UC Irvine
ICS Technical Reports

Title
Graph models for reachability analysis of concurrent programs

Permalink
https://escholarship.org/uc/item/9qm0m9q5

Authors
Pezze, Mauro
Taylor, Richard N.
Young, Michal

Publication Date
1992-01-07

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

ARCHIVE
Z
699
C3
no. 92-27
c.2

Graph Models for Reachability Analysis of Concurrent Programs

Mauro Pezze
Richard N. Taylor
Michal Young

Technical Report 92–27

January 7, 1992

Graph Models for Reachability Analysis of Concurrent Programs

Mauro Pezzè [1]
Richard N. Taylor

Department of Information and Computer Science
University of California,
Irvine, CA 92717

Michal Young

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 49707

January 7, 1992

[1]This material is based upon work supported by the National Science Foundation under Award No. CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, by the National Science Foundation under Award No.s CCR-8451421, and TRW (PYI program).

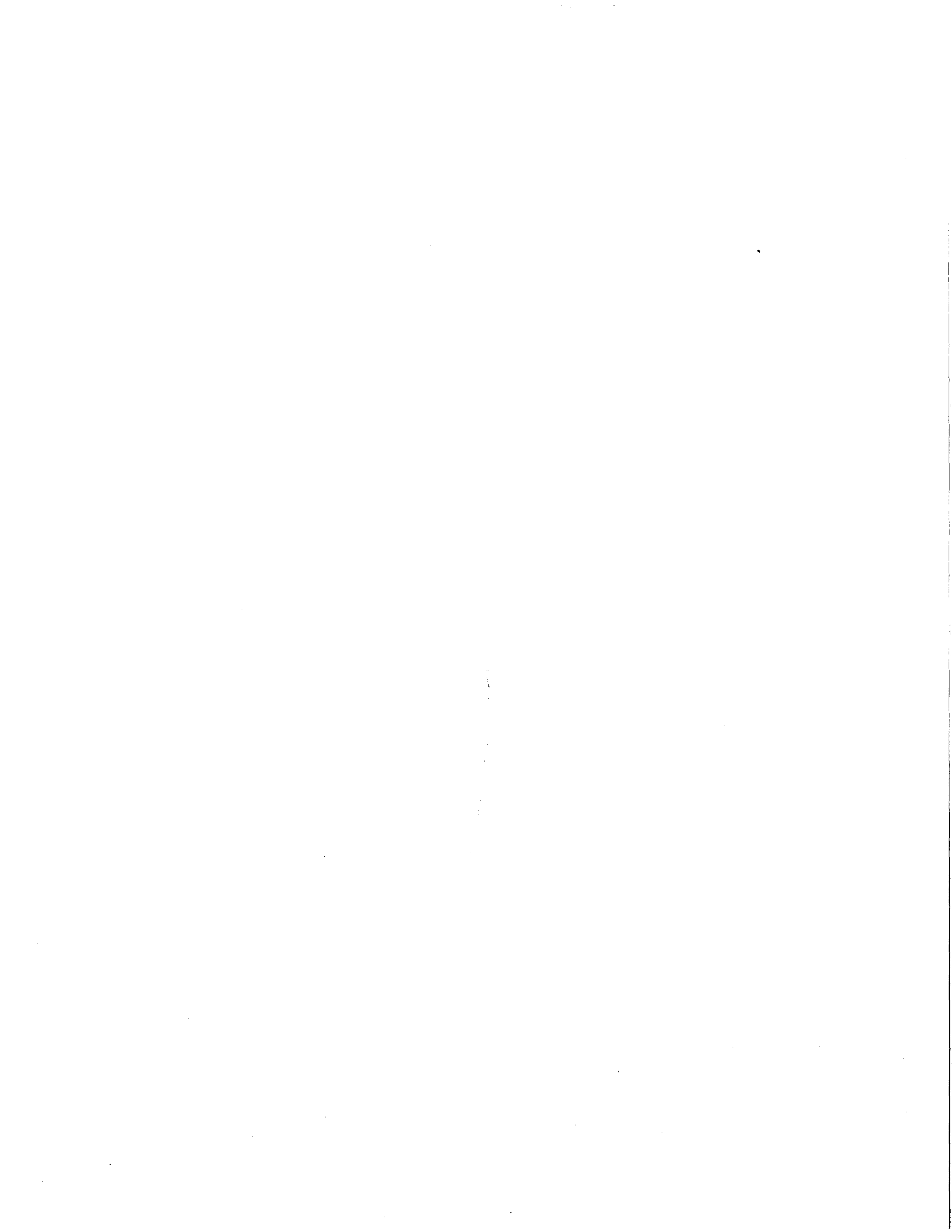
Abstract

Reachability analysis is an attractive technique for analysis of concurrent programs because it is simple and relatively straightforward to automate, and can be used in conjunction with model-checking procedures to check for application-specific as well as general properties. Several techniques have been proposed differing mainly on the model used; some of these propose the use of flowgraph based models, some others of Petri nets.

This paper addresses the question: What essential difference does it make, if any, what sort of finite-state model we extract from program texts for purposes of reachability analysis? How do they differ in expressive power, decision power, or accuracy? Since each is intended to model synchronization structure while abstracting away other features, one would expect them to be roughly equivalent.

We confirm that there is no essential semantic difference between the most well known models proposed in the literature by providing algorithms for translation among these models. This implies that the choice of model rests on other factors, including convenience and efficiency.

Since combinatorial explosion is the primary impediment to application of reachability analysis, a particular concern in choosing a model is facilitating divide-and-conquer analysis of large programs. Recently, much interest in finite-state verification systems has centered on algebraic theories of concurrency. Yeh and Young have exploited algebraic structure to decompose reachability analysis based on a flowgraph model. The semantic equivalence of graph and Petri net based models suggests that one ought to be able to apply a similar strategy for decomposing Petri nets. We show this is indeed possible through application of category theory.



Contents

1 Introduction 1

2 Background 2
2.1 Concurrency Graphs 2
2.2 Task Interaction Concurrency Graphs 9
2.3 Petri Nets 11

3 A Unifying Model for Concurrency Graphs and TICGs 17
3.1 Labeled Flow Graphs and Concurrency Flow Graphs 17
3.2 Concurrency Graphs as Concurrency Flow Graphs 18
3.3 TIGs as Flow Graphs 22

4 Concurrency Flow Graphs and Petri nets 26

5 Compositionality and Process Algebras 31

6 Compositionality and Petri Nets 38
6.1 The category safe net 38
6.2 The bounded buffer example 43

7 Conclusions 43

References 47

1 Introduction

The problem of analyzing concurrent systems has been investigated by many researchers, and several solutions have been proposed. The various solutions differs in the way concurrent systems are represented (flowgraphs [Tay83b, LC89, BDER79], Petri nets [GMMP89, SC88, MR87, SMS86], process algebras [Mil80, BHR84], temporal logic [CES86, Wol86] programming languages [Han73, Ger84, TKO91, Tai85]), and in the kind of analysis performed (static analysis [BDER79, Ger84, Tay83b, LC89, TO80], dynamic analysis [Han73, CDK85, Tai85], symbolic execution [YT88, GMMP89, Dila, Dilb, HK88], formal proof of properties [Mil80, Apt83, BR89, CES86, Wol86]).

Among the proposed techniques, reachability analysis — systematic enumeration of reachable states in a finite-state model — is attractive because it is simple and relatively straightforward to automate, and can be used in conjunction with model-checking procedures (e.g., [CES86]) to check for application-specific as well as general properties. Reachability analysis has been used successfully in limited domains like simple communication protocols [Sun81, Hol87]. Combinatorial explosion has stymied application of reachability analysis to general concurrent programs.

One thread of research related to reachability analysis involves extracting models from program texts. Since Taylor proposed a reduced flowgraph model for reachability analysis of programs expressed in Ada and related languages [Tay83b], a variety of alternative representations have been proposed. Long and Clarke have proposed a "task interaction graph" representation that captures synchronization structure more succinctly than Taylor's original model [LC89] (an important attribute since combinatorial explosion is the primary limiting factor in applying reachability analysis). Shatz et al. have proposed extracting Petri nets from Ada programs in order to use existing Petri net analysis tools [SC88] and net reduction techniques [SMBT90].

This paper addresses the question: What essential difference does it make, if any, what sort of finite-state model we extract from program texts for purposes of reachability analysis? How do they differ in expressive power, decision power, or accuracy? Since each is intended to model synchronization structure while abstracting away other features, one would expect them to be roughly equivalent. We confirm that there is no essential semantics difference between Taylor's original model, Long and Clarke's TIG model, and Petri nets by providing algorithms for translation among these three models. This implies that the choice of model rests on other factors, including convenience and efficiency of available tools.

Since combinatorial explosion is the primary impediment to application of reachability analysis, a particular concern in choosing a model is facilitating divide-and-conquer analysis of large programs. Recently Yeh and Young have exploited algebraic structure based on algebraic process theory [Hen88, Mil89, Hoa85] to decompose reachability analysis based on a graph model close to Long and Clarke's TIG model [YY91]. The semantic equivalence of TIGs and Petri nets suggests that one ought to be able to apply a similar strategy for decomposing Petri net models

of Ada programs. We show this is indeed possible through application of category theory [MM90, Win87, Win84].

Section 2 reviews representative graph based models that have been used for reachability analysis of programs in Ada-like languages: task flowgraphs[1] and concurrency graphs, as they are introduced in [Tay83b], task interaction graphs (TIGs) and task interaction concurrency graphs (TICGs) as introduced in [LC89] (both of which concern flowgraph based models), and Petri nets and reachability graphs as introduced in [Rei85].

The substantial equivalence among the flowgraph based approaches is shown in Section 3. The overall equivalence between flowgraph based and Petri net based approaches is discussed in Section 4.

The introduction of compositionality in flowgraph based approaches by using algebraic theory is discussed in Section 5, where the work presented in [YY91] is reviewed; the use of category theory for defining compositional rules for Petri nets is shown in Section 6. Section 7 concludes.

2 Background

Reachability analysis has been broadly used for the analysis of concurrent systems [Apt83, BDER79, CES86, LC89, McD89, Smo84, Tay83b, MR87, SMS86, SC88]. Two broad classes of models have been used for supporting reachability analysis of programs: flowgraph based models [Apt83, BDER79, CES86, LC89, McD89, Smo84, Tay83b]. and Petri nets [MR87, SMS86, SC88].

The use of flowgraph based models for analyzing Ada programs was first proposed in [Tay83b], and an improved model was described in [LC89]. An algorithm for reachability analysis of Petri nets has been first proposed in [ME69]. Reachability analysis of Petri nets has been used for analyzing concurrent programs by several authors ([MR87, SMBT90, SC88]).

2.1 Concurrency Graphs

Task flowgraphs and task interaction graphs (TIGs) have been defined as models of the Ada task system. Ada has been chosen for the increasing interests of the scientific and industrial community, but similar techniques can be defined for other concurrent languages with rendezvous synchronization. In this paper, Ada represents only a common concrete point of reference. Both task flowgraphs and TIGs consider only a subset of Ada. Some of the restrictions introduced on Ada are intrinsic to static analysis algorithms, while some others are introduced in this paper in order to abstract away from details which would only complicate our presentation.

If program objects can be indexed by variable expressions or referenced through a chain of pointers, a static tool is usually incapable of determining the particular

[1] In [Tay83b] task flowgraphs are called simply flowgraphs; the name *task flowgraphs* has been introduced here to avoid confusion with definitions that appear later in the paper and will be used consistently through the paper to refer to flowgraphs as defined in [Tay83b].

identity of an object; thus static analysis algorithms cannot deal satisfactorily with arrays of records that include tasks as members, or tasks that are objects of access types. Dealing with real-time operations also causes problems, since the real-time behavior of an Ada program depends on the performance characteristics of the target machines, usually not available to a static analysis algorithm; thus static analysis algorithms cannot fully model real-time operators like delay statements, timed entry calls, delay alternatives in selective waits, etc. Finally, static analysis requires some restrictions on dynamic task creation: since static analysis consists in building a finite representation of the state space of the analyzed program, the number of tasks composing the program must have an upper bound.

The restrictions introduced in this paper only for simplifying the presentation concern shared variables, task activation and termination, and subprograms. None of these features are considered in this paper, although dealing with them does not increase the complexity of the problem.

The reminder of this section simply recalls the definitions of task flowgraphs, and concurrency graphs as presented in [Tay83b] and the definitions of TIGs and TICGs as presented in [LC89], in order to make the paper self-contained. Most of the definitions reported in this section are informal and/or refer to the semantics of Ada (see Definition 4: concurrency graph). The formal definition of these concepts is one of the contribution of this paper and is given in Section 3, where differences and analogies among such models are discussed.

A task flowgraph represents the structure of a single Ada task abstracting away from all the aspects which are unnecessary in determining possible sequences of synchronization activities. Synchronization activities are statements involved in synchronization, like *accept*, *entry call*, *select*. A task flowgraph can be easily obtained from the annotated flowgraph produced by a compilation system by ignoring all the nodes representing states not involved in synchronization activities.

Definition 1 (synchronization activity) *Given an Ada program, a synchronization activity is one of the following statements: entry call, accept, select, select-else, task-begin, task-end.*

Definition 2 (flowgraph) *a 4-tuple $G = (N, A, s, T)$ is a flow graph if and only if*

1. *N is a finite set of elements called nodes,*
2. *$A \subset N \times N$ is a binary relation on N; elements of A are called arcs*
3. *$s \in N$; s is called the starting node*
4. *$T \subset N$; T is called the set of terminal nodes.*

Definition 3 (task flowgraph) *Given an Ada task A, a task flowgraph is a flowgraph, where each node corresponds to either a synchronization activity in A, or a*

```
task body T1 is
     done: boolean;
begin
     done := true;
     loop
         select
               accept(P)
         or
               accept (Q)
         end select;
         exit when done
     end loop
end T1

task body T0 is
begin
     T1.Q
end T0

task body T2 is
begin
     T1.P
end T2
```

Figure 1: A simple Ada program.

compound statement [2]*, and each arc corresponds to the flow of control among the considered statements in task A. The starting node is the node corresponding to the task-begin; the final nodes are the nodes corresponding to possible terminations of task A.*

Nodes of task flowgraphs will be also called *state-nodes*; a state-node m is called a *successor* of a state-node n, $m \in succ(n)$, if there is an arc from n to m.

As an example, the task flowgraphs corresponding to the Ada tasks of Figure 1 are presented in Figure 2. Notice that the loop of task T_1 is explicitly represented, since it contains a synchronization statement (a selective wait).

An Ada program can be identified with the set of task flowgraphs corresponding to the tasks in the program. The execution space of an Ada program can be described by a *concurrency graph*, a flowgraph where nodes, also called *concurrency-*

[2] compound statements, like *conditional statements* and *loops*, are explicitly modeled in the task flowgraph only if their body contains statements that correspond to synchronization activities.

