Specification-based Testing of Concurrent Systems

Andreas Ulrich[a], Hartmut König[b]

[a] Fakultät für Informatik, Otto-von-Guericke-Universität, PF 4120, 39016 Magdeburg, Germany; E-mail: ulrich@cs.uni-magdeburg.de
[b] Fakultät für Informatik, BTU Cottbus, PF 101344, 03013 Cottbus, Germany; E-mail: koenig@informatik.tu-cottbus.de

The paper addresses the problem of test suite derivation from a formal specification of a distributed concurrent software system by presenting a concurrency model, called *behavior machine*, and its construction algorithm from a collection of labeled transition systems. It outlines how test derivation can be based on the new concurrency model to derive test suites that still exhibit concurrency between test events. A toolset is presented to support the generation of concurrent test suites from specifications given in the formal description technique LOTOS. Finally, some comments on requirements for the design of a distributed test architecture are given.

Keywords: Distributed concurrent software systems; conformance testing; test derivation; labeled transition systems; Petri nets.

1 INTRODUCTION

Testing is an important phase in the development cycle of software. A challenging problem is the derivation of test suites that are able to firmly detect faulty implementations of a system. Driven by requirements in testing telecommunication systems, approaches were developed to assist the automatic derivation of test suites [ADL+91] [Fer96]. These approaches are usually based on a finite description of the behavior of the system, mostly the model of a finite state machine, that is also exploited in the verification phase of the system. However, current test derivation approaches only support the derivation of test suites for sequential systems. One reason is that they are faced with computational problems due to state explosion if they resolve specified concurrent behavior in an interleaving sequence of actions of a derived test suite. Furthermore, its execution in a standard black-box test architecture might be not sufficient to assess conformance of truly concurrent systems since the message exchange between components of the concurrent system must be observed and controlled by a tester in addition in order to avoid nondeterministic test runs.

This paper continues work on the use of partial orders for test suite derivation of concurrent systems. It improves the previous work done in [UlCh95] and also other known work on this subject, e.g. [LSK+93], [KCK+96], by providing a sound concurrency model that can be constructed automatically from a collection of communicating labeled transition systems (LTSs). The new concurrency model, called *behavior machine* (BM), is an interleaved-free and finite description of concurrent and recursive behavior. The construction algorithm works as follows. First, the LTSs are mapped into a single Petri net representing the system. This Petri net is further used to construct its *unfolding*, another Petri net with a simpler structure, using an algorithm from [ERV96]. The behavior machine is then constructed from the finite prefix of a Petri net unfolding. An early version of the construction algorithm is given in [Ulr97].

After the behavior machine is introduced as description model of concurrent behavior, the test derivation approach is extended to support the new model. It is shown how an extension of the transition tour can be derived using algorithms already known from sequential systems. First results of the new testing approach that were obtained with a prototype implementation supporting specifications in Full LOTOS are discussed.

The paper is organized as follows. Section 2 introduces the model assumptions on a concurrent system. Section 3 sets up the new concurrency model. Some Petri net notions are explained in Section 4 that are needed for an easy understanding of the construction algorithm. Section 5 presents the algorithm to compute a behavior machine. In Section 6, the behavior machine serves as model for test derivation. An algorithm to derive a test suite from a behavior machine is presented. Section 9 discloses first results of a prototype implementation using a larger example of a concurrent system, different realizations of leader election algorithms from [GaMo96], and finally, Section 8 explains concepts of the design of a distributed test architecture.

2 A MODEL FOR DISTRIBUTED CONCURRENT SYSTEMS

We consider distributed concurrent software systems consisting of a collection of software modules running on different host machines and connected through a computer network. Each module is implemented as a sequential unit realizing a certain function of the system. Modules communicate synchronously via interaction points. The synchronous communication pattern fits the properties of programming languages for concurrent systems, e.g. Ada, and function calls in high-level network programming, like remote procedure calls, which are used, for instance, in the middleware platform CORBA.

Starting point of our investigations is a formal specification that defines the desired behavior of the concurrent system. Sequential behavior of a module in a concurrent system is modelled as a *labeled transition system* (LTS). The model of a LTS is an abstraction that focuses on interactions of a module with other modules in the system and/or with its environment.

Definition (1): A *labeled transition system* (LTS) is defined by the quadruple $(S, A, \rightarrow, s_0)$, where S is a finite set of states; A is a finite set of actions (the alphabet); $\rightarrow \subseteq S \times A \times S$ is a transition relation; and $s_0 \in S$ is the initial state.

A concurrent system $\Im = M_1 \parallel M_2 \parallel \ldots \parallel M_n$ is composed from a fixed number of communicating LTSs M_i. A composite machine $C_\Im$ of $\Im$ (also an LTS) is expressed by means of a composition operator $\parallel$ similar to that used in CSP. $P \parallel Q$ is the parallel composition of modules P and Q with synchronization of the actions common to both of their alphabets and interleaving of the others. The parallel composition $P \parallel Q$ of two LTSs $P = (S_1, A_1, \rightarrow_1, s_1)$ and $Q = (S_2, A_2, \rightarrow_2, s_2)$ is defined as a composite LTS $(S, A, \rightarrow, s)$, where $S \subseteq S_1 \times S_2$, $A \subseteq A_1 \cup A_2$, $s = (s_1, s_2)$, and the transition relation $\rightarrow$ is given as follows: If $P \text{–}a\rightarrow_1 P'$ then $(P \parallel Q) \text{–}a\rightarrow (P' \parallel Q)$ if $a \notin A_2$. If $Q \text{–}a\rightarrow_2 Q'$ then $(P \parallel Q) \text{–}a\rightarrow (P \parallel Q')$ if $a \notin A_1$. If $P \text{–}a\rightarrow_1 P'$ and $Q \text{–}a\rightarrow_2 Q'$ then $(P \parallel Q) \text{–}a\rightarrow (P' \parallel Q')$ if $a \in A_1 \cap A_2$.

3 A CONCURRENCY MODEL

The representation of concurrent behavior in a composite machine is accomplished by a tedious repetition of concurrent actions in order to construct all possible total orders. However, concurrent actions are independent to a certain extent from their occurrence in a total order. Instead of interpreting causality information in an interleaved-based model, we apply the notion of a *labeled partially ordered set* and its extension to a *partially ordered multiset*, which are interleaved-free representations of concurrent behavior [Pra86].

