edge between the two (the current state and the next one); otherwise, we first have to create a vertex in the diagram to represent the new state. In the first case, when the next state already exists, the simulation for this path of the diagram finishes, since it came back to a state that was already visited and that already generated its associated branches that are being simulated. In the second case, we continue the simulation until we get to the first case (come back to an already visited state) or until a deadlock is detected. The information on which actions can be executed in one step - from the current global state until the next - are obtained from the wait graph being built for each component, when we are verifying if some of them is in deadlock.

We now explain how the search for complementary ports is performed. We take the current action of the component being simulated and look for its complement in all the other components from their current states. When a complement is found in a possible path of a component, the search finishes for this path (recall that if in the path from the current state until the complementary port we did not encounter any operator "+", then this path is unique; otherwise, if before finding the complementary port we come across with this operator, i.e., if we come across with a summation, then we will have exactly one different path for each of its options); otherwise, if this component finishes the search after returning to a previously visited state, that is because no complement was found in this component. Recall that if the complement of a

restricted port is not found in any other component then it cannot be communicated, and then we have a deadlock.

We now present the algorithm in a summarized form:

Algorithm 1:

```
NewGlobalState(vertex) {
  For each component
    build wait graph and check if there is knot in it
  if there is a knot in some wait graph
    finish simulation and signal deadlock
  For each next global state N (reachable by 1 transition)
    CheckState(N)
}
CheckState(N) {
    if N has an associated vertex in the global state diagram {
      connect current state C to N by an edge incident to N
    } else {
      create vertex n for N and connect it to c (for C)
      NewGlobalState(n)
    }
}
```

4.1 Examples

We present two examples that illustrate the application of our algorithm.

Example 1. $(A1 \mid A2 \mid A3 \mid A4)\backslash\{a_1, a_2, a_3, a_4, a_5\}$

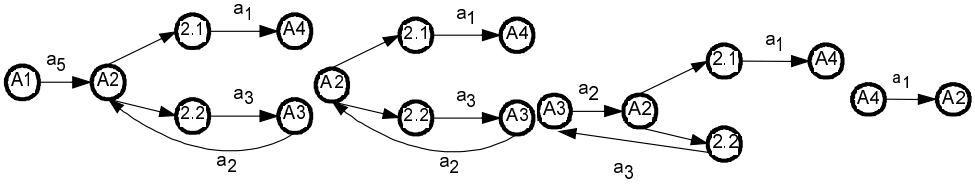
$A1 = a_5.A1$

$A2 = (a_1.\overline{a_2} + \overline{a_3}.a_4.\overline{a_5}).A2$

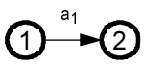
$A3 = a_2.a_3.A3$

$A4 = a_1.\overline{a_4}.A4$

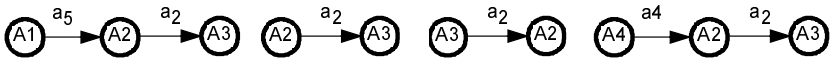
We first build the wait graphs for each one of the four components in the first global state. The corresponding graphs of each one are:



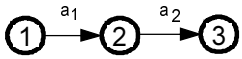
None of them presents a knot, then we simulate the only possible action (a1), we create an associated vertex in the global state diagram and connect it to the first one. Note that the vertex that represents component A2 in the wait graphs was split in two new ones, since its current state is a summation of two options.



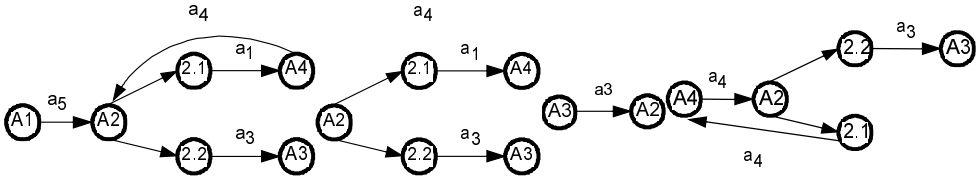
We now build the wait graphs for the second global state:



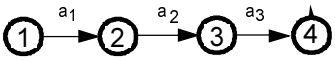
Once again we do not have a knot, we simulate the only possible action (a2), we create a corresponding vertex in the global state diagram and we connect it to the previous one.