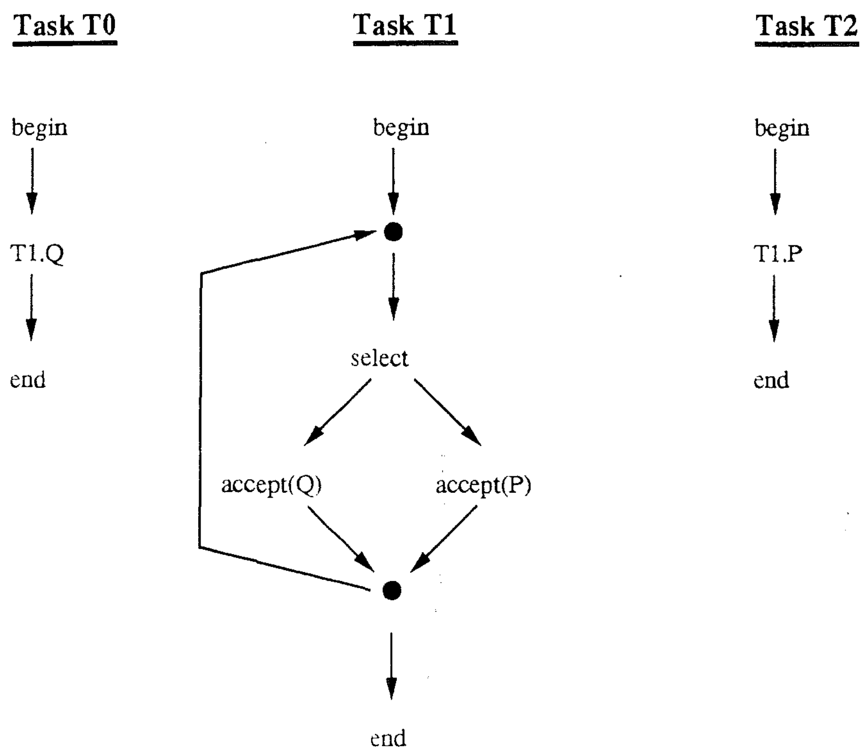


Figure 2: Task flowgraphs corresponding to the Ada program of Figure 1.

nodes, are k-tuples of state-nodes. one node for each task flow graph, and arcs represents actions that can cause a state transition in the program.

Definition 4 (concurrency graph) *Let P be a set of k task flowgraphs; a concurrency-node is a k-tuple of state-nodes; one node for each task flow graph in P; the concurrency-node $M =< m_1, ..., m_k >$ is a successor of the concurrency-node $N =< n_1, ..., n_k >$ if and only if:*

1. *$\forall i, 1 \leq i \leq k$ either*

   (a) $m_i \in succ(n_i)$

   (b) $m_i = n_i$

2. *there exists at least one $m_j, 1 \leq j \leq k$, which represents application of case 1a.*

3. *adherence to the semantics of Ada is reflected in the selection of the successors of concurrency-nodes,[3] and*

4. *the changes between the concurrency-state M and the concurrency-state N are the minimum number required to satisfy conditions (2) and (3), i.e., if the concurrency-state M is a successor of the concurrency-state N, then there does not exists a successor S of N such that the set of components for which S differs form N is a proper subset of the set of components for which M differs from N.*

A concurrency graph is the transitive closure of the successor relation.

Condition 2 prohibits self-loops in the concurrency graph, since it requires that two concurrency-nodes in the successor relation differ for at least one component. Condition 3 requires that the flow of control on the task flowgraphs considered in building the concurrency graph follows the Ada semantics. Condition 4 limits the set of successors to the set of nodes reachable with at most one action, be it concurrent or sequential: concurrency states reachable from the concurrency node N with more than one action performed concurrently, do not belong to the set of immediate successor of N, but they can be reached from N through intermediate nodes. (Such models are commonly called *interleaving,* as versus *true concurrency* and *partial order* models.)

As an example, the concurrency graph corresponding to the Ada program of Figure 1 is reported in Figure 3. The correspondence between the enumeration of the concurrency-nodes in Figure 3 and the state-node of Figure 2 is indicated in Table 1.

[3]This definition is taken from [Tay83b]; a formal definition that does not refer to the Ada semantics is presented in Section 3.

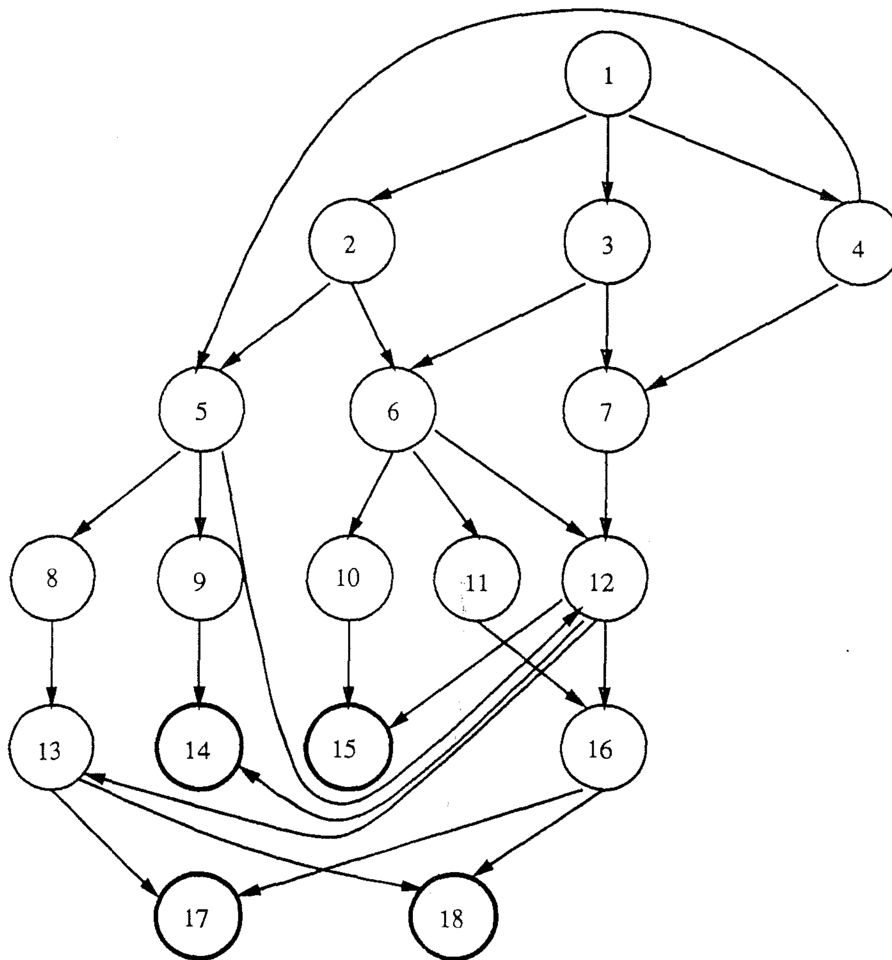


Figure 3: The concurrency graph corresponding to the Ada program of Figure 1.

concurrency node	state-node of task T_0	state-node of task T_1	state-node of task T_2
1	begin	begin	begin
2	begin	select	begin
3	T1.Q	begin	begin
4	begin	begin	T1.P
5	begin	select	T1.P
6	T1.Q	select	begin
7	T1.Q	begin	T1.P
8	begin	select	end
9	begin	end	end
10	end	end	begin
11	end	select	begin
12	T1.Q	select	T1.P
13	T1.Q	select	end
14	T1.Q	end	end
15	end	end	T1.P
16	end	select	T1.P
17	end	select	end
18	end	end	end

Table 1: Correspondence between the concurrency-nodes of the concurrency graph of Figure 3 (first column) and the state-nodes of the task flowgraphs of Figure 2 (columns 2, 3, 4).

2.2 Task Interaction Concurrency Graphs

Task interaction graphs (TIGs) and task interaction concurrency graphs (TICGs) have been introduced for reducing the size of the task flowgraphs and concurrency graphs. TIGs are defined starting from a partition of the statements of a single task in regions called *task regions.*

Definition 5 (task region) *Given an Ada task A, a task region is a portion of code starting from the task-begin, or from a statement that can immediately follow the execution of an accept or an entry call, and ends with the termination of the task execution or with accept or entry calls.*

The task regions for the Ada program of Figure 1 are shown in Figure 4.

Tasks T_0 and T_2 are divided in two regions, the first one ($C(01)$ and $C(21)$ respectively) goes from the beginning of the program to the entry call, the second one ($C(02)$ and $C(22)$ respectively) goes from the end of the entry call to the end of the program. Task T_1 is divided in three regions; the first one ($C(11)$) goes from the first statement of the program to either one of the two accepts; thus, it has two exit points; the second one ($C(12)$) goes from the end of the accept of entry Q to either the end of the program or one of the two accepts, thus it has three exit points; the third one ($C(13)$) goes from the end of the accept of entry P to either the end of the program or one of the two accepts, thus it has three exit points as well. Task regions end and start with beginning or ending of synchronization activities (entry call, accept, select, select-else, task-begin, task-end). Task regions can be linked together by a relation that represents the synchronization activities of the program; e.g., task regions C01 and C02 are related by the entry call T1.P; task regions C11 and C12 are related by the accept of entry Q.

Definition 6 (task interaction graph (TIG)) *Given an Ada task A, a TIG is a flow graph, whose nodes correspond to the task regions in A, and whose arcs represent task interactions, i.e., flow of control between task regions; arcs are labeled with the type of represented interaction.*

The nodes of TIGs will be also called *TIG-nodes.* The TIG-node corresponding to the initial region of the task is called the *initial TIG-node* of the TIG; the TIG-nodes corresponding to regions of the task where the execution may end, are called *final TIG-node* of the TIG. The initial TIG-node cannot be a final TIG-node.

The TIGs corresponding to the Ada program of Figure 1 are shown in Figure 5. For each node there are as many exiting arcs as there are exiting points corresponding to possible task interactions.

An Ada program can be represented by means of the set of TIGs corresponding to the tasks of the program. The execution space of an Ada program represented as a set of TIGs can be represented by means of a TICG, a flowgraph whose nodes are tuples of TIG-nodes, one for each TIG in the program, and arcs corresponds to the possible interactions between tasks in the program.

task T0	
C(01)	ENTER(TASK ACTIVATION) T1.Q EXIT(CALL START T1.Q)
C(02)	ENTER(CALL END T1.Q) end T0 EXIT(TASK TERMINATION)
task T1	

C(11)	ENTER(TASK ACTIVATION)
	
	loop
	    select
	        accept(P)
	EXIT(ACCEPT START P)
	    or
	        accept (Q)
	EXIT(ACCEPT START Q)

C(12)	ENTER(ACCEPT END Q) end select; exit when done EXIT(TASK TERMINATION) end loop EXIT(ACCEPT START P) or EXIT(ACCEPT START Q)

C(13)	ENTER(ACCEPT END P)
	    or
	        accept (Q)
	end select;
	exit when done
	EXIT(TASK TERMINATION)
	end loop
	EXIT(ACCEPT START P)
	    or
	EXIT(ACCEPT START Q)

task T2	
C(21)	ENTER(TASK ACTIVATION) T1.P EXIT(CALL START T1.P)
C(22)	ENTER(CALL END T1.P) end T2 EXIT(TASK TERMINATION)

Figure 4: Task regions for the Ada program of Figure 1.

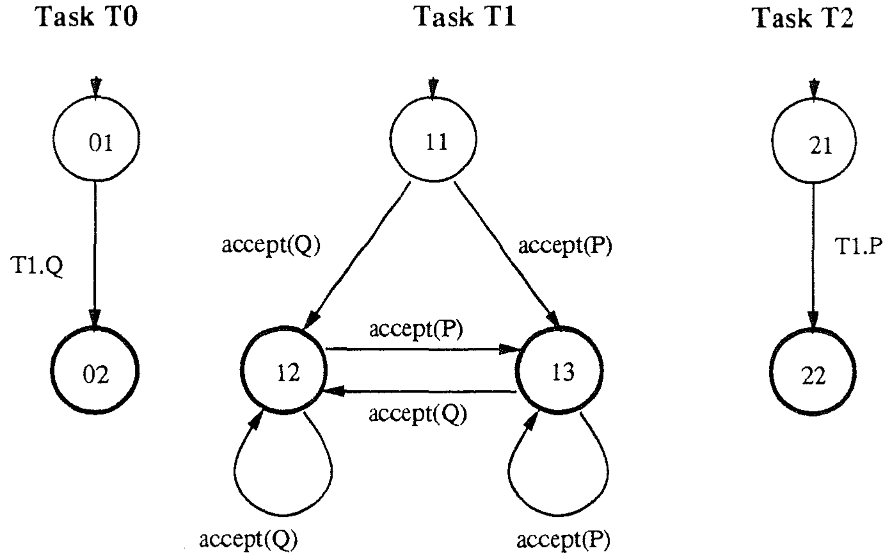


Figure 5: Task interaction graphs (TIGs) for the Ada program of Figure 1.

Definition 7 (task interaction concurrency graph (TICG)) *Given a set of k TIGs; a TICG-node is a k-tuple of TIG-nodes, one for each TIG; arcs are defined by the following successor relation: a node* $M = < m_1, ..., m_k >$ *is a successor of a node* $N = < n_1, ..., n_k >$ *if and only if there exists* i *and* j *such that for all* $l \neq i, j, m_l = n_l$ *and*

1. $< n_i, m_i >$ *is an arc in* TIG_i
2. $< n_j, m_j >$ *is an arc in* TIG_j
3. *the labels associated with arcs* $(< n_i, m_i >)$ *and* $(< n_j, m_j >)$ *match, i.e., the arc* $(< n_i, m_i >)$ *belongs to task* T_a *and is labeled with accept(P) and the arc* $(< n_j, m_j >)$ *belongs to task* T_b *and is labeled with* $T_a.P$.

A TICGs *is the transitive closure of the successor relation.*

The TICG corresponding to the Ada program of Figure 5 is shown in Figure 6.

2.3 Petri Nets

Petri nets have been originally proposed as a model for representing and analyzing concurrent systems; however they have been used also for representing and analyzing Ada programs [MZGT85, MP89, SMBT90].