Definition (2): An *lposet* (*labeled partially ordered set*) is defined by the quadruple $(E, A, \leq, l)$, where E is a set of event names; A is a set of action names; $\leq$ is a partial order expressing the causality information between events, i.e. $e \leq f$ if event e precedes event f in time; $l: E \rightarrow A$ is a labeling function assigning action names to events. Each labeled event represents an occurrence of the action labelling it, with the same action possibly having multiple occurrences.

A *pomset* (*partially ordered multiset*) is an isomorphism class over event renaming of an *lposet*, denoted $[E, A, \leq, l]$. A *process* describing the behavior of concurrent system $\Im$ is a set of pomsets where each pomset describes a possible execution sequence of concurrent actions. Since the behavior of a system is frequently infinite due to recursive parts in the system description, the pomsets of a process are infinite, too. If branching occurs in a process, the set of pomsets forms an infinite *pomtree* [PLL+91], where an arc in the pomtree is an lposet or a concatenation of lposets, and a vertex is a branching point of the process (see Figure 3).

Since the construction of the composite machine from a set of communicating LTSs is not feasible in many cases due to state explosion, it follows that we need a new model that combines the advantages of both concepts: true concurrency between actions as preserved in an lposet and finiteness of the description as preserved in a LTS. This model is a *behavior machine* (BM), a similar model to the one introduced in [PLL+91]. However, the main advantage of a behavior machine is that it can be constructed automatically, as it will be shown in the paper.

Definition (3): The *behavior machine* of concurrent system $\Im$ is a quadruple $BM_{\Im} = (G, LPO, T, g_0)$ consisting of a finite set of global states G, where each element of G is an n-tuple of local states of all LTSs of $\Im$, i.e. $G \subseteq S_1 \times \ldots \times S_n$; a set of finite lposets LPO representing concurrency in $\Im$; a concurrent transition relation $T \subseteq G \times LPO \times G$ that maps a start state to an end state by performing the actions of the corresponding lposet; and an initial global state $g_0 = (s_1, \ldots, s_n) \in G$.

A global state of behavior machine $BM_{\Im}$, excluding its initial state, expresses always a branching point or a recurrence point within concurrent system $\Im$. A branching point is a global state where further behavior of the system branches off. A recurrence point is a global state where the behavior of the system repeatedly continues. An lposet in $BM_{\Im}$ is constructed in such a way that it connects always two global states of $BM_{\Im}$ by a concurrent transition. A pomtree can be obtained from $BM_{\Im}$ if its concurrent transitions are unrolled. In this case, branching points in the behavior machine correspond to branching points in the pomtree, whereas recurrence points are skipped. Thus, unrolling of a behavior machine is similar to the construction of a spanning tree from a directed graph.

Let t_1 an t_2 be two concurrent transitions of $BM_{\Im}$ with $t_1 = (g_1, lpo_1, g_2)$ and $t_2 = (g_2, lpo_2, g_3)$. The operation $t_1 \oplus t_2$ expresses concatenation of the two concurrent transitions. Concatenation is carried out in the way that each local state of the end state g_2 in t_1 is connected with the same local state of the start state g_2 in t_2. That means, the lposets lpo_1 and lpo_2 are merged according to the causal dependencies between their events.

Consider the simple system $\Im = A \parallel B$ whose LTSs are given in Figure 1. Under the assumption that actions a and c in each LTS synchronize, removal of parallel operator $\parallel$ by applying interleaved-based semantics rules yields the composite machine $C_{\Im}$. Figure 2 shows the behavior machine of system $\Im$. It contains three global states $\{S_0, S_1, S_2\}$ and four concurrent transitions $\{t_1 – t_4\}$ and describes the same behavior of system $\Im$ as given in Figure 1. Each concurrent transition is described by an lposet that exhibits concurrency among actions (see transition t_4). If the behavior machine is unrolled, the pomtree of Figure 3 is obtained. The process of unrolling exhibits the full degree of concurrency between events. For instance, if transitions t_2 and t_4 are concatenated, we realize that event b is concurrent to event e.

Although the behavior machine in Figure 2 is not the smallest representation of concurrent behavior caused by its construction algorithm discussed below, it is still a very compact representation of concurrent behavior. In this specific example, action *a* is redundantly represented within the concurrent transitions t_1 and t_4 what can be avoided in the minimal description. Furthermore, a behavior machine is able to distinguish concurrency from branching. This knowledge is lost in the composite machine $C_{\mathfrak{I}}$.

FIGURE 1. LTSs *A* and *B*, and the composite machine $C_{\mathfrak{I}}$ of system $\mathfrak{I} = A \parallel B$.

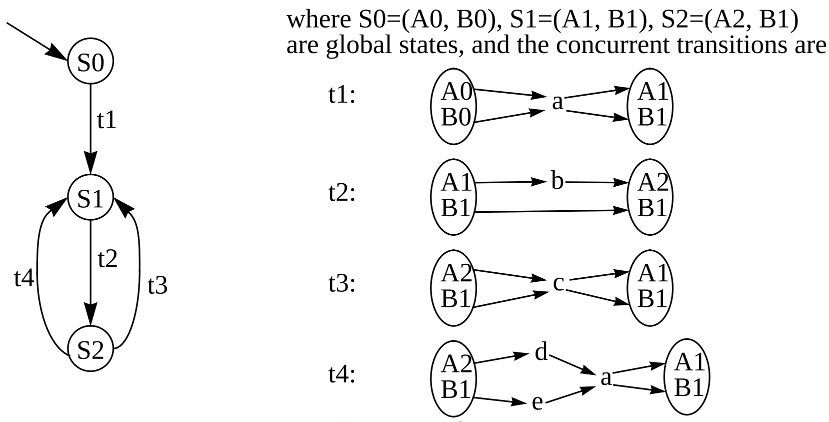

FIGURE 2. The behavior machine $BM_{\mathfrak{I}}$ of system $\mathfrak{I}$.