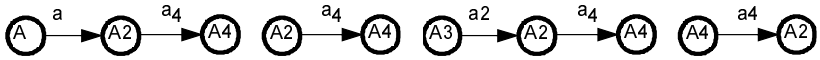
We then build the wait graphs for the third global state:



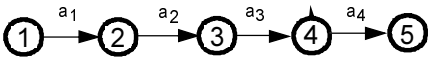
No graph has a knot; we simulate the only possible action (a3), create an associated vertex in the global state diagram and connect it to the previous one.



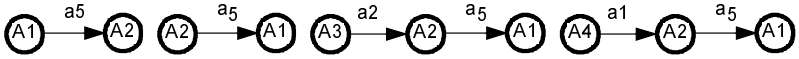
We build the wait graphs for the fourth global state:



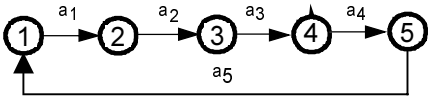
No graph has a knot; we simulate the only possible action (a4), create a corresponding vertex in the global state diagram and connect it to the previous one.



We build the wait graphs for the fifth global state:



No graph has a knot, we simulate the only possible action (a5), but when we try to create an associated vertex in the global state diagram we note that this was already created and visited, then we connect the vertex representing the fifth global state to the first one, which is the vertex representing the new current state, which was already visited. Thus, the simulation finishes signaling absence of deadlock.



Example 2. (A1 | A2 | A3)\{a_1, a_2, a_3, a_4}

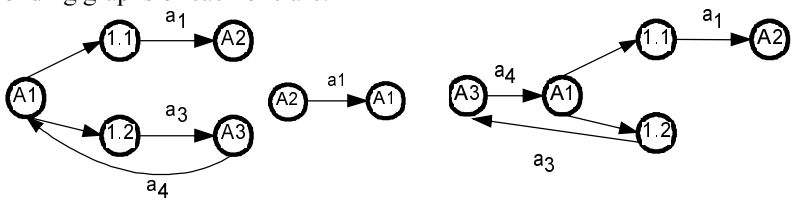
A1 = ($a_1.\overline{a_2}$ + $\overline{a_3}.a_4$).A1

A2 = $\overline{a_1}.\overline{a_4}.a_3$.A2

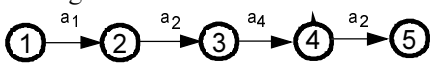
A3 = $a_4.a_2.\overline{a_3}$.A3

We first build the wait graphs for each one of the three components in the first global state. None of them presents a knot, then we simulate the only possible action

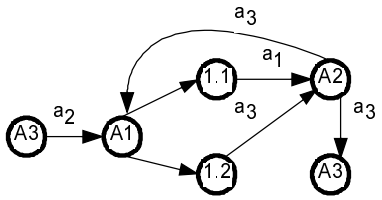
(a1), we create an associated vertex in the global state diagram and connect it to the first one. Note that the vertex that represents component A1 in the wait graphs was split in two new ones, since its current state is a summation of two options. The corresponding graphs of each one are:

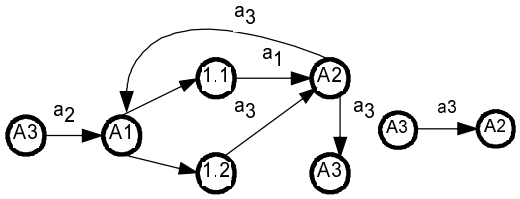


The execution follows normally and gets to the fifth global state, where we have the following global state diagram:

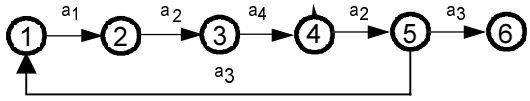


The wait graphs of each component are:

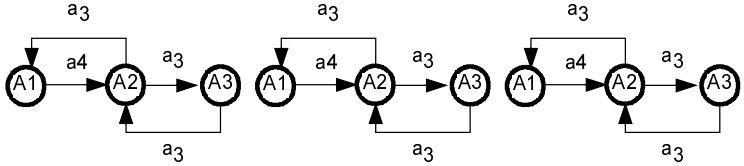




Two different communications can be executed through the edge a3, between A2, A1 and A2, A3. We then simulate the two possible actions, we create a vertex associated with the new state in the global state diagram and we connect it to the previous one. We connect the vertex associated with state 4 to the one of state 1, since the second option of communication makes us come back to state 1.