Definition 8 (Petri net) *A Petri net is a 4-tuple* (P, T, F, M_0) *where,*

1. *P is a non-null set of elements, called places*

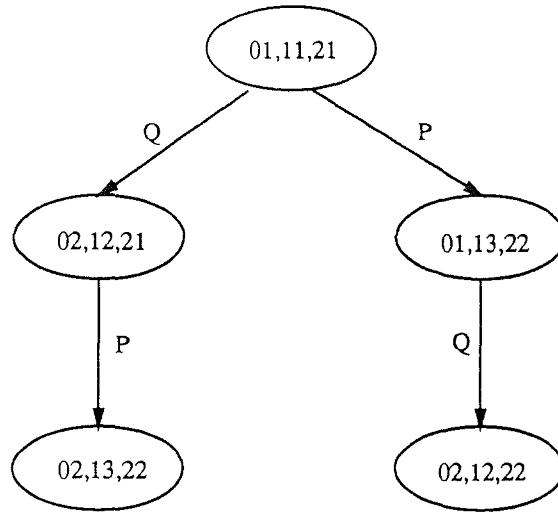


Figure 6: The task interaction concurrency graph (TICG) corresponding to the Ada program of Figure 1.

2. *T is a non-null set of elements, called transitions*

3. *$F \subseteq (P \times T) \cup (T \times P)$ is a flow relationship between places and transitions; an element of F is called arc.*

4. *$M_0 : P \rightarrow N$ is an assignment of natural numbers to places, called the initial marking. The number associated with a place p by the marking m_0 is referred to as the number of tokens in place p*

Definition 9 (preset) *The set of places for which there exists an arc leading to a transition t is called the preset of transition t, and is indicated by $^\bullet t$.*
$^\bullet t = \{x \in P \mid < x, t > \in F\}$.

Definition 10 (postset) *the set of places to which there exists a pointing arc from transition t is called the postset of transition t, and is indicated by $t^\bullet$.*
$t^\bullet = \{x \in P \mid < t, x > \in F\}$.

As an example, a possible representation of the Ada program of Figure 1 is presented in Figure 7. In Figure 7, transitions represent actions, places represent conditions and arcs give the relations between conditions and actions. Transitions are labeled for convenience of the reader. Different algorithms for obtaining a Petri net from Ada programs can be defined, resulting in different representations. The one chosen in the example of Figure 7 represents an Ada program by modeling only synchronization activities.

The dynamic behavior of a system modeled by means of a Petri net can be represented by the *firing rule*, which describes the effect of actions on the marking of the net.

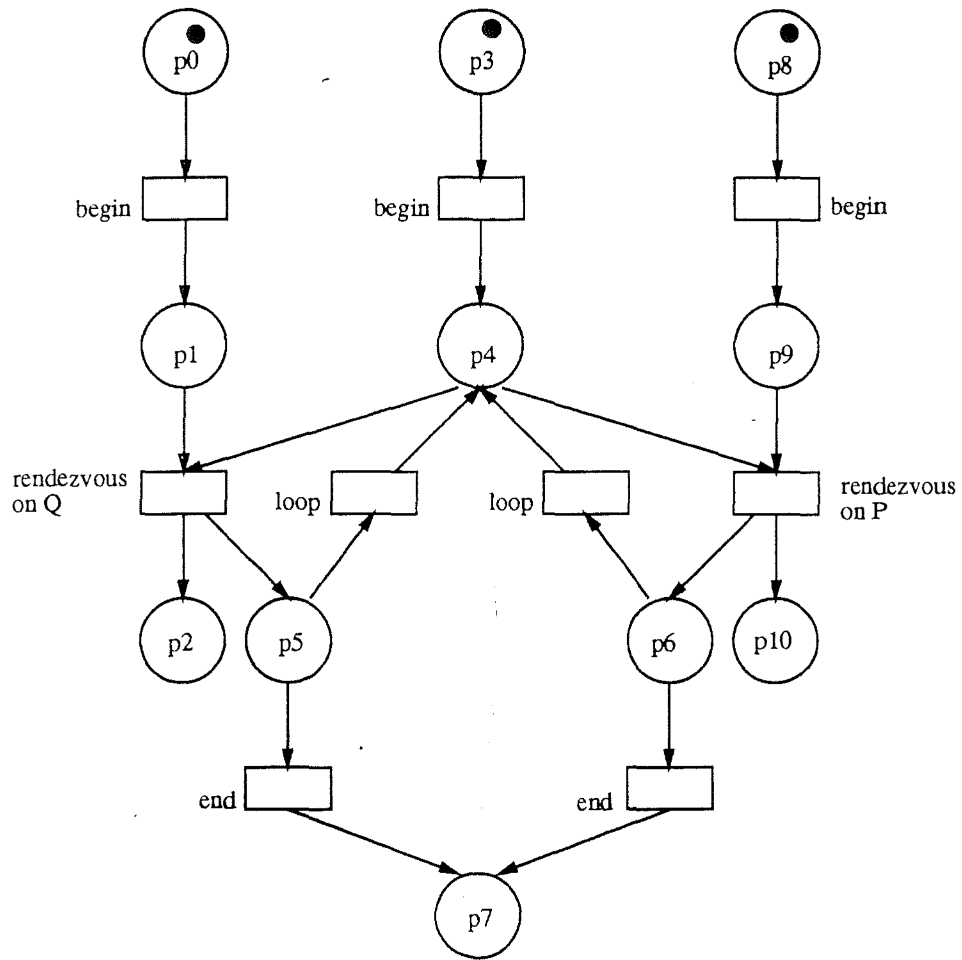


Figure 7: A Petri net corresponding to the Ada program of Figure 1.

Definition 11 (firing) *Given a net N with marking m, a transition t is enabled if and only if* $\forall$ *places* $p \in {}^{\bullet}t, m(p) \geq 1$.

The firing of a transition t enabled in a marking m produces a new marking m' defined as follows:

1. $m'(p) = m(p) - 1, \forall p \in {}^{\bullet}t - t^{\bullet}$
2. $m'(p) = m(p) + 1, \forall p \in t^{\bullet} - {}^{\bullet}t$
3. $m'(p) = m(p)$, *otherwise.*

Definition 12 (firing sequence) *Given a marking m, a firing sequence is a sequence of transitions $\sigma = < t_1, ..., t_n >$ such that transition t_1 is enabled in the marking m, and transition t_{i+1} is enabled in the marking produced by the firing of transition t_i. The marking produced by the firing of transition t_n is called the marking produced by the firing sequence σ.*

Definition 13 (reachability set) *A marking m' is reachable for a marking m if the marking m' can be produced starting from the marking m by means of a firing sequence σ.*

The set of all markings reachable for the initial marking m_0 is called the reachability set.

The reachability set represents the execution space of the Petri net, and can be described as a graph whose nodes represent markings and whose arcs represent transition firings.

The reachability graph of the Petri net of Figure 7 is presented in Figure 8. The correspondence between nodes of the reachability graph of Figure 8 and markings in the net of Figure 7 is given in Table 2, where the marking corresponding to the nodes of the reachability graph are described indicating which places are marked (in this case, the number of tokens in each place is always one).

Definition 14 (safeness) *A Petri net $N = (P, T, F, M_0)$ is* safe *if and only if for all reachable markings the number of tokens in each place is less than or equal to 1.*

From the reachability graph of Figure 8 it is easy to verify that the Petri net of Figure 7 is safe. It is always possible to represent an Ada program composed of a fixed number of tasks by means of a safe net. Intuitively, if the number of tasks in the program is bounded, it is possible to represent the tasks by disjoint sets of places and the state of each task by a single token. In the net of Figure 7 places p_0, p_1, and p_2 represent the set of possible states for task T_0; places p_3, p_4, p_5, p_6, and p_7 represent the set of possible states for task T_1; places p_0, p_9, and p_{10} represent the set of set of possible states for task T_2. Since each place represents the state of exactly one task, only the token corresponding to such a task can mark that place. The resulting net is thus safe.

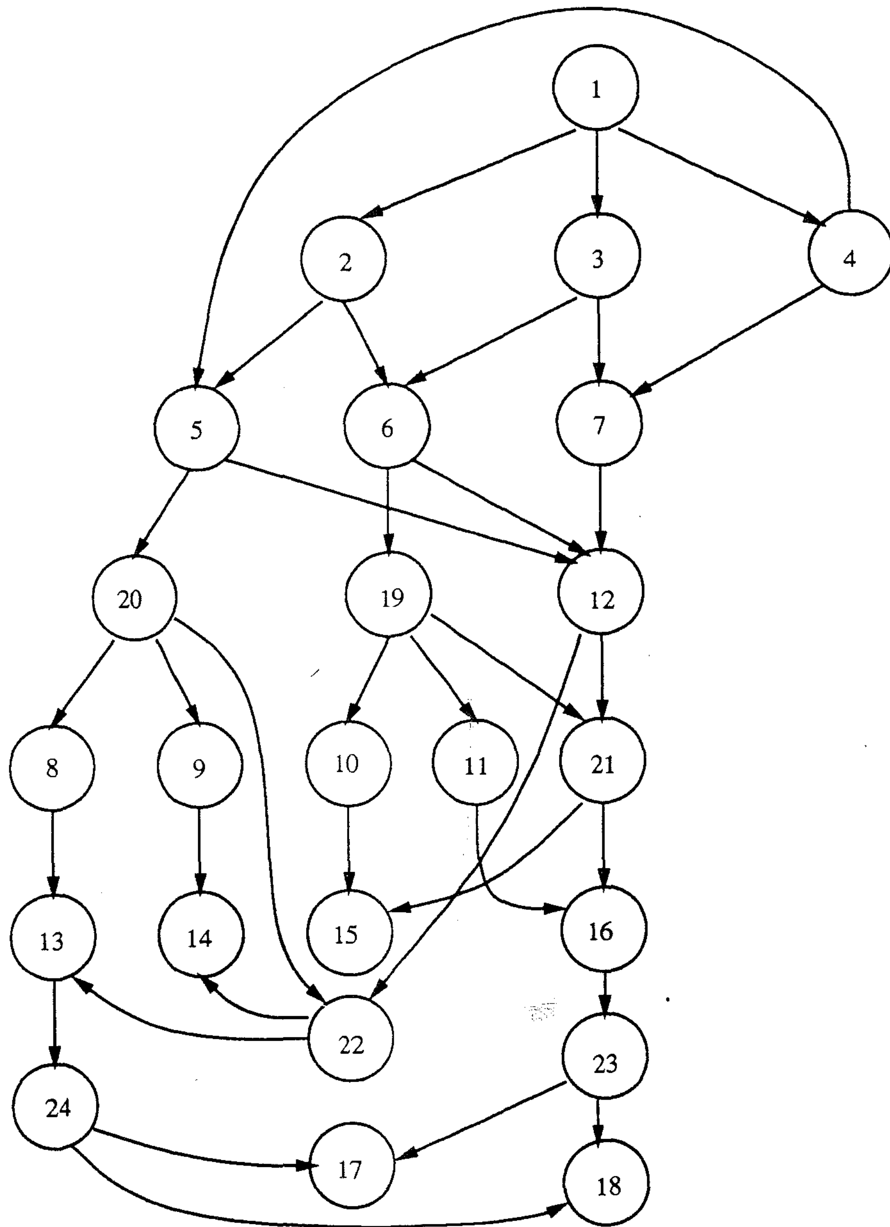


Figure 8: The reachability graph of the Petri net of Figure 7.

nodes of the reachability graph	marked place	marked place	marked place
1	p_0	p_3	p_8
2	p_0	p_4	p_8
3	p_1	p_3	p_8
4	p_0	p_3	p_9
5	p_0	p_4	p_9
6	p_1	p_4	p_8
7	p_1	p_3	p_9
8	p_0	p_4	p_{10}
9	p_0	p_4	p_7
10	p_2	p_7	p_8
11	p_2	p_4	p_8
12	p_1	p_4	p_9
13	p_1	p_4	p_{10}
14	p_1	p_7	p_{10}
15	p_2	p_7	p_9
16	p_2	p_4	p_9
17	p_2	p_4	p_{10}
18	p_2	p_7	p_{10}
19	p_2	p_5	p_8
20	p_0	p_6	p_{10}
21	p_2	p_5	p_9
22	p_1	p_6	p_{10}
23	p_2	p_6	p_{10}
24	p_2	p_5	p_{10}

Table 2: Correspondence between the nodes of the reachability graph of Figure 8 (first column) and the markings of the Petri net of Figure 7 (columns 2, 3, 4 show the corresponding marked places in the Petri net of Figure 7.)

3 A Unifying Model for Concurrency Graphs and TICGs

In this section, we show that task flow graphs and TIGs can be described by means of the same model, providing different interpretation algorithms for Ada. In this way, we prove that task flowgraphs and TIGs do not differ from the model point of view, but they only differ in the representation mechanism of Ada programs. Moreover, we define a general composition mechanism for graphs and we show that both concurrency graphs and TICGs can be obtained by the application of such a mechanism on the same model, i.e., the reachability analysis methods proposed in [Tay83b] and [LC89] are substantially equivalent. By defining a common model for task flowgraphs and TIGs, we both provide a formal definition of flowgraph based models which is Ada independent, and a means for comparing and evaluating the graph based models proposed in [Tay83b] and [LC89] and their efficacy for reachability analysis of concurrent programs. The formal definition of graph based models introduced in this section is also used in the next section as a basis for comparing these models with Petri nets.

3.1 Labeled Flow Graphs and Concurrency Flow Graphs

In order to define a compositionality rule, we introduce a set of labels, similar to the labels used in process algebras for modeling elementary actions [Mil89, BK84, BvG87]. The correspondence will be discussed in detail in Section 5.

Definition 15 (action) *Given a set Σ of names and a set $\overline{\Sigma}$ of co-names, such that for each name $\sigma \in \Sigma$, there exists a name $\overline{\sigma} \in \overline{\Sigma}$ ($\overline{\sigma}$ is called the complementary action of σ), and the special action η, the set L of actions is defined as $L = \Sigma \cup \overline{\Sigma} \cup \{\eta\}$.*

The only action without a complement in L is the special action η.

Definition 16 (labeled flowgraph) $G = (N, A, L, s, T, l)$ *is a* labeled flow graph *if and only if*

1. *(N, A, s, T) is a flow graph (see Definition 2);*
2. *L is a set of actions;*
3. *$l : A \to L$, is a labeling function for arcs.*

The special action η will be used to label local actions (i.e. non-synchronization actions); actions belonging to the set $A \cup \overline{A}$ will be used to label synchronization actions. The presence of complementary actions in different graphs allows the identification of corresponding synchronization actions in different processes.

Definition 17 (program) *A program is a finite set of labeled flowgraphs sharing the same set of labels; thus potentially communicating.*

Definition 18 (concurrency state) *Given a program* $P = \{(N_i, A_i, s_i, T_i) \mid 1 \leq i \leq k\}$, *a concurrency state is a k-tuple* $< n_1, ..., n_k >$ *of state-nodes, one for each labeled flow graph in the program; a concurrency state* $M =< n_1, ..., n_k >$ *is a successor of a concurrency state* $N =< m_1, ..., m_k >$ *if and only if*

either $\exists i, j$ *such that*

1. $\forall l \neq i, j, n_l = m_l$
2. $< n_i, m_i > \in A_i \wedge < n_j, m_j > \in A_j \wedge l(< n_i, m_i >) = \overline{l(< n_i, m_i >)}$

or $\exists i$ *such that*

1. $\forall l \neq i, n_l = m_l$,
2. $< n_i, m_i > \in A_i \wedge l(< n_i, m_i >) = \eta$

The initial concurrency state *is the k-tuple* $< s_1, ..., s_k >$.