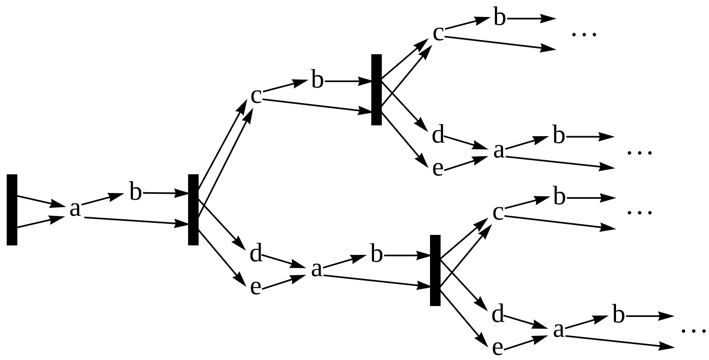

FIGURE 3. A pomtree of system $\mathfrak{I} = A \parallel B$.

4 PETRI NET CONCEPTS

The construction algorithm of a behavior machine is based on a Petri net description of the concurrent system. In [McM95] and [ERV96], a verification approach was described that is based on the technique of net unfolding, a partial order semantics of Petri nets. The unfolding of a Petri net is another (usually infinite) net with a simpler structure. The proposed algorithms in both papers aim at constructing the initial part of the net unfolding that contains all reachable states of the original net, called the *finite complete prefix*.

A net is a triple (S, T, F), where S is the set of places, T is the set of transitions, $S \cap T = \emptyset$, $F \subseteq (S \times T) \cup (T \times S)$ is the flow relation, If $M: S \to N$ is a marking of a net (N denotes the set of non negative integers), the 4-tuple $N = (S, T, F, M)$ is called a Petri net [Rei91]. The unfolding of a Petri net is a (unmarked) net (B, E, F), where B is the set of conditions, E is the set of events, with the properties: (1) it is an acyclic graph, (2) if two events (transitions) $e_1, e_2 \in E$ of the unfolding are in conflict, meaning that they are enabled from the same condition (place), then there exist two paths leading to e_1 and e_2 that start at the same condition and immediately branch off from another, (3) the nodes in the unfolding have a finite number of predecessors, and (4) no event is in self-conflict.

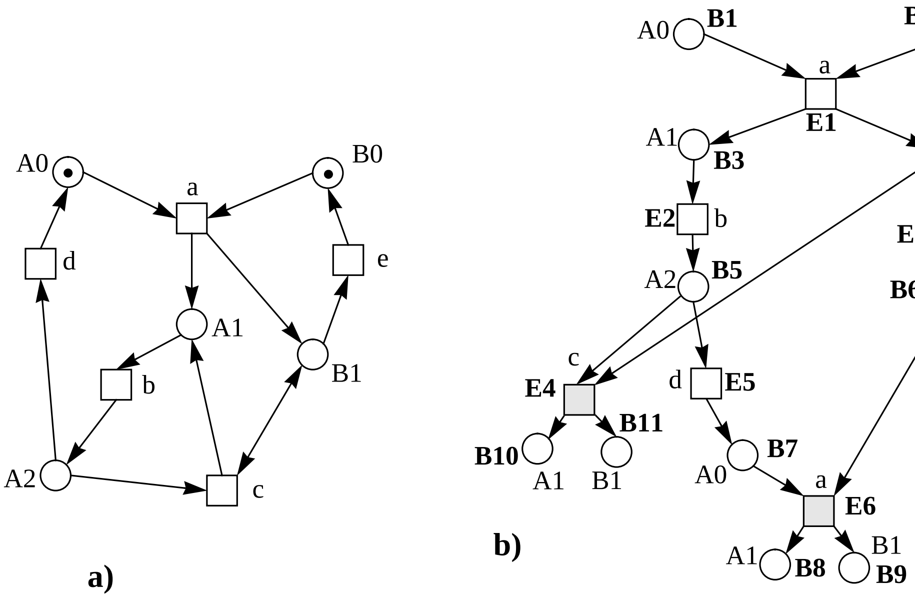

FIGURE 4. The marked Petri net of system $\Im = A \parallel B$ (a) and its unfolding (b).

Figure 4 depicts the Petri net description of system $\Im$ and the initial part of its unfolding. Note that the unfolding is not finite. For instance, if event **E4** is performed, the unfolding continues by substituting place **B10** with **B3** and **B11** with **B4**, respectively. This applies similarly to event **E6**. Events **E4** and **E6** are also in conflict. The branching point where the paths leading to the events branch off from another is the marking {A1, B1}, represented by the place set {**B5**, **B4**}.

A *local configuration* $[e]$ of event e in the unfolding describes a possible partially ordered run of the system which executes event e as its last event. It is a set of events satisfying the following two conditions: (1) if any event is in the local configuration, then so are all of its predecessors, and (2) a local configuration is conflict-free. The local configuration captures the precedence relation between events. Any total order on these events that is consistent with the partial order is an allowed totally ordered run of the system. Throughout the paper, we use the notions *local configuration* and *configuration* interchangeable.

To compute the finite complete prefix of a Petri net, it is necessary to define a break-off condition to stop the construction of the unfolding. This is done by introducing *cut-off events*. An event e is a cut-off event if the local configuration $[e]$ belonging to event e reaches a marking *Mark*($[e]$) in the unfolding that was reached before by a smaller local configuration of a different event.

Consider the unfolding in Figure 4b. The configuration of event **E6** is the set of events [**E6**] = {**E1**, **E2**, **E3**, **E5**, **E6**}. The reachable marking of this configuration is *Mark*([**E6**]) = {A1, B1}. This marking was reached before, however, by configuration [**E1**] = {**E1**}. We say, event **E6** corresponds to event **E1**. Since the configuration of **E1** has fewer elements than the configuration of **E6**, it follows that **E6** is a cut-off event. The second cut-off event is **E4**.

Proposition (1): Given the unfolding of a 1-save Petri net as it is obtained if the net is constructed from a set of communicating LTSs describing a concurrent system. Any deadlock-free system is completely represented by the finite set of tuples of cut-off and corresponding events $\{(e_{\text{cutoff}_1}, e_{\text{corresp}_1}), (e_{\text{cutoff}_2}, e_{\text{corresp}_2}), \ldots, (e_{\text{cutoff}_n}, e_{\text{corresp}_n})\}$, where the two configurations of events in a tuple reach the same marking, $Mark([e_{\text{cutoff}_i}]) = Mark([e_{\text{corresp}_i}])$.

The proposition requires cut-off events for each execution branch in the net unfolding. Their existence was proven in [ERV96]. A deadlocking system, however, reaches a final marking that does not relate to a cut-off event. The construction algorithm of a behavior machine is currently restricted to concurrent systems without deadlocks. This is, however, not a restriction for the purpose of test derivation since we usually require that the specification of a system had been verified to be deadlock-free before it is implemented and finally tested.