We build the wait graphs for the sixth global state, which is generated after the communication between A1 and A2 through a3:



Note that all the graphs are identical and present knots, and then the simulation finishes signaling deadlock.

When A2 communicates with A3 the execution follows perfectly. However, when A2 communicates with A1 in a3, instead of communicating with A3, all the components keep waiting for messages that will never arrive, therefore we have a deadlock.

Theorem 1. If there is no deadlock in the entered CCS specification, then Algorithm 1 always finishes its execution successfully, signaling the absence of deadlock.

Argument. Since there is no deadlock in the entered specification, then eventually each component will be capable to execute an action. Since the number of possible actions to be executed is finite, then so is the number of global states, and consequently we get back to a global state that was already visited for each branch of the computation.

This argument also holds for the case where each one of the components is not recursive, but finite, i.e., they finish to execute; the only difference in the argument is that now we do not get back to a global state for each branch of the computation, but all the branches of the computation finish the simulation successfully. ∎

Theorem 2. Algorithm 1 always finishes its execution signaling deadlock in the cases where the entered CCS specification presents this property.

Argument. Assume that the entered specification has a deadlock; this means that a group of components will reach a state where no action can be executed. In Algorithm 1, this implies that we will get to a state in the transition diagram where no action - of some component of this group - can be executed. However, this will occur only if this group gets to a knot; or if - for at least one component - the search of complements for all of its restricted actions (possible to be executed at the moment) returns nothing. ∎

5 Specifications in π-Calculus

The algorithm for deadlock detention in π-Calculus specifications is practically the same that was presented for CCS specifications with the operator "+", as there are only some small changes. This is surprising, since, due its dynamism, π-Calculus is a much richer calculus than CCS, since the last allows the transmission of ports as communication parameters, being able to modify connections between components dynamically. However, concerning a deadlock occurrence, its handling and detection, there are only few differences, if comparing CCS and π-Calculus specifications, as will be shown in this section. The only new property of this calculus that must be taken in account when detecting deadlock is scope extrusion. Because of this property, the algorithm now needs to check also open ports, since a deadlock may occur involving this kind of ports after a scope extrusion takes place. This situation is depicted in Example 5. A simple illustration of a deadlock that occurs due to a scope extrusion follows: a component (A) receives from another one (B) - through a

previously existing channel between the two - a private port as parameter; then this becomes private also to the component that received it (A) - in addition to all the others that shared it with the component that sent it (B). Thus, it is possible that the names that now represent this port - after the change of the associated names (since in x(y).y after receiving a name in x, y is changed to this name) - do not succeed to communicate with their complements or cause a deadlock in components that were functioning perfectly. This situation is presented in Example 5.

Basically, there is only one small change in the algorithm, at least in its body, its main part, as now we call the TreatKnot procedure when a knot is found in some wait graph, instead of signaling deadlock. It is clear that there are some changes in the part that deals with the step-by-step simulation of the actions of the composed agent, since in the π-Calculus ports may be passed as parameters in communications, what is not supported by CCS. Moreover, there are also certain differences with relation to the operators, as of restriction, which is different of the one considered in CCS. Thus, the part of the algorithm that deals with the simulation of the specification is considerably modified. On the other hand, the only modification in the part that deals with the deadlock detection is the addition of a code section that deals with scope extrusion.

The basic alteration that is needed to detect all possible deadlock is to treat also communications with parameters, mainly the ones that have ports as parameters, even the ones executed by ports opened to the environment. When we execute an action of this type the part of the algorithm that deals with the simulation of the composed agent has to make all the necessary changes in the associated names.

Back to the part of the algorithm that deals with the deadlock detection of π-Calculus specifications, the function representing this algorithm practically is the same as presented in Section 4 for CCS specifications, with only one difference.