A *concurrency flow graph* is the transitive closure of the successor relation. A *final concurrency state* is a concurrency state in the concurrency flow graph without successors.

A concurrency state represents the *global* state of the concurrent program as the set of the *local* states of all the sequential processes composing the concurrent program. A concurrency state M is a successor of a concurrency state N if M can be reached from N either through a synchronization action between two sequential processes in the concurrent program (the synchronization action is represented by the simultaneous evolution of the two processes on arcs labeled with complementary actions), or it be reached from N through a non-synchronization action performed by one of the component processes.

3.2 Concurrency Graphs as Concurrency Flow Graphs

An algorithm for representing Ada programs by means of labeled flowgraphs can be given recursively on the basic operators: the labeled flowgraph corresponding to a particular Ada program can be obtained by repeated applications of the rules, once the statements determining possible synchronization activities has been extracted.

Algorithm 1 *Let A be an Ada task;*

1. *the first step of the algorithm consists in extracting from task A all the synchronization activities together with the flow relations among them in a manner analogous to the one proposed in Section 2.1.*
2. *the second step consists of the recursive application of the following rules to the statements composing task A, starting from the innermost statements, i.e., the statement at the deepest nesting level in the Ada program, until the most external structure has been reached:*

Figure 9: The labeled flowgraph corresponding to a begin statement.

(a) *The statement* begin *can be represented by a pair of nodes and an arc joining the two nodes labeled with a silent action* η *as in Figure 9.*

(b) *A* call *to an* entry Q *can be represented by a pair of nodes and an arc joining the two nodes labeled with* Q *as in Figure 10. Here we assume unique names for entries in the whole program: there does not exists two entries in two different tasks with the same name; thus it can be assumed that the label* Q *uniquely identifies the entry* Q *of task* T_i*. This does not affect the generality of the algorithm, since names can be always disambiguated at compilation time.*

(c) *the* accept of an entry Q *can be represented by a pair of nodes and an arc joining the two nodes labeled with* $\overline{(Q)}$ *as in Figure 11.*

(d) *the* select *of two or more statements* $S_1, ..., S_n$*, being guarded or not, can be obtained by joining the initial places of the subgraphs representing the statements that occurs in the body of the select as in Figure 12.*

(e) *the* sequential composition *of two statements* $S_1; S_2$ *can be obtained by joining the final place of the subgraph corresponding to statement* S_1 *with the initial place of the subgraph corresponding to statement* S_2 *as in Figure 13.*

(f) *A* loop *can be represented, starting from the subgraph modeling the body of the loop, by adding a return arc for each terminal node of the subgraph representing the body of the loop. If the loop contains exit conditions then an arc leading to an external node should be added for each node from where the loop can be exited, as in Figure 14.*

(g) *an* end *statement can be represented by simply marking the terminal nodes of the preceding statements as terminal.*

The labeled flowgraphs obtained for the program of Figure 1 are shown in Figure 15, the corresponding concurrency flowgraph is shown in Figure 16; the correspondence between nodes of the concurrency flowgraph and nodes of the labeled flowgraphs is given in Table 3.

It is easy to see the correspondence between the labeled flowgraphs of Figure 15 and the task flowgraphs of Figure 2, and between the concurrency flowgraph

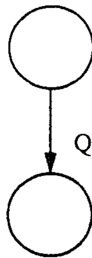


Figure 10: The labeled flowgraph corresponding to an entry call.

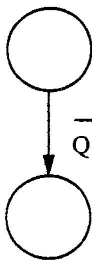


Figure 11: The labeled flowgraph corresponding to an accept statement.

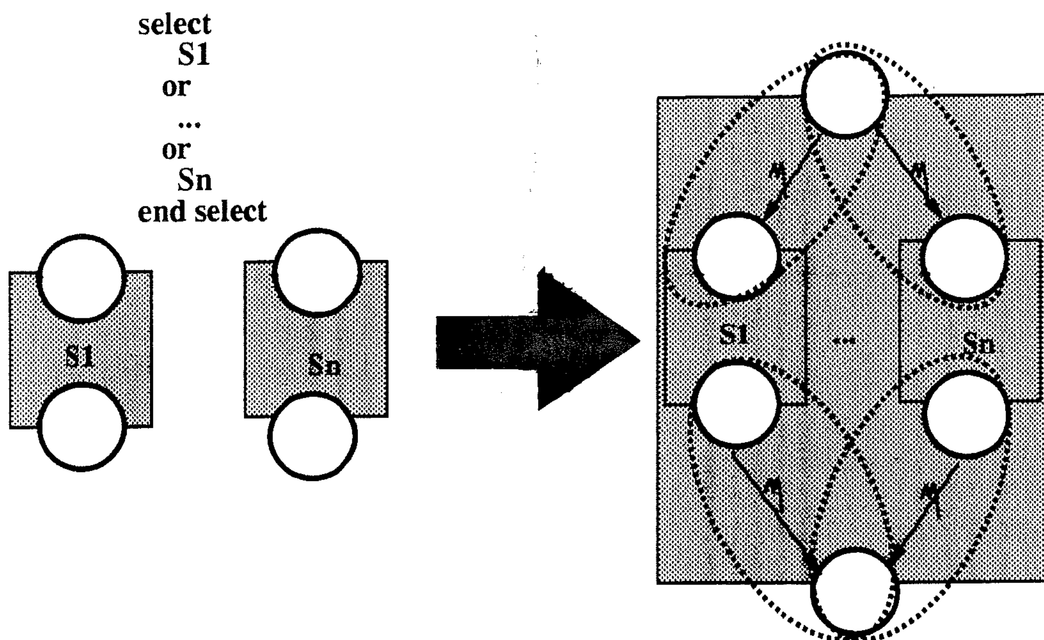


Figure 12: The labeled flowgraph corresponding to a select statement.

concurrency-node	state-node	state-node	state-node
1	1	4	9
2	1	5	9
3	2	4	9
4	1	4	10
5	1	5	10
6	2	5	9
7	2	4	10
8	1	5	11
9	1	8	11
10	3	8	9
11	3	5	9
12	2	5	10
13	2	5	11
14	2	8	11
15	3	8	10
16	3	5	10
17	3	5	11
18	3	8	11
19	3	6	9
20	1	7	11
21	3	6	10
22	2	7	11
23	3	7	11
24	3	6	11

Table 3: Correspondence between the nodes of the concurrency flowgraph of Figure 16 (first column) and the nodes of the labeled flowgraphs of Figure 15 (columns 2, 3, 4).

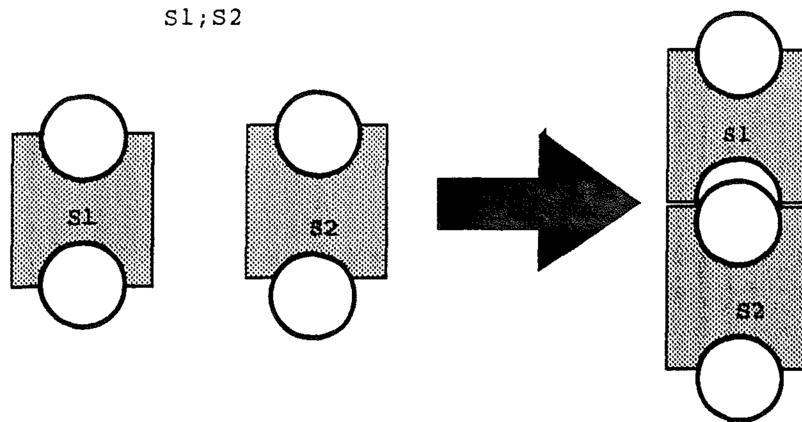


Figure 13: The labeled flowgraph corresponding to the sequential composition of two statements.

of Figure 16 and the concurrency graph of Figure 3. The main difference between the labeled flowgraphs of Figure 15 and the task flowgraphs of Figure 2 is the representation of the loop iteration: in Figure 2, the loop iteration is represented with an arc connecting two extra nodes indicated in the picture by two black dots, while in Figure 15, is represented with two different arcs, one for each possible termination point of the body of the loop. This duplication causes the presence of few more nodes in the concurrency flowgraph, indicated in bold in Figure 16: the concurrency graph of Figure 3 can be obtained from the concurrency flowgraph of Figure 16 by simply deleting nodes 19, 20, 21, 22, 23, 24.

The correspondence shown for the example is fully general, and derives from the definition of task flowgraphs and the algorithm used for interpreting Ada programs by means of labeled flowgraphs: in both the cases the nodes correspond to the same set of Ada statements and the arcs connect the same nodes. Moreover, the algorithms for building a concurrency graph starting from a set of task flowgraphs and the definition of concurrency flowgraph starting from a set of labeled flowgraphs are based on the matching of elements which are related to the same Ada constructs. In other words, the algorithm for translating Ada programs into labeled flow graphs can be considered as a formalization of the concepts introduced in Section 2.1.

3.3 TIGs as Flow Graphs

An algorithm for translating Ada tasks to labeled flow graphs so that the obtained flowgraphs will be equivalent to TIGs can be given in a very similar way.

Algorithm 2 *Let A be an Ada task*

1. *the first step of the algorithm consists in extracting from task A the task regions together with their entry points and their exit points (see Definition 5).*

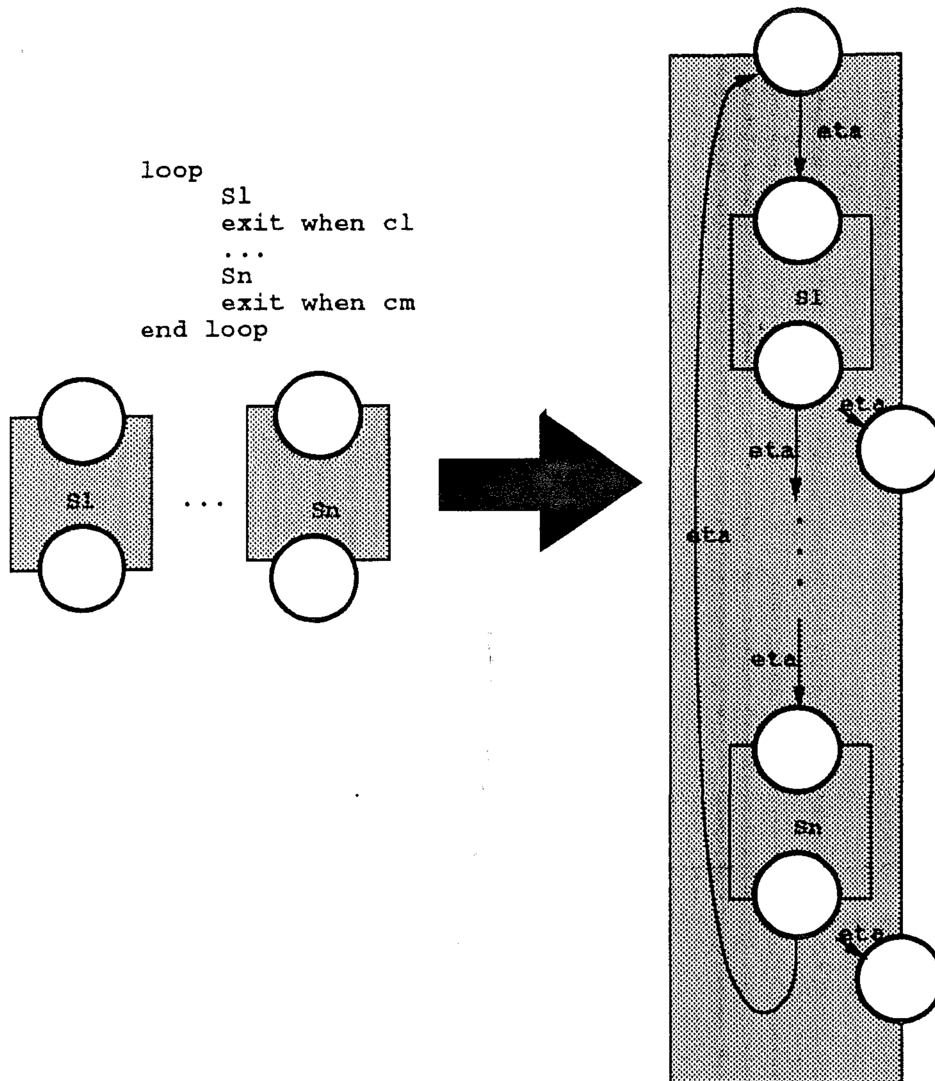


Figure 14: The labeled flowgraph for a loop statement.

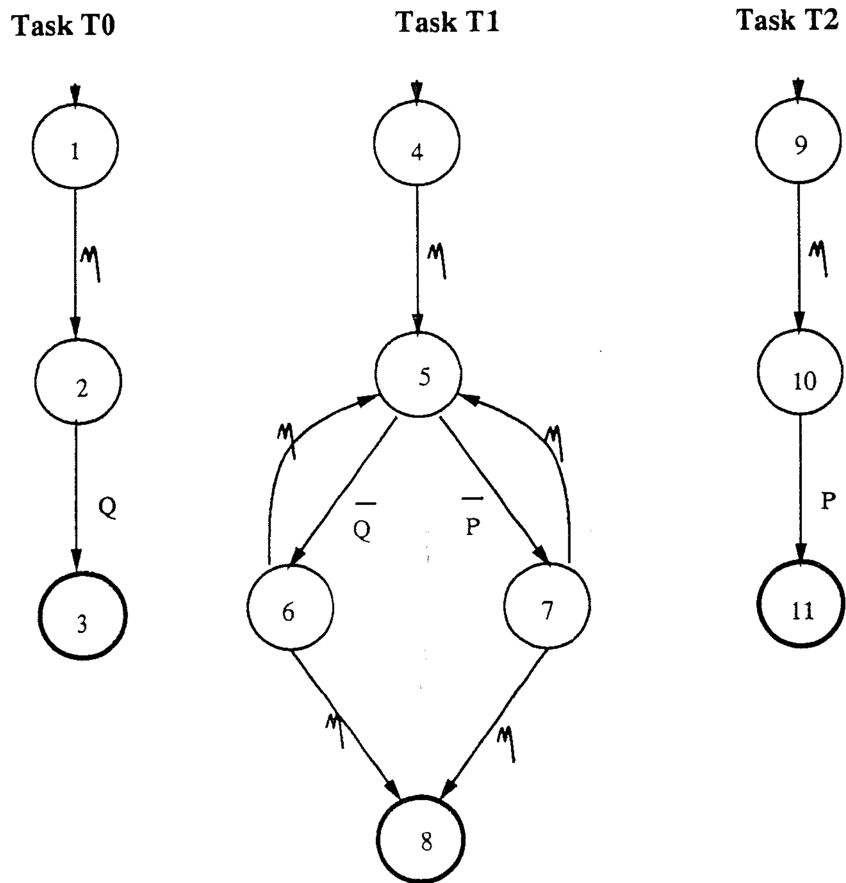


Figure 15: The labeled flowgraphs corresponding to the program of Figure 1.