5 CONSTRUCTION ALGORITHM OF A BEHAVIOR MACHINE

The first step in constructing a behavior machine from a set of communicating LTS is a transformation of the LTSs into a global Petri net. After the transformation, the unfolding algorithm is applied to unfold the Petri net and to construct the set of pairs of cut-off and corresponding events. Finally, the behavior machine is constructed from the unfolding.

Constructing a Petri net from a set of communicating LTSs is simple. The following algorithm is applied: first, each single LTS is transformed into a Petri net; then, all Petri nets are merged according to the synchronization constraints in order to obtain a global Petri net. If the same action name labels several transitions in several LTSs, then a net transition for each allowed way of synchronization has to be constructed in the global Petri net. The transformation was already presented in [GaSi90] and is used in the *Cæsar/Aldebaran* toolset that supports verification of specifications given in the formal description language LOTOS.

Figure 4a shows the Petri net constructed from the two LTSs in Figure 1. The next step is the construction of the finite complete prefix. This is done by applying the algorithm presented in [ERV96]. Figure 4b depicts the prefix of the example system. The last step, the construction of the behavior machine, is described below.

We assume that the finite complete prefix of a Petri net unfolding, including the set of cut-off and corresponding events, is given. The local configuration of an event describes an execution path through the behavior machine from the initial state to this particular event. The marking reached by the configuration of an event defines a global state in the behavior of a concurrent system. The reachable marking of a configuration can be identified with places in the unfolding that are reached if all events in the configuration are executed.

When the behavior machine is constructed, it is not necessary to compute all reachable markings in the unfolding. Instead, only those reachable markings have to be known that are recurrence or branching points. Cut-off events and events corresponding to them define recurrence points of the behavior machine. Yet, branching points have to be computed.

To identify the branching points, we do the following considerations. Given the finite complete prefix of an unfolding, each local configuration of a cut-off event or a corresponding event starts in the initial state of the system, i.e. the initial marking, and ends in a marking reached by the configurations of those events. Since the finite complete prefix covers all reachable states of the system, branching points exist only somewhere inside the configurations of cut-off and corresponding events. If we analyze any two configurations $[e_1]$ and $[e_2]$ with $e_1 \neq e_2$ from the same unfolding, we realize that the configurations start with a same subset of events and branch off from another after a certain event e_{branch} occurred in both configurations. Now, a branching point can be defined exactly by the reachable marking of the configuration formed by this event e_{branch} assuming that $e_{\text{branch}} \in [e_i]$; and $[e_{\text{branch}}]$ is the maximal configuration that holds the condition $[e_{\text{branch}}] \subseteq [e_i]$, with $i = \{1, 2\}$.

This observation leads to the construction algorithm of a behavior machine. It takes as input the set of cut-off events and corresponding events that are contained in the finite complete prefix of an unfolding. The idea of this algorithm is to construct the configurations of the given cut-off and corresponding events first. Then, the events in the configurations are analyzed in order to identify the branching points.

```
1   • let E be the set of cut-off events and corresponding events in a finite complete prefix;
2   • let ℰ initially be the set of configurations from all events in E, i.e. ℰ = {[e1], [e2], …};
3   forall configurations [e] ∈ ℰ do
4       forall events d ∈ [e] with d ≠ e do
5           if (([d] ∉ ℰ) AND (successors(d) are branching places)) then
6               • mark d as branching event;
7               ℰ = ℰ ∪ {[d]};
8           end
9       end
10  end
11  forall configurations [e] ∈ ℰ do
12      forall configurations [d] ∈ ℰ do
13          if ((|[e]| < |[d]|) AND ([e] ⊆ [d])) then
14              • mark [e] if it is a maximal configuration contained in [d];
15      end
16  end
17  forall configurations [e] ∈ ℰ do
18      if ((e is a branching event) AND
            ([e] is marked as a maximal configuration less than twice)) then
19          ℰ = ℰ \ {[e]};
20  end
21  return ℰ;
```

ALGORITHM 1. Generation of configurations represented in the behavior machine.

As discussed above, a branching point is defined by the reachable marking of a configuration of maximum size contained within two or more other configurations. To identify these points, we analyze the successor places of an event *e*. If at least one of the successor places has more than one successor event, the reachable marking of the configuration [*e*] might be a branching point in the behavior machine. Since this result is obtained from a local analysis of a single event rather than from an analysis of the global system, not all events found refer really to a branching point. Algorithm (1) takes into account this aspect and returns only those events and their configurations that will be finally considered in the construction of a behavior machine.

The initial set of configurations $\mathcal{E}$ is obtained from the configurations of cut-off and corresponding events contained in the prefix of the unfolding (line 2 in Algorithm (1)). In the next step, further configurations of events are added to $\mathcal{E}$ if these events possess successor places that cause local branching (lines 3–10). The third step (lines 11–16) determines whether a configuration is contained in another one and marks the maximal configuration that fulfills this property. The final step (lines 17–20) deletes configurations of events added to $\mathcal{E}$ before if they are not marked as maximal configurations or if they are marked only once in another configuration. That means, configurations that do not determine a branch in the behavior are omitted in the construction of the behavior machine.

Algorithm (1) returns the set of configurations $\mathcal{E}$ relevant in the construction of the behavior machine, i.e., the configurations have the property that they reach the marking of a recurrence point or a branching point. In Algorithm (2), this knowledge is used to construct the global states and the concurrent transitions of the behavior machine of a concurrent system.

In line 4 of Algorithm (2), the reachable marking is computed. Note that the reachable markings are the same for the configuration of a cut-off event and the configuration of its corresponding event, thus the second computation is redundant. A concurrent transition in a behavior machine is computed in line 10. It is simply the set difference of configuration $[d]$ and the maximal configuration $[e]$ contained in $[d]$. This computation is correct since the configuration $[e]$ is a subset of $[d]$, and all events in $[e]$ occur in the behavior machine in one or more other concurrent transitions. The behavior machine is now nearly complete. The missing initial global state of the behavior machine is computed from the reachable marking of the empty configuration, i.e., it is the initial marking in the Petri net.

```
1    • let E be the set of local configurations;
2    global_states = ∅;
3    forall configurations [d] ∈ E do
4        • compute the reachable marking reachable_marks of [d];
5        global_states = global_states ∪ {reachable_marks};
6    end
7    conc_trans = ∅;
8    forall configurations [d] ∈ E do
9        • let [e] be the maximal configuration of [d];
10       conc_trans = conc_trans ∪ {[d] \ [e]};
11   end
12   return global_states, conc_trans;
```

ALGORITHM 2. Construction of global states and concurrent transitions in a BM.