Algorithm 2:

```
TreatKnot {
  if current action of the component is a restricted port {
    build graph with restricted ports and check for knot
    if there is a knot {
      search other comps for out ports that pass this action
      if the search returned an empty set {
        finish simulation and signal deadlock
      } else {
        visited = {component id, current action address}
        CheckPort
      }
    }
  } else {
    search other comps for output ports that pass this action
    if the search returned an empty set {
      leave this open port and advance to the next action
      CheckState(N) /* N is the new global state reached */
    } else {
```

```
      visited = {component id, current action address}
      CheckPort
    }
  }
}

CheckPort {
  finished = FALSE
  success  = FALSE
  update {component id, address of current action}
  While finished = FALSE
    if {comp id, address of current action} is in visited {
      finished = TRUE
    } else {
      visited = visited + {comp id, address of curr action}
      For each output port returned by search {
        build graph with for its comp and check for knot
        if there is a knot {
          build graph with restricted ports & check for knot
          if there is a knot {
            search comps for out ports that pass this action
            if the search returned an empty set
              finished = TRUE
          } else {
            finished = TRUE
            success  = TRUE
          }
        } else {
          finished = TRUE
          success  = TRUE
        }
      }
    }
  if success = FALSE
    if current action of the component is a restricted port {
      finish simulation and signal deadlock
    } else {
      advance this action and get to a new global state
      CheckState(N) /* N is the new global state reached */
    }
}
```

The difference is that besides the internal ports we also search the communications where parameters are passed. Maybe the channel that is transmitting the parameter is an open port, but this does not matter, since due to scope extrusion this communication may cause a deadlock. Therefore the search now also looks for ports that fulfill this condition.

Another modification in the algorithm takes place in the building of the wait graph, when the search for complementary ports - in NewGlobalState - returns an empty set we only assign true to the variable that represents that the graph has a knot, so that NewGlobalState calls the TreatKnot procedure to check if it has or not deadlock in this situation; since TreatKnot will call CheckPort, which verifies if this port's name is passed as parameter by some output port, allowing a possible scope extrusion, and verifies if this takes place and prevents the deadlock.

In this situation, when the complementary port is not found and we get a knot in the wait graph, it is possible that there is no deadlock, even if the port is restricted, while in Algorithm 1 this situation already characterizes a deadlock. This happens due to scope extrusion, since the restricted port may be passed as a parameter in some action, and then its complement may come out later. The same holds when there is a knot in some wait graph with only restricted ports, since a new port can come out after a scope extrusion and communicate with the port that generated the knot in the graph. The application of the algorithm in these cases is illustrated in Example 3.

In CheckPort, when we search a complementary port, there is a difference in the search if no complement is found. In this case, CheckPort is called to check if this port is not being passed as a parameter in another action, as its complement may then appear later in the simulation. That is also shown in Example 3.

These modifications are needed because of π-Calculus' scope extrusion law, which does not appear in CCS and is the only new law that can generate or prevent a deadlock. If we use Algorithm 1 to check a π-Calculus specification, not treating open ports that possess parameters, it may detect absence of deadlock when there is one, and vice-versa. This does not happen in Algorithm 2, since it simulates open ports with parameters.

5.1 Examples

The examples shown in this section aim only to show the power of the scope extrusion and how the algorithm treats this law. A specification that initially does not present deadlock, but possess open ports, may reach a deadlock state - because of a scope extrusion - when receiving a restricted port as parameter. In a similar way, a specification that reaches a deadlock may leave this state by means of scope extrusion when receiving a restricted port; but certainly only if it has open ports. We can affirm with certainty if a composed agent has a deadlock or not only if it possess no open ports that either receive or send a restricted port as parameter, since in these cases a scope extrusion may take place and generate a deadlock.

In the subsequent examples, we illustrate cases where we investigate the specification, initially there is no deadlock, but after a communication with the environment - scope extrusion - it comes to a deadlock. This situation is also shown in the opposing order.

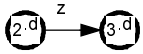
All the examples shown so far for CCS also fit π-Calculus, since the last contains all the functionality of the former.

Example 3. $A = (x)(x \mid \bar{z}y \mid z(t).t(u).\bar{u} \mid \bar{y}x)$.