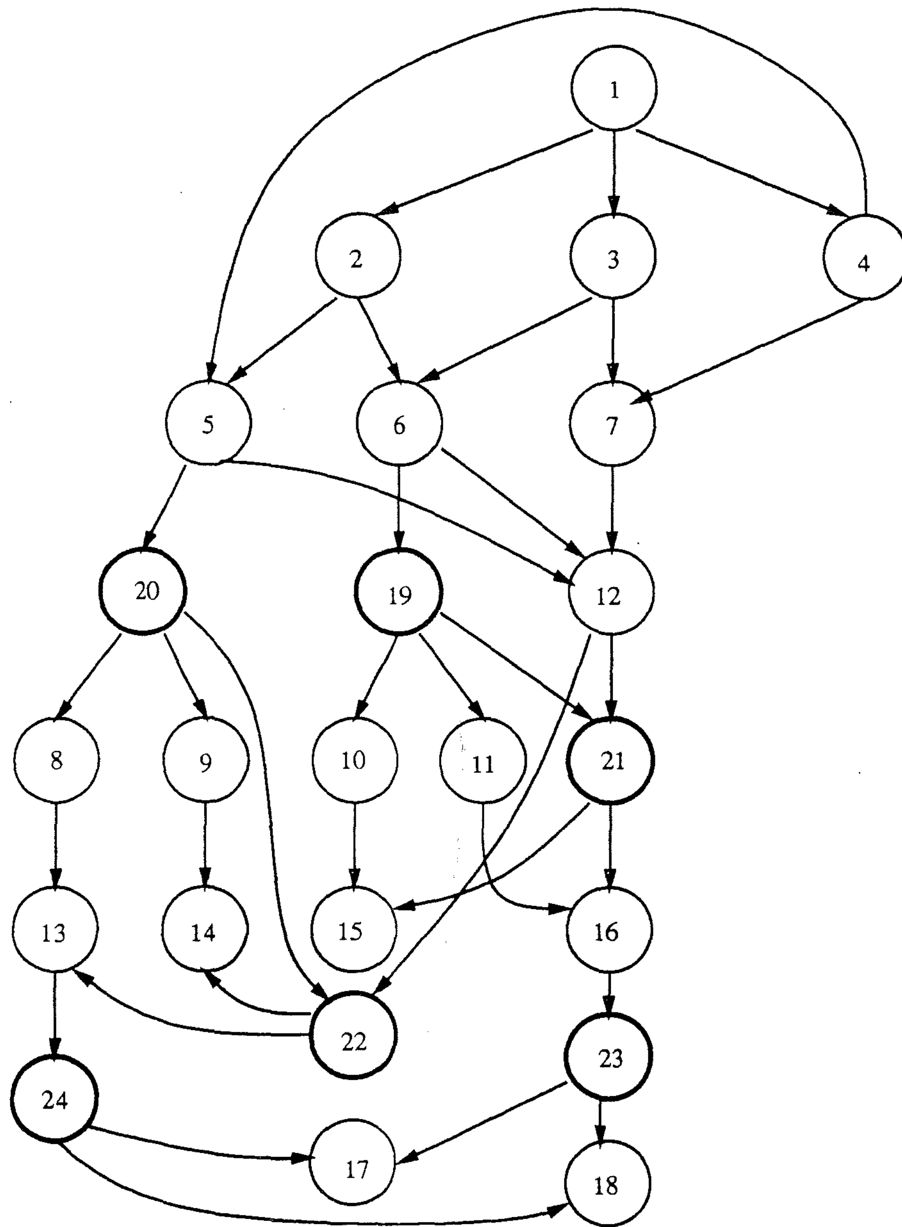


Figure 16: The concurrency flowgraph for the program of Figure 1 obtained from the labeled flowgraphs of Figure 15.

2. *the second step consists in the recursive application of the following rules to the task regions comprising task A:*

   (a) *for each task region identified at step 1, add a node to the labeled flowgraph.*

   (b) *for each possible synchronization action linking two task regions (for more details see Section 2.2) a and b, add an arc going from the node corresponding to task region a to the node corresponding to task region b, i.e., add an arc from node a representing task region a, to node b representing task region b, if task region a ends with the start of execution of an accept statement for entry X (resp. a call statement of an entry X), and task region b starts with the end of the execution of an accept statement of the corresponding entry X (resp. the end of a call of the corresponding entry X).*

   (c) *if the synchronization action linking task regions a and b corresponds to a call of an entry E, then add label E to the arc going from the node corresponding to task region a to the node corresponding to task region b.*

   (d) *if the synchronization action linking task regions a and b corresponds to an accept statement for entry E, then add label $\overline{E}$ to the arc going from the node corresponding to task region a to the node corresponding to task region b* [4].

   (e) *The initial node corresponds to the initial region of the task.*

   (f) *The final nodes correspond to task regions with an exit to the end of the program.*

The labeled flowgraphs and the concurrency flowgraph obtained for the Ada program of Figure 1 are the TIGs shown in Figure 5 and the TICG shown in Figure 6 respectively.

The correspondence shown for the example is fully general, and derives from the definition of TIGs and the algorithm used for interpreting Ada programs by means of labeled flowgraphs: in both the cases the nodes corresponds to task regions and the arcs to transfers of control between task regions and are labeled consistently. Moreover, the algorithms for building a TICG starting from a set of TIGs and the definition of concurrency flowgraph starting from a set of labeled flowgraphs are based on the matching of elements which are related to the same control transfers (see Definition 18).

4 Concurrency Flow Graphs and Petri nets

In this section, we define an algorithm for translating a program (defined as a set of labeled flowgraphs (see Definition 17) into a Petri net, and we show that the

[4] We assume unique names for all the entries in the whole program, as in Section 3.2.

reachability graph of the Petri net obtained from a set of labeled flowgraphs is the concurrency flowgraph corresponding to the same set of labeled flowgraphs.

Algorithm 3 (representing a program as a Petri net) *Given a concurrent program, i.e., a set of labeled flowgraphs according with Definition 17, it is possible to obtain a Petri net representation by applying the following rules:*

- *for each node of each labeled flow graph, add a place to the Petri net.*
- *for each arc a in the set of labeled flow graphs from node n_1 to node n_2 such that $l(a) = \eta$, add a transition t to the Petri net; the preset of transition t is the place corresponding to node n_1 and the postset of transition t is the place corresponding to node n_2.*
- *for each pair of arcs $< a, b >$ in the set of labeled flow graphs, arc a from node n_1 to node n_2, arc b from node n_3 to node n_4, such that $l(a) = \overline{l(b)}$, add a transition t to the net; the preset of transition t contains the places corresponding to nodes n_1 and n_3, the postset of transition t contains the places corresponding to nodes n_2 and n_4.*
- *if node n is an initial node then the corresponding place is marked with one token (in the initial marking)*
- *if node n is not an initial node then the corresponding place is not marked (in the initial marking)*

The Petri net obtained from the set of labeled flowgraphs of Figure 15 is the one already presented in Figure 7. The reachability graph of the Petri net of Figure 7 is the concurrency flowgraph obtained from the same set of labeled flowgraphs, as it is easy to see from Figures 16 and 8. This observation is fully general and can be stated as a theorem.

Theorem 1 *Let P be a program, i.e., a set of labeled flowgraphs, and PN the Petri net obtained from program P applying Algorithm 3. The reachability graph of PN and the concurrency graph of P are isomorphic, i.e., for each node n of the reachability graph there exists a node n' in the corresponding concurrency flowgraph, and for each arc a in the reachability graph there exists an arc a' in the corresponding concurrency flowgraph so that if arc a goes from node n to node m in the reachability graph, arc a' goes from node n' to node m' in the concurrency flowgraph, being node n' (m') the node of the concurrency flowgraph corresponding to node n (m) of the reachability graph, and vice versa.*

Proof

The theorem can be proven by induction on the size of the labeled flowgraphs comprising program P.

Basis of the induction

Given a program P composed of k labeled flowgraphs, each one composed of a single initial node and no arcs, the Petri net PN obtained from program P applying

algorithm 3 is composed of k marked places, one for each node in program P, no transitions, and no arcs. Both the concurrency flowgraph for P and the reachability graph for PN are composed of a single node and no arcs, and thus are isomorphic.

Induction step

Let P be a program composed of k labeled flowgraphs F_1, ..., F_k. Let h_i and k_i be respectively the number of nodes and arcs of flowgraph F_i for $1 \leq i \leq k$. Let PN be the Petri nets obtained from program P with algorithm 3.

Let assume by inductive hypothesis that the concurrency flowgraph corresponding to program P and the reachability graph for the Petri net PN are isomorphic. In order to demonstrate the theorem, we must prove that the concurrency flowgraph for program P', obtained from program P by adding either a node or an arc to one of the flowgraphs comprising P, let say flowgraph F_i, and the reachability graph of the Petri net PN' obtained from program P' with Algorithm 3 are isomorphic. In fact it is always possible to obtain a program composed of k flowgraphs in a finite number of steps starting with a program compose of k flowgraphs each one composed of a single initial node and adding at each step either a node of an arc to one of the flowgraphs comprising P.

If program P' is obtained from program P by adding a node to flowgraph F_i, then the concurrency graph of program P is also the concurrency flowgraph of program P': since the concurrency flowgraph is defined as the transitive closure of the successor relation, which is defined starting from arcs and their associated labels, adding only places does not change the reachability space.

The Petri net PN' corresponding to program P' is characterized by the same set of transitions and arcs of PN and a new set of places obtained from the set of places of PN adding a new unmarked place p with empty preset and postset. Place p can never be marked and consequently the reachability graph of PN' is the reachability graph of PN.

The concurrency flowgraph corresponding to program P' and the reachability graph corresponding to the Petri net PN' are isomorphic by inductive hypothesis.

Adding an arc a from state-node m to state-node n to the flowgraph F_i changes both the concurrency flowgraph and the reachability graph of the corresponding Petri net.

For what concern the relations between the Petri net PN' corresponding to program P' and the Petri net PN corresponding to program P there can be two cases:

1 *arc a is labeled with η, i.e., the added arc does not represent a possible new synchronization. The Petri net PN' corresponding to program P' can be obtained from the Petri net PN corresponding to program P by adding a new transition t with only the place p_m corresponding to the node m in its preset and only the place p_n corresponding to the node n in its postset.*

  The concurrency flow graph CG' corresponding to program P' can be obtained from the concurrency flowgraph CG corresponding to program P adding an exiting arc ca from each concurrency-node cm containing the state-node m as i-th

component. Let cm' be the concurrency-node hit by arc ca. The concurrency-node cm' differs form the concurrency-node cm for the i-th component which is the state-node n instead of m. The concurrency-node cm' can either be already in CG or be a new concurrency-node. If cm' is in the concurrency graph CG, the concurrency graph CG' differs from the concurrency graph CG only for arc ca. If the concurrency-node cm' is not in the concurrency graph CG then cm' can be either a final node or the source of new concurrency-arcs. In the first case the concurrency graph CG' differs from the concurrency graph CG for arc ca and state cm', otherwise it differs for a subgraph, rooted in cm'.

To show that the concurrency flowgraph CG' corresponding to P' and the reachability graph RG' corresponding to PN' are isomorphic, we must show that for each new arc ca exiting a concurrency-node cm leading to a concurrency-node cm', in the concurrency-flowgraph CG' there is an arc ra, corresponding to ca, exiting the node rm corresponding to the concurrency-node cm in the reachability graph RG' . We must also show that the node rm' reached from rm through arc ra corresponds to the concurrency-node rm'.

Since the concurrency-node cm belongs to the concurrency flowgraph CG, in the reachability graph RG corresponding to PN there exists a node rm that corresponds to cm by inductive hypothesis. Since the concurrency-node cm contains as i-th component the state-node m, the corresponding node rm corresponds to a marking where place p_m is marked, being p_m the place corresponding to node m. Thus in the reachability graph RG' exists an arc ra corresponding to the firing of the new transition t added to the net exiting node rm (transition t has only place p_m in its preset and thus it can fire). Arc ra leads to a node rm' that represents a marking where all the nodes marked in rm are still marked, except for place p_m, that is not marked any more and place p_n corresponding to node n that becomes marked. Arc ca corresponding to arc ra in the concurrency flowgraph leads to a node cm' differing from node cm only for the i-th component, which is the state-node n instead of m. If the concurrency node cm' belongs to CG, node rm' belongs to RG by inductive hypothesis; in this case the theorem is proved.

If the concurrency-node cm' is a terminal node in the concurrency graph CG', we must prove that node rm' is a terminal in the reachability graph RG'. Let us assume by contradiction that rm' is not a terminal node, then there exists an arc exiting rm', that corresponds to a transition t' enabled in marking m' corresponding to state rm'. In state rm' all the places which were marked in rm are still marked, except for the places in the preset of transition t' which are not marked any more and the places in the postset of transition t' which are newly marked. By construction, transition t' corresponds to an arc a' in flowgraph F_i. Since the components of the concurrency state cm' corresponds to the place marked in the marking represented with state rm', a new concurrency state differing from state rm' for the components corresponding to the places in the preset of transition t' can be reached from rm' through arc a'.

But this contradicts the hypothesis that rm' is a terminal state in CG'. Thus state rm' is also a terminal node and the theorem is proven.

Finally, if the concurrency state cm' is not in CG and is not a terminal node for CG', we must prove that node rm' is not in RG and is not a terminal node in RG'. We must also prove that for each node that can be reached from cm' in CG' there is a corresponding node in RG'. Since the concurrency graph is finite (see Theorem 2), the proof can be iterated until either a node already in the considered flowgraph or a terminal node is reached.

If the concurrency node cm' is not in CG, the corresponding node rm' is not in RG by inductive hypothesis. If the concurrency node cm' is not a terminal node in CG' then there exists at least an arc a' exiting cm'. Arc ca' corresponds to either one arc a_1 labeled η or two arcs a_2 and a_3 with complementary labels in the flowgraphs comprising program P'. Let assume that arc ca' corresponds to one arc a_1 labeled with η, the demonstration in the other case is analogous. In PN' there exists a transition t' corresponding to arc a_1 by construction. Since arc ca' exits the concurrency state cm', one component of cm' (e.g., component i) corresponds to the input node of arc a_1. Since state rm' corresponds to state cm', place p_i corresponding to the i-th component of the concurrency-state cm' is marked in rm'. thus transition t' is enabled in the marking corresponding to state rm'. The firing of transition t' corresponds to an arc ra' in the reachability graph RG'. ra' corresponds to arc cm' and leads to a state rm'' that corresponds to state cm'' reached from cm' through arc a'. State rm'' corresponds to cm'' because they differ from corresponding states (rm' and cm') for corresponding components (e.g., the components corresponding to the input/output state of arc ca' and the preset/postset of arc ra').