The construction of a behavior machine is based on cut-off events and corresponding events in the finite complete prefix and the configurations belonging to them. If we assume that the events and conditions of the prefix are stored in doubly linked lists, local configurations and reachable markings can be computed in linear time. Thus, the highest complexity of the construction algorithm is contained in Algorithm (1), lines 11–16. The complexity of these few lines is bound on $O(n^2 \cdot (\log_k n)^2)$, where n is the number of places in the prefix of the unfolding, and k is the largest number of successor places of any transition. All other parts of the construction algorithm are less complex.

To demonstrate the feasibility of the construction algorithm, we compute the behavior machines for a variable number of processes of the Dining Philosophers example. The results are given in Table (1). Note that this classic example contains a deadlock state whose configuration leading to this state is not represented in the behavior machine due to Proposition (1). However, all other behavior parts are truly represented. The second column of the table shows the number of reachable states computed in a traditional reachability analysis. This number of states grows clearly exponentially with the number of philosophers. The size of the unfolding is given in the third column. The following two columns reveal the numbers of concurrent transitions and global states of the constructed behavior machines. We realize that the number of global states increases slightly worse than quadratic. Even though the computation time increases fast and seems to be a function of $n^{5.5}$, where n is the number of philosophers, the time is still reasonable small. This is also particularly true for the memory space used. The computation time of the finite complete prefix that is used as input for our construction algorithm was always less or around few seconds.

# philo.s	# reachable states	# conditions in prefix	# global states	# concurrent transitions	mem. usage (kByte)	computation time (sec)
5	392	135	16	35	36	0.11
7	4,247	273	36	77	61	0.73
9	46,763	459	64	135	98	2.96
11	510,116	693	100	209	151	9.12
13	5,564,522	975	144	299	222	22.76
15	—	1305	196	405	314	48.83

TABLE 1. Results of the Dining Philosophers example, computed on a SPARC Station 5.

6 Test generation based on behavior machines

The behavior machine is an appropriate model for test suite derivation. A test suite has to fulfill certain properties to be useful in software testing. Especially, it must distinguish faulty implementations from correct ones according to a chosen conformance relation. A conformance relation commonly used in testing is the *trace equivalence* between LTSs modelling the specification and the implementation. We extend trace equivalence over LTSs to an equivalence over behavior machines. First, equivalence of lposets is defined as follows borrowing ideas from [Bri88].

Definition (4): Lposet $lpo_1 = (E_1, A_1, \leq_1, l_1)$ *reduces* lposet $lpo_2 = (E_2, A_2, \leq_2, l_2)$, denoted $lpo_1 \sim lpo_2$, iff $A_1 \subseteq A_2$, and for all $e, f \in E_1$, if $e \leq_1 f$ then there exist $r, s \in E_2$ such that $r \leq_2 s$, and $l_2(r) = l_1(e)$, $l_2(s) = l_1(f)$. Two lposets are *equivalent*, $lpo_1 \approx lpo_2$, iff $lpo_1 \sim lpo_2$ and $lpo_2 \sim lpo_1$.

We define further a sequence *seq* of lposets as a concatenation of a matching sequence of lposets according to the $\oplus$ operator (see Section 3): $seq = lpo_1 \oplus lpo_2 \oplus lpo_3 \oplus \ldots$. Thus, a sequence of lposets describes a pomset, i.e. an execution branch of the concurrent system as depicted in its pomtree. Two sequences seq_1 and seq_2 equal, $seq_1 = seq_2$, if their lposets are equivalent.

Definition (5): Given two concurrent systems $\mathfrak{I}$ and $\mathfrak{R}$ and their behavior machines $BM_{\mathfrak{I}}$ and $BM_{\mathfrak{R}}$, respectively. Let $Seq(BM)$ refer to the set of sequences of lposets of which behavior machine BM is able to perform. $\mathfrak{I}$ and $\mathfrak{R}$ are *equivalent*, $\mathfrak{I} \approx_c \mathfrak{R}$, iff $Seq(BM_{\mathfrak{I}}) = Seq(BM_{\mathfrak{R}})$.

A test suite consisting of a finite set of finite test cases is *sound* w.r.t. a fault model if any conforming implementation passes the test suite. A test suite is *complete* w.r.t. a fault model if any non-conforming implementation from the implementation domain fails the test suite [PYB96]. A possible implementation domain is the set of implementations with acceptance faults, i.e., those implementations may not accept all actions by corresponding transitions as required in the specification, they reduce the specification [Lan90].

Acceptance faults can be detected by a *transition cover* through the concurrent system. A transition cover is usually defined over an LTS as a set of traces covering all transitions in the LTS. We extend now the notion of a *transition tour* [ADL+91] over a behavior machine as the least sequence of lposets covering all lposets. Such a sequence of lposets can be seen as a "transition cover" through a behavior machine. The extended notion is called *concurrent transition tour* (CTT) [UlCh95].

Definition (6): (CTT) A *concurrent transition tour* through a behavior machine $BM_{\Im}$ of a concurrent system $\Im$ is the least pomset $CTT = [E_{CTT}, A_{CTT}, \leq, l]$ such that all actions of $\Im$ are covered in the pomset, i.e. if $a \in A_1 \cup A_2 \cup \ldots \cup A_n$, then $a \in A_{CTT}$, and E_{CTT} is minimal.

Derivation of a CTT from a behavior machine is straightforward. Since the description of concurrent behavior is reduced to a finite directed graph, simple graph algorithms can be applied. To construct a CTT, an algorithm that solves the *Chinese postman problem* is appropriate [ADL+91]. First, all strongly-connected components of maximum size contained in a behavior machine are computed. After that, a CTT is derived for each strongly-connected component. This approach assures full coverage of all transitions in the behavior machine. The complete algorithm is given in Algorithm (3). Note however that the size of CTTs computed in this algorithm might not be minimal if we assume, for example, a behavior machine consisting of two components where the second component is reachable through the first one. Further optimization strategies might become applicable in this case.