When analyzing this agent it is not trivial to perceive that it is not in deadlock, since the port x is restricted and the input port input x does not have a complement, and then we can erroneously think that it is waiting for a message that does not arrive. However, x is passed as a parameter by port y in the fourth component. Nevertheless, no complement of y is found either, but y is passed as a parameter through port z, which has a complement. Thus, the communication is executed between the second and third components in channel z, and the port name t is replaced by y in the third component. The third and fourth components then communicate via y, and u is substituted by x in the third component. Finally, the first and third components communicate through x the execution finishes.

We now illustrate how the algorithm simulates this specification. At first there is only one vertex in the global states diagram and the wait graphs of this state are built for each one of the four components. When building the graph of the first component the algorithm does not find the complement of port x, afterward it assigns true to the variable that represents if there is a knot in the graph, and then the same happens in TreatKnot when building the graph of restricted ports, as it also contains a knot. Thus, CheckPort is called to verify if x is the parameter of some output port, what indeed happens, as x is the parameter of port y in the fourth component. CheckPort builds the wait graph for the fourth component, verifies if its current action (y) possess a complement, but none is found. It checks if the name of the current action (y) is the parameter of some output port, and the result is positive, since it appears as such in port z of the second component.

The complement of the current action of the second component (z) is the current action of the third, thus the creation of the wait graph built finishes without knots, with only one edge between the vertices representing the second and third components.



The construction of the wait graph for the second and third components is direct, and these do not present knots, since their current ports are complementary and the wait graph is alike to the one just shown. Meanwhile, the creation of the wait graph for the fourth component repeats the last steps of the first component's graph creation. As no deadlock was detected, we simulate the only possible action (z), create a new vertex related to this global state in the diagram, and this is connected to the first vertex through a labeled edge z.

In this global state, the creation of the wait graphs repeats the last steps seen in the creation of these for the first global state. The second component finishes execution, and the graphs of the third and fourth components are constructed directly, as their current actions are complementary. These graphs have an edge between the vertices that represent these two components, thus there is no knot. After creating the vertex related to the next state and connecting it to the previous, we create the wait graphs for the first and third components, which contain only one edge between the associated vertices, since their current actions are complementary. Therefore, there is

no knot, the composed agent finishes its execution and the algorithm detects the absence of deadlock.

Example 4. $A = (x)(\bar{y}x.z(a) \mid x(b).\bar{u}c)$.

It is easy to see that the agent is in deadlock, since port x is restricted and the input port x does not possess a complement, thus waiting for a message that does not arrive. Algorithm 2 would detect this deadlock.

We now analyze agent B:

$B = A \mid y(v).\bar{v}t$.

A communication for port y takes place between B's two components, where v becomes x, which becomes an internal port of the two components due to scope extrusion, since x is already internal to A. Thus, the communication through port x takes place between the two components and the simulation finishes without deadlock, since B remains with only open ports, which do not have complements into the composed agent.

Example 5. $A = (x)(\bar{y}x.z(a) \mid x(b).\bar{u}c \mid \bar{x}d)$.

A is not in deadlock, since the communication through its restricted port x takes place normally, as port x of the first component has its complement in the third.

We now treat agent B:

$B = A \mid y(v).\bar{v}t$.

In contrast to the previous example, the addition of the last component to A generates a deadlock, since y will receive x as parameter, which becomes an internal port due to scope extrusion. The problem is that now we have two output ports and only input port for x, hence an output port will be waiting forever to send a message. Therefore, a deadlock is detected.

Example 6. $A = (x)(\bar{y}x.\bar{x}a \mid x(z).\bar{z}c)$.

As in the previous example, when analyzing this agent it is easy to see that it is not in deadlock, since the communication through its restricted port x takes place normally, as port x of the first component has its complement in the second. No deadlock would be detected.

We now treat agent B:

$B = A \mid y(z).\bar{z}b$.

Similarly to Example 5, the addition of the last component to the agent A generates a deadlock, since y will receive x as parameter, which becomes an internal port. The problem is that now once more we have two output ports and only input port for x, hence an output port will be waiting forever to send a message. Therefore, a deadlock is detected.

Example 7. $A = (x)(\bar{y}x.\bar{z}a \mid x(b).\bar{b}c)$ and

$B = A \mid y(v).\bar{v}t$.