2 *arc a is labeled with $l \neq \eta$, and thus there can exists arcs a_j from node m_j to node n_j belonging to some others flowgraphs of program P labeled with $\bar{l}$. The net PN' can be obtained from the net PN by adding a transition t_j for each pair of new matching arcs $< a, a_j >$ with places p_m and p_{m_j}, corresponding to nodes m and m_j, in its preset and places p_n and p_{n_j}, corresponding to nodes n and n_j, in its postset.*

The concurrency flow graph CG' corresponding to program P' can be obtained from the concurrency flowgraph CG corresponding to program P adding an exiting arc ca from each concurrency-node cm containing both the state-nodes m and m_j as i-th and j-th components.

The correspondence between the concurrency flowgraph corresponding to program P' and the reachability graph corresponding to the Petri net PN' can be demonstrated with a deduction analogous to the deduction used for demonstrating point 1, by only changing the considerations about the differences between the concurrency-nodes connected with arc ca in the concurrency graph and the markings corresponding to the nodes connected with arc ra in the reachability

graph. In this case the two concurrency nodes connected with arc ca differs for two components instead of one, and the markings corresponding to the nodes connected with arc ra differs for four places instead of only two, but there is an analogous correspondence between nodes and concurrency-nodes.

Theorem 2 *Given a program P, i.e., a finite set of labeled flowgraphs, the corresponding concurrency graph is finite.*

Proof

The proof follows almost immediately from the definition of program and concurrency graph. Concurrency-states are finite tuples of state-nodes belonging to the flowgraphs in program P. Since the number of flowgraphs comprising a program P is finite and the number of state-nodes in each flowgraph is also finite, the number of finite tuples of state-nodes is finite.

5 Compositionality and Process Algebras

The main drawback of reachability analysis is the combinatorial state explosion, which makes the application of reachability analysis to large complex systems virtually impossible. Although theoretical results imply that the combinatorial state explosion cannot be avoided in the worst case ([Lad79, Tay83a, Smo84, Apt83]), it is possible to obtain significant improvements in many practical cases. The suitability of only one the the two approaches (viz. flowgraphs and Petri nets) to support efficient reachability analysis for a significant set of practical cases would be critical in the comparative evaluation of the two approaches. In this section, we review the approach proposed in [YY91], where the complexity of reachability analysis is controlled by considering labeled flowgraphs as terms of a process algebra and applying suitable algebraic properties to the composition of labeled flowgraphs. In the next section, we show that similar results can be obtained by considering Petri nets a subcategory of the two sorted algebras over multisets and using suitable composition mechanism over Petri nets, thus showing that the two approaches can be considered equivalent from this point of view as well.

The conventional approach to reachability analysis as presented in Section 2 of this paper is based on the construction of the whole reachability graph in a single step, i.e., first either the set of labeled flowgraphs or the Petri net corresponding to the program is built and then the whole reachability graph is obtained from the representation of the program. Any change in the starting program requires the construction of a new reachability graph from scratch, without any chance of reusing the results obtained by analyzing the program before the last changes. Moreover, the approaches so far reviewed do not give any means of reusing the results obtained by reachability analysis of subsystems for analyzing the whole system. Thus, the reachability analysis of a system must be carried out from scratch even if its subsystems have been previously analyzed.

In contrast, an incremental approach allows the derivation of the reachability graph of a system to be built step by step by incremental modifications of the reach-

ability graph built from an initial version of the system, without requiring the construction of a new reachability graph for each step of the incremental development of the system. A compositional approach allows the reuse of the reachability analysis of the subsystems for analyzing a system obtained by composition of different subsystems. Incremental and compositional approaches can result in a substantial reduction of the complexity of the reachability analysis for a wide class of modular and incrementally built systems.

In order to introduce compositionality, a composition mechanism has to be defined for both labeled flowgraphs and concurrency flowgraphs, and it has to be associative (i.e., the order in which parts are combined together must not affect the final result). The extension of the composition rule, as defined in Section 3 for concurrency flowgraphs can be obtained by simply introducing a labeling function for the arcs of the concurrency flowgraphs. This can be easily done by attaching a label representing the action modeled by the arc to each arc of the concurrency flowgraph. Unfortunately the so-defined operator is not associative, as shown in Figures 17, 18, and 19. Figure 17 shows a program composed of three labeled flowgraphs (T_1, T_2, T_3); the concurrency flowgraph obtained as defined in Section 3 is shown in Figure 18; the concurrency flowgraphs obtained by composing the tasks incrementally is shown in Figure 19. In Figures 18 and 19, the arcs of the concurrency graph are labeled according to the actions taking place and the labeled flowgraphs involved: for instance, $Q(T_3, T_2)$ represents the simultaneous occurrence of actions Q in the flowgraph T_3 and $\overline{Q}$ in T_2.

In Figure 19, parentheses are used to indicate the order in which labeled flowgraphs are composed. (T_1 composed T_2) composed T_3 indicates the concurrency flowgraph obtained by first composing the labeled flowgraphs T_1 and T_2, and then composing the obtained concurrency flowgraph with the labeled flowgraph T_3.

The concurrency flowgraph built in a single step (Figure 18) shows (correctly) the two different possible evolutions of the program: either the simultaneous occurrence of actions Q in T_1 and $\overline{Q}$ in T_2 followed by actions P in T_1 and $\overline{P}$ in T_2, or actions Q in T_3 and $\overline{Q}$ in T_2 followed by actions P in T_3 and $\overline{P}$ in T_2.

The concurrency flowgraph obtained by composing only the labeled flowgraphs T_1 and T_2 shows correctly the synchronous sequence between the two flowgraphs, but the successive composition with the labeled flowgraph T_3 does not show the possible communication between tasks T_2 and T_3. The label matching between the labeled flowgraphs T_1 and T_2 hides actions which do not correspond to actual synchronization; in particular the complementary actions of the labeled flowgraph T_2 cannot be composed any more with actions P and Q of the flowgraph T_3. Similarly, if only the flowgraphs T_2 and T_3 are composed first, the complementary actions of the labeled flowgraph T_2 cannot be composed any more with actions P and Q of the flowgraph T_1. In other words, the composition operation used in conventional reachability analysis is not associative.

To overcome the lack of associativity, new composition operators need to be defined. In [YY91], a solution based on process algebras is presented. This solution considers labeled flowgraphs and concurrency flowgraphs as models of terms

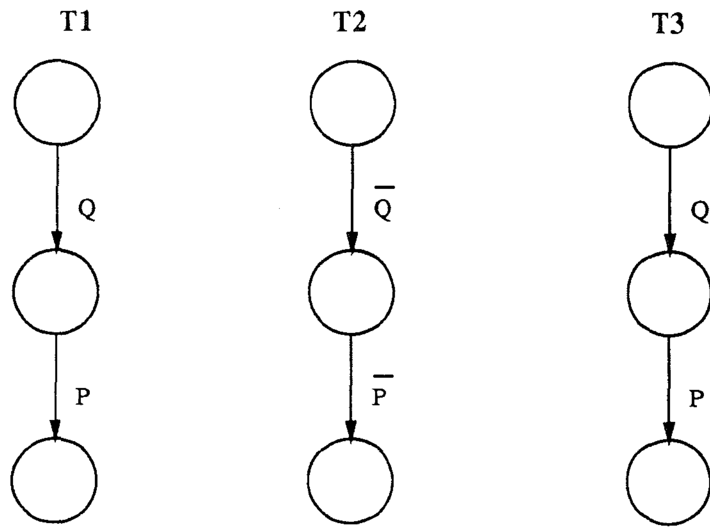


Figure 17: A simple program.

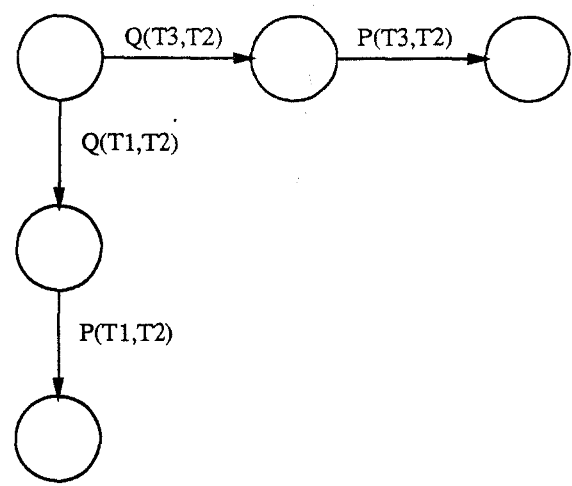
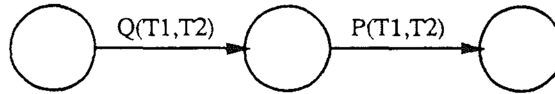


Figure 18: The concurrency flowgraph for the program of Figure 17 built in a single step.

(T1 composed T2) composed T3



(T2 composed T3) composed T1

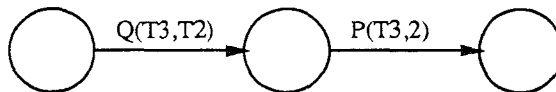


Figure 19: The concurrency flowgraphs for the program of Figure 17 built in two sequential steps.

of a process algebra, and uses the product defined in the process algebra as the composition mechanism. In the chosen algebra, the product should be associative with respect to an equivalence relation reflecting the intuitive notion of equivalence among flowgraphs.

In [YY91], the algebra utilized is $ACP-\eta$, whose definition can be found in [BK84, BvG87].

Figure 20 shows the composition of the labeled flowgraphs T_1 and T_2 of Figure 17 using the compositional approach. Both possible synchronization actions and non-synchronization actions are explicitly modeled in the concurrency flowgraph. From the initial state S_0 three actions are possible: the synchronization action $Q(T_1, T_2)$ between tasks T_1 and T_2 on action Q and its complement $\overline{Q}$, but also action Q performed by task T_1 without any move of task T_2, or action $\overline{Q}$ performed by task T_2 without any move of task T_1. In this way the successive composition of the labeled flowgraph T_3 results in the modeling of all possible synchronization actions. The same result would be obtained by first composing the labeled flowgraphs T_2 and T_3 and then adding the flowgraph T_1, as expected from the properties of the algebraic composition operator. The price paid to the associativity is an additional state explosion of the concurrency flowgraph. Fortunately, the algebraic approach provides several mechanisms for simplifying the intermediate flowgraphs without affecting the final results. For instance, in the example of Figure 20 only the arcs labeled with actions $\overline{Q}$ and $\overline{P}$ are interesting. In fact only these actions can match actions P and Q that are performed by task T_3, the only task to be further composed to obtain the final system. Arcs labeled with P and Q in the example of Figure 20 are not interesting any more, since we know they they will never match any further

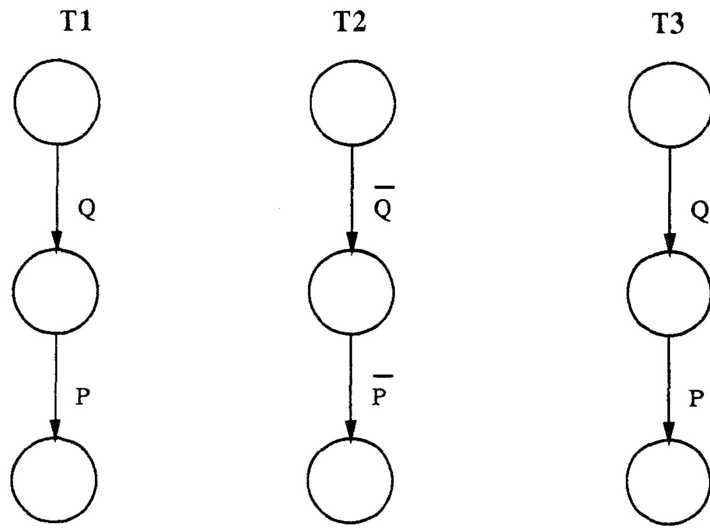


Figure 17: A simple program.

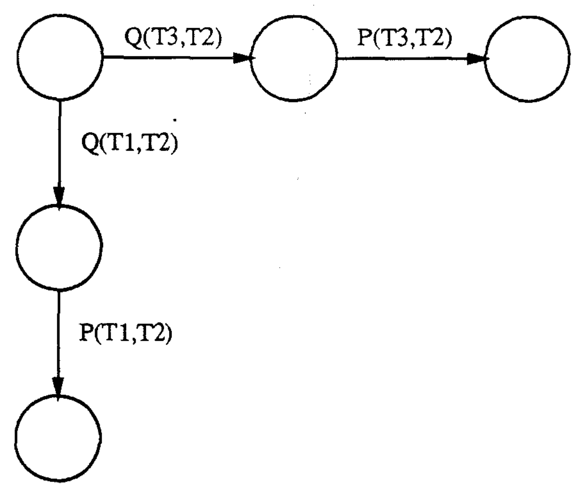


Figure 18: The concurrency flowgraph for the program of Figure 17 built in a single step.

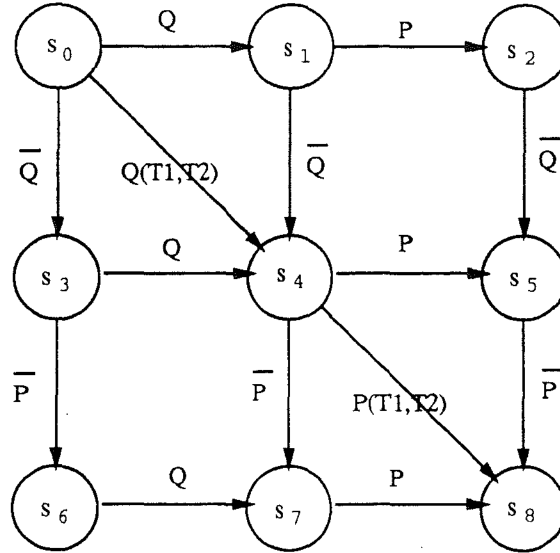


Figure 20: Composition of the labeled flowgraphs T_1 and T_2 according to the rules of ACP_η.

action that can be performed by some other part of the system not yet included in the description of Figure 20. Thus, the arcs labeled with Q and P can be dropped from the intermediate flowgraph without affecting the final result. This can be easily done by using the restriction operator of the underlying algebra.