1 Find all strongly-connected subgraphs of maximum size $bm^s{}_1, \ldots, bm^s{}_n$ in behavior machine $BM_{\Im}$.
2 For each $bm^s{}_i$ find the shortest path p_i from the initial state of bm to $bm^s{}_i$.
3 For each $bm^s{}_i$ find the Chinese postman tour pt_i.
4 A CTT of a subgraph of bm is found by concatenation of p_i and pt_i: $CTT_i = p_i \oplus pt_i$.
5 The test suite is the set of all CTTs found: $TS = \{CTT_1, \ldots, CTT_n\}$.

ALGORITHM 3. Test suite derivation.

Consider the behavior machine $BM_{\Im}$ of the system $\Im = A \parallel B$ in Figure 2. It contains one strongly-connected component consisting of the states S_1 and S_2. The initial path to reach this component is given by concurrent transition t_1. The Chinese postman tour through the component is the sequence of concurrent transitions $t_2 \oplus t_3 \oplus t_2 \oplus t_4$. The final test suite of system $\Im$ contains only a single CTT and is given in Figure 5 as a time-event sequence diagram where the gray-shaped arrows denote synchronization constraints between the modules A and B. This test suite describes the shortest path through the concurrent system fulfilling the requirements of a CTT.

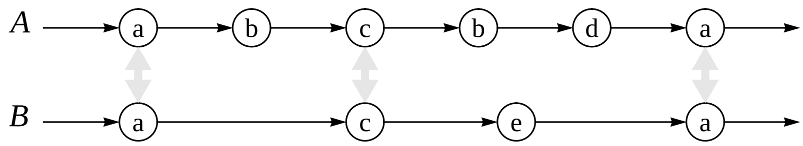

FIGURE 5. A concurrent transition tour for concurrent system $\Im = A \parallel B$.

7 AN EXAMPLE SYSTEM

We discuss our test derivation approach for concurrent systems with a larger example taken from [GaMo96]. It describes different LOTOS specifications of distributed leader election algorithms for unidirectional ring networks. These algorithms address the problem that n stations need to share a common resource. Each station s_i is given a unique address A_i. Two actions, “OPEN $!A_i$” and “CLOSE $!A_i$”, are used when s_i starts and stops to access the resource. A token is exchanged between the stations to control the access. A distributed leader election algorithm has to grant access to the resource for all stations fulfilling at least the two properties that (1) access is mutually exclusive and (2) each station has equal opportunities to access the resource. The simplest version of an algorithm assuming three stations is given in the Annex. Its LOTOS specification consists of three stations connected via three reliable links. Other, more elaborated approaches that even work in the case of unreliable links, are discussed in [GaMo96].

Ref	Composite machine		Behavior machine		Concurrent test suite	
#	# states	# transitions	# states	# transitions	# test cases	# test events
1	42	52	38	61	27	182
2	50	72	66	125	66	575
3	126,577	319,010	1159	2506	1354	11,329
4	43,296	115,108	1040	2269	962	9344
5	16,985	42,423	1432	2971	1378	13,355
6	6572	14,516	1303	2703	1044	11,671
7	30,085	77,680	2178	5134	1265	22,221
8	9,308	21,078	2049	4930	807	19,884
9	255,292	657,751	2871	7505	1352	30,876
10	31,914	74,202	2686	6993	920	26,338
11	—	—	2048	5428	1257	22,892
12	689,515	2,960,587	1808	4807	859	17,694

TABLE 2. Results of the computation of test suites for different leading election algorithms.

In order to operate with Full LOTOS applications, we use the *Cæsar* / *Aldebaran* verification toolset [GaSi90]. *Cæsar* produces a global Petri net of the specification as it is a required input for the construction of a behavior machine. The produced Petri net has also the advantage that values of variables are still represented symbolically. The Petri net of the specification serves then as input for the construction of the complete finite prefix of the net using the *PEP* tool [GrBe96]. Finally, the behavior machine is constructed from the prefix as outlined in Section 5. The corresponding tool searches for pairs of cut-off and corresponding events in the prefix, constructs the local configurations of the events and finally the behavior machine. Eventually, test suites according to the acceptance fault model are generated from each behavior machine. Table 2 depicts the results of the computations for different specifications from [GaMo96]. Results for the specification given in the Annex are contained in the first row (reference no. 1). The last two columns show the number of test cases, i.e. the number of different CTTs to cover the behavior machine, and the overall size of a test suite derived from the behavior machine. As a comparison the sizes of the interleaved-based composite machines are also given. These results were obtained from the *Aldebaran* tool.

The results in the table disclose that the advantage in a smaller size of a behavior machine over the composite machine depends on the considered specification. If the degree of concurrency is low, as it is indeed the case for specifications 1 and 2, the behavior machine reaches the size of the composite machine or is even larger if recursion points cannot be detected efficiently. However for the other concurrent systems, the computed behavior machines are much smaller than their interleaved-based models. In case of specification 11, the composite machine could not be computed at all due to memory shortage (results are from a SPARC 20 with 64 MB memory).

At this place it should be noted that the construction of a behavior machine from the prefix of a Petri net unfolding using the verification tool *PEP* does not always give satisfying results since *PEP* stops to unfold an execution branch of the Petri net if it has covered all reachable states (a sufficient condition in verification) instead of continuing unfolding till a cut-off event is reached. This results in behavior machines containing cycles that are not yet completed and explains the sometimes very high number of CTTs (test cases) required to obtain full coverage in a test suite. A new implementation of the testing approach should eliminate the disadvantage by including the unfolding algorithm and BM construction algorithm into a single tool.