This example shows a case where initially we have a deadlock, but that is reverted with the addition of another component; further, we may guarantee that the composed agent will never reach a deadlock.

A is in deadlock, as already shown in Example 3. In B, with the communication through y and then x (which replaces v in the second component of B) we have

$B = (x)(\bar{z}a \mid \bar{t}c \mid 0)$.

It is not possible that B reaches a deadlock, since scope extrusion may happen in B. This agent finishes successfully after communicating z and t with some another agent of the environment.

Theorem 3. If there is no deadlock in the entered π-Calculus specification, then Algorithm 2 always finishes its execution successfully, signaling the absence of deadlock.

Argument. Same of Theorem 1, since the only difference between the two algorithms is the call to the TreatKnot procedure when the wait graph contains a knot. This procedure always finishes, when variable finished is assigned true, with variable success containing false (when we detect the impossibility of a communication between a port and its complement, or when its complement is not found and this port is not passed as a parameter through any output port) or true (when the created wait graph does not contain a knot). ∎

Theorem 4. Algorithm 2 always finishes its execution signaling deadlock in the cases where the entered π-Calculus specification presents this property.

Argument. Assume that the entered specification has a deadlock; this means that a group of components goes will reach a state where no action can be executed. In Algorithm 1, this implies that we will get to a stage in the transition diagram where no action - of some component of this group - can be executed. However, this will occur only if: first, this group gets to a knot; or if - for at least one component - the search of complements for all of its actions (possible to be executed at the moment) returns nothing. In addition to one of these conditions, we also have to check if new complements of the current action may appear, what perhaps is possible when this is the parameter of an output port. This possibility is verified, and only when no complementary port - that may revert this deadlock - comes out we get to a situation where no action can be executed. ∎

6 Conclusion

In this work we propose an algorithm for deadlock detection in algebraic specifications for concurrent systems. A specification is simulated and if some kind of deadlock may occur during its execution, then this is detected. The algorithm acts on a composed agent in CCS or π-Calculus, which mostly possess some ports that are restricted to internal communication among its component agents. Its execution is simulated step by step, where all the execution paths are tested. In case there is some

possibility that a component or all of them deadlock, then this is detected by the algorithm. An advantage of this technique is that it determines all the types of deadlock and their locations in the composed agent. Thus, when one executes its specification he will already be assured that it is deadlock free.

We introduce the new algorithm and some examples that illustrate its application both for CCS and π-Calculus. We explain the differences between the solutions for π-Calculus and CCS, showing that the deadlock detection algorithm itself is basically the same. We present theorems that demonstrate that the proposed algorithms always detect deadlock when this exists, and always finish execution normally in the cases where there is no deadlock, signaling the absence of this property.

As future work, we would like to finish the implementation of the algorithms and make them available to every one that needs to test specifications with relation to the property of deadlock. We also intend to improve the algorithm with regard to its efficiency, diminishing the number of wait graphs created and tests performed; in short, to reduce its computation. We think we may reach these results using a new graph-theoretic strategy.

References

1. V. C. Barbosa and M. R. F. Benevides. *A Graph-Theoretic Characterization of AND-OR Deadlocks*, Technical Report, COPPE-Sistemas, UFRJ, ES-472/98, Jul. 1998.
2. M. R. F. Benevides and M. Sihman. Automatic Generation of CCS Specifications for Resource Sharing Problems. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 111-122, Chantilly, France, Dec. 1997.
3. U. Engberg and M. Nielsen. *A Calculus of Communicating Systems with Label-Passing*. Technical Report, Computer Science Department, University of Aarhus, DAIMI PB-208, 1986.
4. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
5. R. Milner, J. G. Parrow and D. J. Walker. A Calculus of Mobile Processes, Part I. *Information and Computation*, 100(1):1-40, Sep. 1992.
6. R. Milner, J. G. Parrow and D. J. Walker. A Calculus of Mobile Processes, Part II. *Information and Computation*, 100(1):41-77, Sep. 1992.
7. J. Misra and K. M. Chandy. A Distributed Graph Algorithm: Knot Detection. *ACM Transactions on Programming Languages and Systems*, 4(4):678-686, Oct. 1982.