The operators and the axioms of the underlying algebra can be used to further reduce the intermediate results, in order to incrementally simplify the final results, as shown in the example of Figure 21. This figure shows how to obtain a bounded buffer with two positions by composition of two bounded buffers with a single position.

Figure 21 a) shows two bounded buffers P and Q with capacity one. A bounded buffer with capacity one can perform only two actions: accept a message (action $\overline{a}$ for buffer P, action $\overline{b}$ for buffer Q) if the buffer is empty (state s_1 for buffer P, state s_3 for buffer Q) and transmit the received message (action b for buffer P, action c for buffer Q) if the buffer is full (state s_2 for buffer P, state s_4 for buffer Q). Different labels for the two buffers have been chosen here only for simplifying the description of the composition mechanism. Process P can be transformed to process Q by application of the relabeling operator of $ACP-\eta$, and vice versa.

Figure 21 b) shows the process $(P||Q)$ obtained by composing the bounded buffers P and Q presented above. Process $P||Q$ can evolve as either process P or process Q, or the synchronization of processes P and Q when possible. In the initial state s_5 can either accept a message on the input position a (action $\overline{a}$) as process

P and move to state s_6, or accept a message on the input position b (action $\overline{b}$), as process Q and move to state s_7. Once accepted a message on the input position a (action $\overline{a}$) (and thus being in state s_6), process $P||Q$ can either accept a message on the input position b (action $\overline{b}$) as process Q, or accept a message on the input position b as process P, or can evolve with action η corresponding to processes P and Q synchronizing on the complementary actions b and $\overline{b}$.

If we consider b as an internal channel for synchronization between processes P and Q and a and c as external channels, i.e., channels that process $P||Q$ uses to communicate with its external world, then we would like to forbid process $P||Q$ to communicate on channel b, i.e., perform action b or $\overline{b}$. Formally this is represented by the application of the hiding operator, as shown in Figure 21 c), where the process $(P||Q)\backslash b$ is shown. Initially (state s_9) process $(P||Q)\backslash b$ can only accept a message on the input channel a (action $\overline{a}$) and move to state s_{10}. From state s_{10} it can only perform action η representing an internal non-observable move, namely process P passing the newly received message to process Q on the internal channel b. From state s_{11}, it can either accept a new message on channel a (action $\overline{a}$), or transmit the former message on channel c (action c). If we only consider the visible action a and c, we can see that process $(P||Q)\backslash b$ acts as a bounded buffer with two position, i.e., it accepts at most two consecutive messages and it transmits the received messages in the same order it receives them.

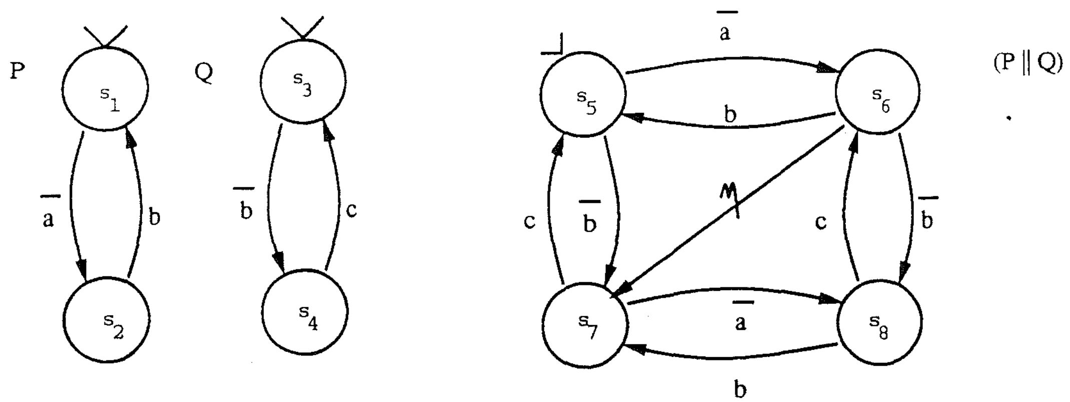
Figure 21 d) presents process $(P||Q)\backslash b$ simplified by removing the internal action η.

The example described in Figure 21 shows how the state spaces of two components (the one position bounded buffers) can be composed to obtain the state space of a bigger process (the two positions bounded buffer). Key features of process algebras that provide compositionality and incrementality to flowgraph models are:

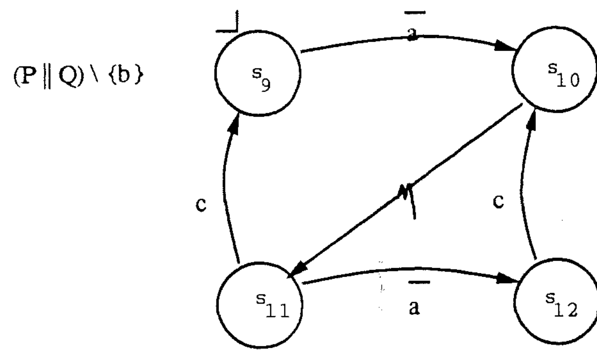
1. *an equivalence relation.*
   The equivalence relation provides mathematical support for transformation and simplification of processes; for instance, the processes of Figure 21 c) and d) are equivalent in terms of an equivalence relation defined in $ACP-\eta$, and thus the process of Figure 21 c) can be simplifyed in the process of Figure 21 d).

2. *a composition operator associative with respect to the equivalence relation.*
   The associativity of the composition operator ensures that the order of compositions of subsystems does not affect the final result; for instance, the composition of the three tasks of Figure 17 can be done in two steps obtaining the same result regardless the order of composition.

3. *a restriction operator which allow the incremental simplification of flowgraphs.*
   The restriction operator can be successfully used for abstracting away from internal actions, relevant before the composition of subsystems, but not relevant any more after their composition. Abstracting away from non essential

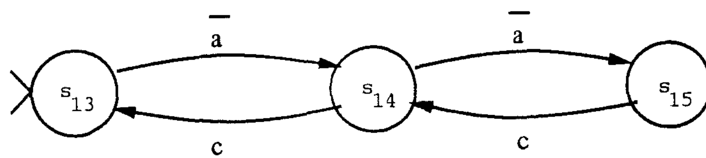


a) b)



c)

simplified (P || Q) \ {b}



d)

aspects while incrementally composing subsystems can be crucial for controlling the state explosion of the overall system. Intuitively, the number of states and actions of the resulting systems is proportional to the number of states and actions of the component subsystems, thus using the two bounded buffer representation of Figure 21 b) as a component for a bigger system instead of the simplifyed representation of Figure 21 d) would produce a much bigger system description. Moreover, the application of the restriction operator can be necessary to guarantee the correctness of construction of the final system. For instance, if the subsystem of Figure 21 b) is composed with another subsystem that is not supposed to communicate with the bounded buffer on port b, but uses b as a name for one of its local ports, then their composition could present some undesirable communication between the two subsystems.

Several tools have been constructed for experimenting with process algebras and verification of finite-state systems [CPS91, CPS90, MSGS90, Fer88]. Although these typically provide a variety of algebraic manipulations, they can be used to perform reachability analysis as a series of composition and reduction steps. Yeh [YY91] describes a prototype tool that derives both process graphs and scope structure from program texts in an Ada-like language, and then uses the scope structure to guide a sequence of composition and reduction steps and thereby avoid combinatorial explosion. The extent of the advantage of algebraic structure for reachability analysis of practical systems requires more study, but results so far are encouraging.

6 Compositionality and Petri Nets

In the former section we showed that algebraic properties can be used to reduce the complexity of reachability analysis based on flowgraphs for an interesting class of systems; namely, systems that present some kind of modularity and can thus be decomposed into subsystems that can be analyzed separately. We identified three main features needed to control the complexity of reachability analysis: an equivalence relation, a composition operator and a restriction operator. In this section we show how to introduce similar mechanisms for Petri nets and thus how it is possible to reduce the complexity of reachability analysis for the same class of systems also by using Petri nets. The theoretical background relies on category theory and it is well documented in [MM90, Win87, Win84]. Subsection 6.1 informally presents the concepts formally introduced in [Win84], stressing the concepts used in this paper. Subsection 6.2 shows how the properties of Petri nets as a category introduced in Subsection 6.1 can be used for mastering the complexity of reachability analysis based on Petri nets in a similar way as properties of process algebras can be used to master the complexity of reachability analysis based on flowgraphs.

6.1 The category safe net

In this section we refer to safe nets, since safe nets are powerful enough to represent the class of systems that can be statically analyzed, as suggested in Subsection 2.3.

In order to compare Petri nets and thus define equivalent nets we need to introduce net transformations. Since our main interest is in reachability analysis, we are looking for net transformations that preserve the reachability set of the transformed nets. Intuitively, a net transformation that preserve the initial marking and the preset and postset relations relates nets with the same reachability set.

In [Win84], Winskel calls such transformations net morphisms and shows that these transformations preserve the reachability set:

Definition 19 (Net morphism) *Let N and N' be safe nets; a net morphism from $N = (P, T, F, M)$ to $N' = (P', T', F', M')$ is a pair $< \eta, \beta >$, where $\eta : T \rightarrow T'$ is a partial function, $\beta \subseteq P \times P'$ is a relation such that*

1. $\beta(M) = M'$
2. $\forall t \in T, {}^{\bullet}(\eta t) = \beta({}^{\bullet}t) \cup (\eta t)^{\bullet} = \beta(t^{\bullet})$

Theorem 3 *Net morphisms preserve reachable markings; i.e., if M is a reachable marking of a net N and $< \eta, \beta >$ is a net morphism, then $\beta(M)$ is a reachable marking of N'.*

Proof
see [Win84]

Isomorphic nets, i.e. nets that can be transformed one into the other by means of a bijective morphism, can be considered equivalent from the static analysis point of view. Isomorphisms for the category of safe nets play the role of equivalence relations for process algebras.

In [Win84], Winskel also shows that safe nets together with net morphisms as defined above form a category with product, the *category of safe nets.* Products in categories have several nice properties very important in our framework, the most important one being associativity with respect to isomorphisms, i.e. $((N_1 \times N_2) \times N_3)$ is isomorphic to $(N_1 \times (N_2 \times N_3))$. One can think of products in the category of nets as the basic composition mechanism among nets. Figure 22 shows two simple nets (N_0 and N_1) and their product ($N_0 \times N_1$). Intuitively, the product of two nets contains all the transitions of the original nets with the same preset and postset (transitions $t_{0,0}$ and $t_{1,0}$ in the net of Figure 22) and a new transition for each pair of transitions in the original nets with the union of the presets and the postsets of the original transitions (transition $t_{0,1}$ in the net of Figure 22). The transitions of the original nets represents the actions of the original nets that can take place independently, while the new transitions represent all the possible synchronizations among the actions of the original nets. The set of places of the product of two nets is the union of the sets of places of the two component nets. It is possible to verify that the net-product preserves the reachability sets of the components, i.e. a marking M is reachable in $N_0 \times N_1$ if and only if $\rho_0(M)$ is reachable in N_0 and $\rho_1(M)$ is reachable in N_1 (ρ_0 and ρ_1 indicates the projections associated to the net-product, i.e., the marking related to the set of places corresponding to the selected component).

(N0 x N1)\{<t0,0>}

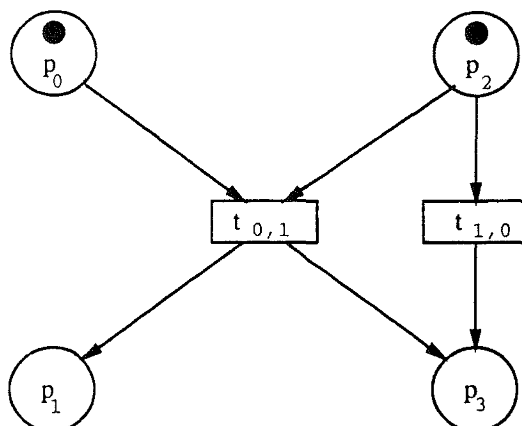


Figure 23: Effects of the application of the net-restriction operator.

The net-product constitutes a basis for composing nets, but it cannot be used as it is: in general, subsystems can communicate only on specific actions and not on any pair of actions as represented by the net product. Thus we need a way of defining the set of communication actions. In [Win84] this problem is solved by using a particular morphism, called *net-restriction*, that can be used to abstract away non-interesting actions. The effect of the net-restriction operator is to delete a subset of transitions from the original net, as shown in the example of Figure 23, where the net $(N_0 \times N_1)\setminus \{< t_{0,0} >\}$ is obtained from the net $(N_0 \times N_1)$ of Figure 22 by restriction over transition $< t_{0,0} >$.

The net-composition operator can be obtained from the net-product by abstracting away from all the actions resulting from composition of non-communication actions.

To define the net-composition operator starting from the net- product and the restriction operator, we assume a labeling mechanism, which associates with each transition a label in a set of actions $L = A \cup \overline{A} \cup \{\eta\}$. The label associated to transition t will be indicated by $\lambda(t)$.

Definition 20 (net composition) *Let* $N_0 = (P_0, T_0, F_0, M_0)$ *and* $N_1 = (P_1, T_1, F_1, M_1)$ *be nets; the composition* $N_0 || N_1$ *is defined as*
$(N_0 \times N_1)\setminus \{ < t_i, t_j >| \ t_i \neq 0 \wedge t_j \neq 0 \wedge \lambda(t_i) \neq \overline{\lambda(t_j)} \ \}$

The net composition is a net with all the transitions of the components plus a transition for each pair of transitions in the component nets with complementary labels, as shown in the example of Figure 24.