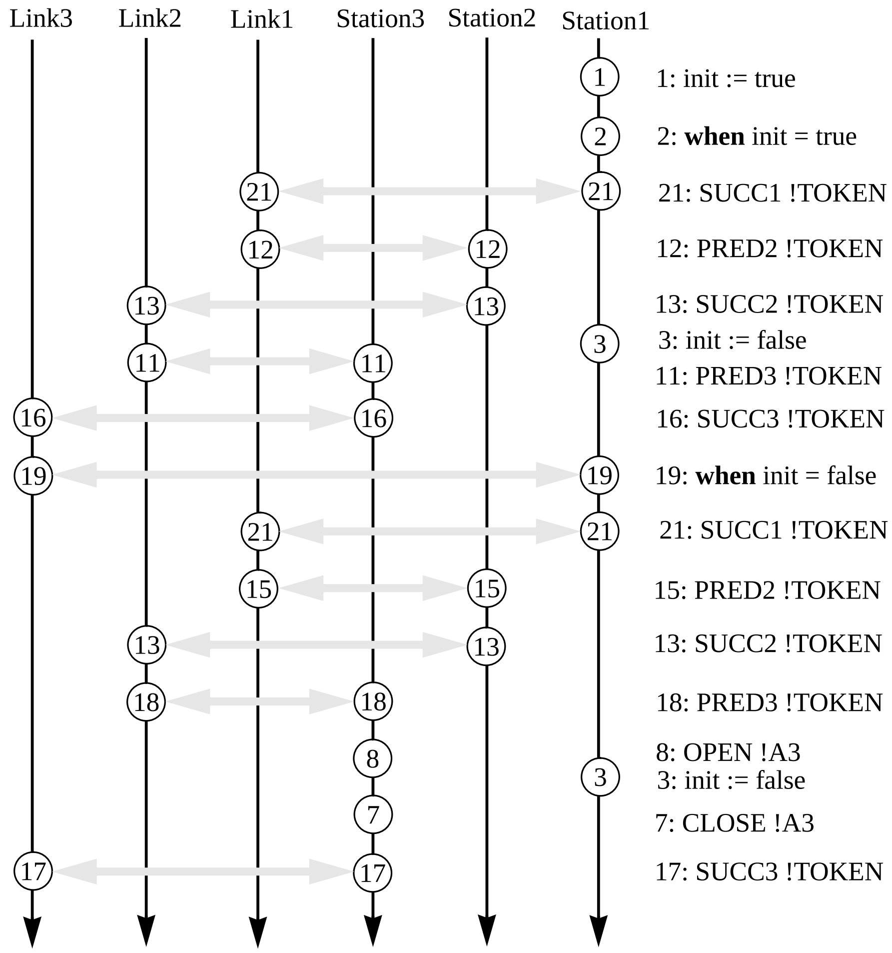

FIGURE 6. The initial part of a test case derived from specification #1.

Finally, Figure 6 gives an impression of the initial part of a concurrent transition tour as it was automatically generated using the tools introduced above. The CTT was obtained from the specification 1 described in the Annex. The representation of a CTT as time-event sequence diagram exhibits the possibility to express true concurrency, although the degree of concurrency is quite low in this example. In fact, since there is only one token in that specific example that is passed from one station to the next, the whole system behaves nearly sequential.

8 DISTRIBUTED TESTER DESIGN

Although the derivation of tests from concurrent systems appears to be manageable now by the approach presented before, there is still the problem to apply such a concurrent transition tour in a real test environment. Here we are faced with some unpleasant properties of concurrent systems, namely with the unpredictable progress of the independent modules, which the concurrent system contains of, when executing a test run and with hidden communication between modules that remain unobserved by the tester. Both issues result in nondeterministic behavior of the implementation under test (IUT). Whereas the second issue can be solved by applying a gray-box testing approach, the first one requires special means to ensure a deterministic test run, e.g. the instant replay technique presented in [TCO91].

Different test architectures can be designed to test a concurrent system. In the simplest case, the tester is represented as a single module that implements a CTT of a given test suite. Since a tester module can exhibit only sequential behavior, the concurrent behavior of a CTT must be linearized in such a way that the causal relationships between events remain unchanged. For example, if the CTT in Figure 5 shall be implemented, the sequence of events "*a.b.c.b.d.e.a*" fulfills the requirements. If the IUT is correctly implemented according to the chosen fault model, then it must be able to accept any sequence of the CTT and the selection of the actual sequence used in testing is arbitrary.

In case that the tester consists itself of distributed concurrent tester parts, further efforts are required to obtain reliable test results after a test run. Here, each tester part observes a sub-set of events from the IUT. The partial behavior containing those events visible to a tester part can be obtained from a projection from the complete CTT to a partial CTT containing only the visible events. Each tester part will then execute the behavior of the projected CTT. Yet, means are required to obtain a globally ordered test run of the IUT from the observed partial behavior. The paper [UlCh95] suggests the use of logical clocks in this case. Although logical clocks are the most elegant solution for this issue, their application requires changes in the IUT in order to provide the exchange of logical clock values between modules of the IUT and between IUT and tester. Another solution to obtain a globally ordered test run is the inclusion of synchronization messages between tester parts: a tester part synchronizes with other tester parts after an interaction with the IUT. A starting point for further research in the area of distributed test architecture design can be the paper [ChVD90] that introduces the ferry-clip approach, a distributed test architecture to support multi-party testing.

9 CONCLUSIONS

The model of a behavior machine is used to support test derivation for concurrent systems. The model has its merits as a finite description of concurrent systems that still exhibits true concurrency among actions. Furthermore, a behavior machine distinguishes concurrency from execution branching. Vertices in the behavior machine refer usually to a small subset of the set of reachable global states in the system. The main contributions in the paper are the presentation of an algorithm that constructs a behavior machine from a Petri net unfolding as well as the application of the behavior machine to the realm of test derivation. For the purpose of test derivation, the notion of trace equivalence was extended to cope with concurrent behavior. Using the equivalence over behavior machines as conformance relation, a test derivation algorithm based was introduced that generates test suites according to the acceptance fault model.

The presented approach to construct a behavior machine can be improved. Especially, the Petri net unfolding algorithm and the construction algorithm of the behavior machine should be combined into a single tool. Furthermore, the construction algorithm of a BM should support any kind of concurrent systems, not just deadlock-free ones, to be applicable in general. Other extensions of a behavior machine may include the support of inputs and outputs and the supply of operations over behavior machines, e.g. the composition of two behavior machines, projections of behavior machines to submachines and other operations.

Test derivation from behavior machines produces currently test suites based on an extension of the traditional transition tour through an LTS according to acceptance faults. Further, more expanded fault models should be investigated and optimized test derivation algorithms should be elaborated for them. There is also a more vigorous attempt needed to prove that the use of a behavior machine of a concurrent system instead of its composite machine in test derivation results in test suites of a comparable fault coverage. Last but not least, the employment of test suites in a distributed test architecture is another big research area where a lot of work still needs to be done.

ACKNOWLEDGMENT

The authors wish to thank Alex Petrenko for a very fruitful discussion and for his hints that helped to improve the quality of the paper.

10 REFERENCES

[ADL+91] A. V. Aho, A. T. Dahbura, D. Lee, M. Ü. Uyar: *An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours*; IEEE Transactions on Communications, vol. 39, no. 11 (Nov. 1991); pp. 1604–1615.

[Bri88] Ed Brinksma: *A theory for the derivation of tests*; 8th International Symposium on Protocol Specification, Testing and Verification (PSTV'88), Atlantic City, USA; 1988.

[ChVD90] S. T. Chanson, S. T. Vuong, H. Dany: *The ferry clip approach to multi-party testing*; 3rd International Workshop on Protocol Test Systems (IWPTS'90); McLean, USA; 1990; pp. 335–353.

[ERV96] J. Esparza, S. Römer, W. Vogler: *An improvement of McMillan's unfolding algorithm*; 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96); Passau, Germany; 1996.

[Fer96] J.-C. Fernandez, C. Jard, Th. Jéron, César Viho: *Using on-the-fly verification techniques for the generation of test suites*; 8th International Conference on Computer Aided Verification (CAV'96); New Brunswick, New Jersy, USA; 1996.

[GaSi90] H. Garavel, J. Sifakis: *Compilation and verification of Lotos specifications*; 10th International Symposium on Protocol Specification, Testing and Verification (PSTV'90); Ottawa, Canada; 1990; pp. 379–394.

[GaMo96] H. Garavel, L. Mounier: *Specification and verification of various distributed leader election algorithms for unidirectional ring networks*; Technical Report No. 2986, Unité de recherche INRIA Rhône-Alpes; Sep. 1996.

[GrBe96] B. Grahlmann, E. Best: *PEP – More than a Petri net tool*; 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96); Passau, Germany; 1996.

[KCK+96] M. C. Kim, S. T. Chanson, S. W. Kang, J. W. Shin: *An approach for testing asynchronous communicating systems*; 9th International Workshop on Testing of Communicating Systems (IWTCS'96); Darmstadt, Germany; Sep. 1996.

[Lan90] R. Langerak: *A testing theory for LOTOS using deadlock detection*; 9th International Symposium on Protocol Specification, Testing and Verification (PSTV'90); Enschede, The Netherlands; 1990.

[LSK+93] D. Lee, K. K. Sabnani, D. M. Kristol, S. Paul: *Conformance testing of protocols specified as communicating FSMs*; IEEE INFOCOM'93; San Fransisco, CA, USA; 1993.

[McM95] K. L. McMillan: *A technique of state space search based on unfolding*; Formal Methods in System Design, vol. 6, no. 1 (Jan. 1995); pp. 45–65.

[Pra86] V. Pratt: *Modelling Concurrency with partial orders*; International Journal of Parallel Programming, vol. 15, no. 1 (Feb. 1986); pp. 33–71.

[PLL+91] D. K. Probst, H. F. Li, K. G. Larsen, A. Skou: *Partial-order model checking: a guide for the perplexed*; 3nd International Conference on Computer-aided Verification (CAV'91); Aalborg, Denmark; 1991.

[PBY96] A. Petrenko, G. v. Bochmann, M. Yao: *On fault coverage of tests for finite state specifications*; Special Issue on Protocol Testing, Computer Networks and

ISDN Systems, vol. 29, 1996; pp. 81–106.

[Rei91] W. Reisig: *Petri nets*; Springer Verlag, 1991.

[TCO91] K. C. Tai, R. H. Carver, E. E. Obaid: *Debugging concurrent Ada programs by deterministic execution*; IEEE Transactions on Software Engineering, vol. 17, no. 1 (Jan. 1991); pp. 45–63.

[UlCh95] A. Ulrich, S. T. Chanson: *An approach to testing distributed software systems;* 15th International Symposium on Protocol Specification, Testing and Verification (PSTV'95); Warsaw, Poland; pp. 107–122; 1995.

[Ulr97] A. Ulrich: *A description model to support test suite derivation for concurrent systems*; in M. Zitterbart (ed.): Kommunikation in Verteilten Systemen, GI/ITG-Fachtagung (KiVS'97); Braunschweig, Germany; Springer Verlag, Reihe Informatik aktuell, 1997; pp. 151-166.

ANNEX

The simplest version of the distributed leader election algorithm from [GaMo96] assuming three stations connected via reliable unidirectional links.

```
specification EXPERIMENT_01 [OPEN, CLOSE] : noexit
library BOOLEAN endlib
type FRAME is
   sorts FRM
   opns TOKEN (*! constructor *) : -> FRM
endtype
type ADDRESS is
   sorts ADDR
   opns A1 (*! constructor *) : -> ADDR
        A2 (*! constructor *) : -> ADDR
        A3 (*! constructor *) : -> ADDR
endtype
behaviour
   hide PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3 in
   (  (
      STATION [OPEN, CLOSE, PRED1, SUCC1] (A1, true)
      |||
      STATION [OPEN, CLOSE, PRED2, SUCC2] (A2, false)
      |||
      STATION [OPEN, CLOSE, PRED3, SUCC3] (A3, false)
      )
   |[PRED1, SUCC1, PRED2, SUCC2, PRED3, SUCC3]|
      (
      LINK [SUCC1, PRED2]
      |||
      LINK [SUCC2, PRED3]
      |||
      LINK [SUCC3, PRED1]
   )  )
where
process LINK [INPUT, OUTPUT] : noexit :=
   INPUT !TOKEN;
      OUTPUT !TOKEN;
         LINK [INPUT, OUTPUT]
endproc
process STATION [OPEN, CLOSE, PRED, SUCC] (Ai:ADDR, INIT:BOOL) : noexit :=
   [INIT = true] ->
```

```
      PRIVILEGE [OPEN, CLOSE, PRED, SUCC] (Ai)
   []
   [INIT= false] ->
      PRED !TOKEN;
         PRIVILEGE [OPEN, CLOSE, PRED, SUCC] (Ai)
endproc
process PRIVILEGE [OPEN, CLOSE, PRED, SUCC] (Ai:ADDR) : noexit :=
   SUCC !TOKEN;
      STATION [OPEN, CLOSE, PRED, SUCC] (Ai, false)
   []
   OPEN !Ai;
      CLOSE !Ai;
         SUCC !TOKEN;
            STATION [OPEN, CLOSE, PRED, SUCC] (Ai, false)
endproc
endspec
```