In Figure 24, the net $N_0 || N_1$ contains all the transitions corresponding to independent actions plus a transition obtained by matching the pair of transitions

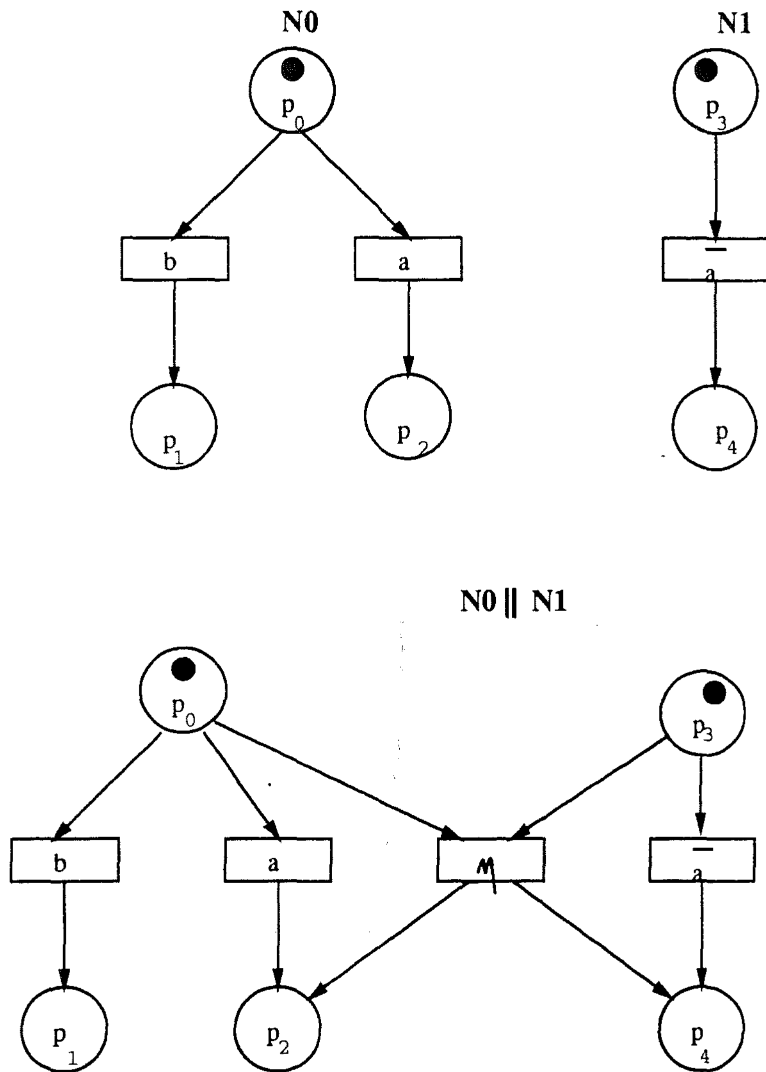


Figure 24: The composition of the two simple nets of Figure 22.

labeled a and $\overline{a}$, but not the transition obtained by matching transition b of net N_0 with transition a of net N_1. The transition corresponding to independent actions are labeled according to the action represented. The transitions corresponding to a synchronization are labeled with the null action η.

More generally the net-restriction operator can be used to abstract away from details before further composing the subsystems, similarly to the algebraic restriction operator. Like the algebraic restriction operator, the net-restriction operator is fundamental in order to be able to master the extra complexity introduced in the system by using the new composition operator. Any time we can deduce that some actions of the subsystems will never match actions performed by units in the rest of the system, those actions can be hidden using the net-restriction operator, thus simplifying the reachability space of the subsystem without exporting the additional complexity into the reachability graph of the whole system.

6.2 The bounded buffer example

In this subsection, we demonstrate how the net composition operator and the restriction operator can be used to compose different sub-networks and simplify the intermediate result using the bounded buffer example already discussed in Section 5

The Petri nets modeling the two processes P and Q and the net obtaining by composing the two processes are shown in Figure 25.

The reachability graph of the net $P||Q$, shown in Figure 26, corresponds exactly to the concurrency flowgraph built applying the algebraic composition operator to the flowgraphs P and Q as shown in Figure 21.

The net reduced by hiding action b and $\overline{b}$ is shown in Figure 27. The reachability graph corresponds exactly to the reduced flowgraph of Figure 21.

It should be noticed that the reachability graph of the reduced net can be obtained from the reachability graph of the whole net by deleting the arcs corresponding to actions b and $\overline{b}$, as expected from the properties of morphisms and products.

7 Conclusions

This paper compares the most used approaches to reachability analysis of concurrent programs: flowgraph and Petri net based approaches. It first compares the approaches from a "classical" point of view and it concludes that there is no essential semantic difference between the considered approaches. It shows that the differences of the various approaches are not in the way the state space is built, but in the way concurrent programs are represented. Any choice of approach must depend on other factors, like convenience and efficiency. A key factor for the reduction of the state space is the amount of detail of the concurrent programs that are taken into account.

This paper also compares the two approaches with respect to their suitability for supporting incremental and compositional analysis. It shows how the results

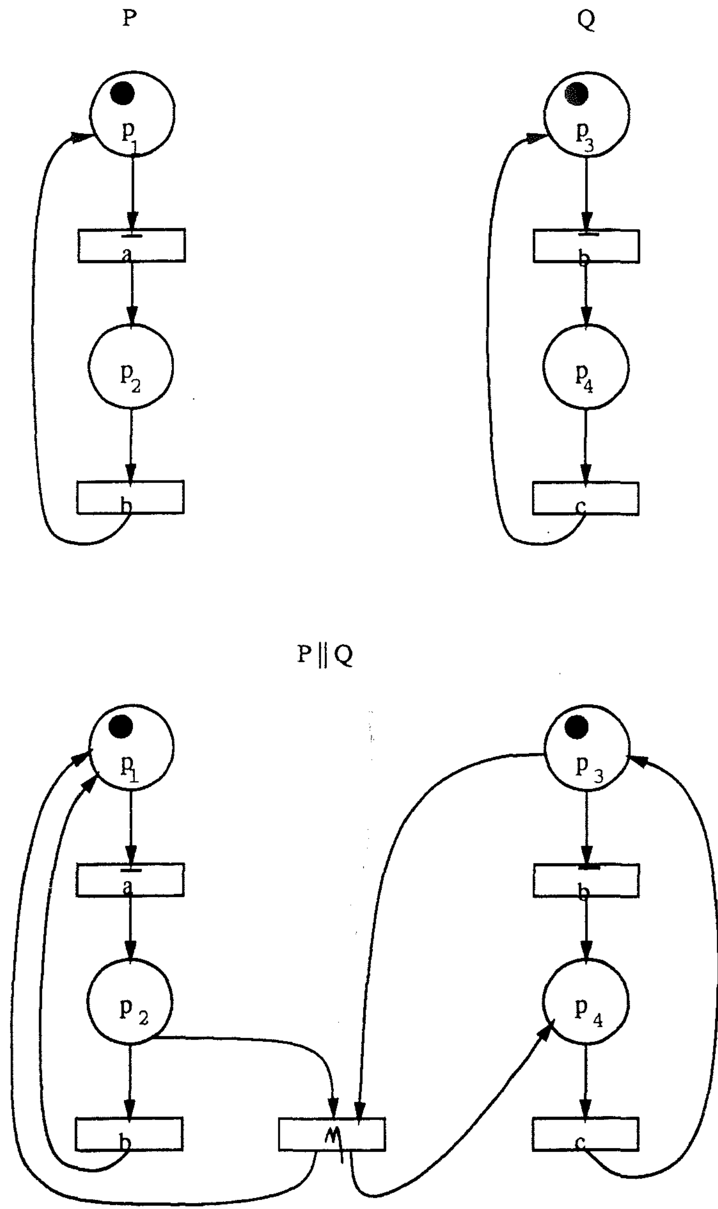
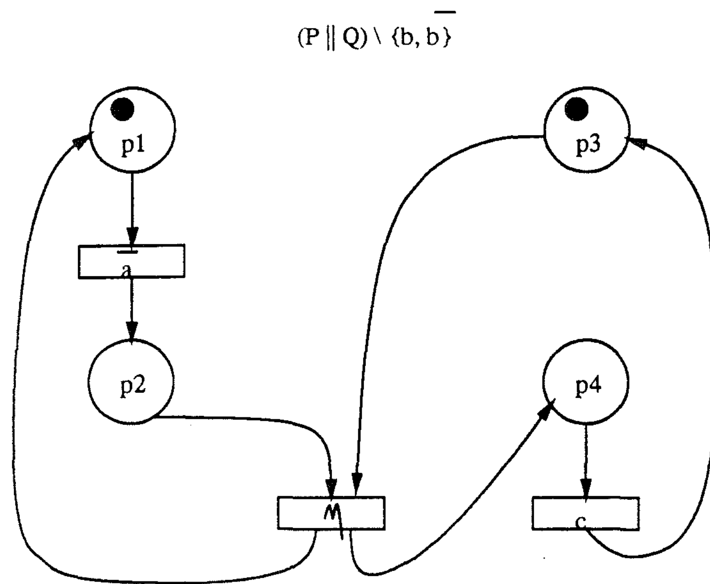


Figure 25: The Petri nets corresponding to two one position buffers and to their composition.



Reachability grapf of (P || Q) \ {b, b̄}

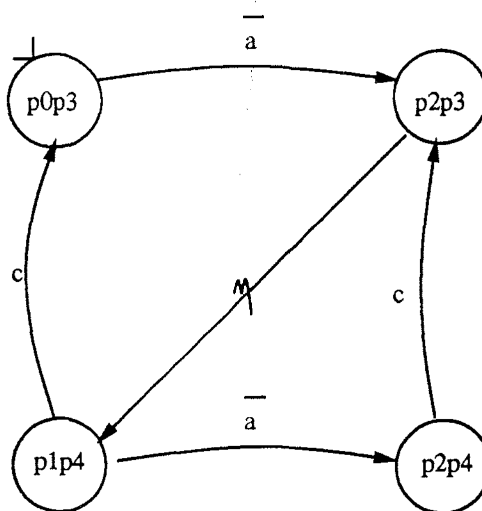
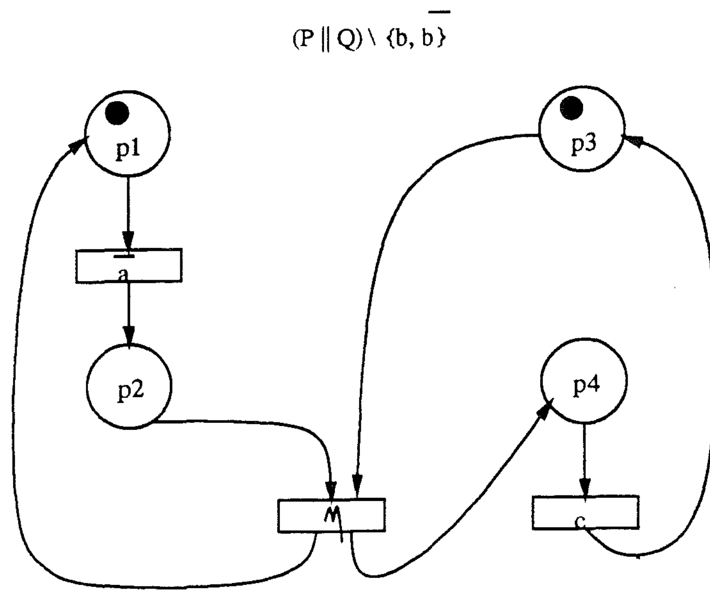


Figure 26: The reachability set of the net $P||Q$ of Figure 25.



Reachability graph of (P || Q) \ {b, b̄}

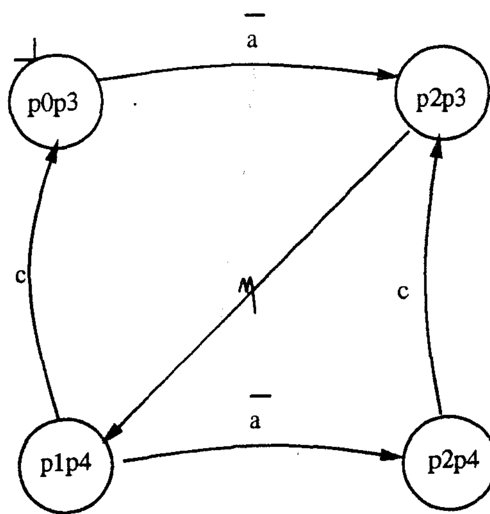


Figure 27: Reduction of the net of Figure 25 and corresponding reachability set.

obtained in [YY91] for flowgraph-based approaches can be obtained for Petri net based approaches as well. It thus shows that the various approaches to reachability analysis are equally amenable to techniques for reducing the complexity of the analysis.

We believe that this paper represents an important step in selecting reachability analysis techniques for large complex practical problems by discussing key factors that can drive the choice of the model to be used. This paper also describes key techniques for extending reachability analysis for tackling complexity problems for a wide class of systems, regardless the choice of the model to support the analysis.

References

[Apt83] Krzysztof R. Apt. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic*, Pittsburgh, PA, June 1983.

[BDER79] G. Bristow, C. Drey, B. Edwards, and W. Riddle. Anomaly detection in concurrent programs. In *Proceedings of the Fourth International Conference on Software Engineering*, pages 265–273, Munich, Germany, 1979.

[BHR84] S. D. Brookes, C.A. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[BR89] S. D. Brookes and A. W. Roscoe. Deadlock analysis in networks of communicating processes. Technical report, Department of Computer Science, Carnegie-Mellon University, 1989. (An earlier version appeared in Logics and Models of Concurrent Systems Springer Verlag 1985).

[BvG87] J. C. M. Baeten and R. J. van Glabeek. Another look at abstraction in process algebra. In *Proceedings of the 14th InternationalColloqium on Automata, Languages, and Programming (ICALP)*, pages 84–94, Karlsruhe, Germany, July 1987.

[CDK85] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto. Requirements-based testing of real-time systems: Modeling for testability. *IEEE Computer*, pages 71–80, April 1985.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CPS90] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. A semantics-based verification tool for finite-state systems. In *Proto-*

col Specification, Testing, and Verification, IX, pages 287–302. North-Holland, 1990.

[CPS91] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Machines*, pages 24–37, February 1991. LNCS 407.

[Dila] Laura Dillon. Verification of Ada tasking programs using symbolic execution; Part 1: partial correctness. Draft.

[Dilb] Laura Dillon. Verification of Ada tasking programs using symbolic execution; Part 2: general safety properties. Draft.

[Fer88] Jean-Claude Fernandez. *Aldébaran: Un Système de Vérification par Réduction de Processus Communicants.* PhD thesis, Université de Grenoble, Grenoble, France, 1988.

[Ger84] Steven M. German. Monitoring of deadlock and blocking in Ada tasking. *IEEE Transactions on Software Engineering*, 10(6), November 1984.

[GMMP89] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzè. Symbolic execution of concurrent systems using Petri nets. *Computer Languages*, 14(4):263–281, 1989.

[Han73] Per Brinch Hansen. Testing a multiprogramming system. *Software — Practice & Experience*, 3:145–150, 1973.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Founddations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[HK88] Linda J. Harrison and Richard A. Kemmerer. An interleaving symbolic execution approach for the formal verification of Ada programs zith tasking. In *Proceedings of the 3rd International IEEE Conference on Ada Applications and Environments*, Manchester, NH, 1988.

[Hoa85] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[Hol87] Gerard J. Holzmann. Automated protocol validation in *argos*: Assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, SE-13(6):683–696, June 1987.

[Lad79] Richard E. Ladner. The complexity of problems in systems of communicating sequential processes. In *Proceedings of the Eleventh Annual*

ACM Symposium on Theory of Computing, pages 214–223, Atlanta, Georgia, April 1979.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh, May 1989.

[McD89] Charles E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6:515–536, 1989.

[ME69] Karp R. M. and Miller R. E. Parallel program schemata. *Journal of Computer and System Science*, May 1969.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[MM90] Jose Meseguer and Ugo Montanari. Petri Nets are monoids. *Information and Computation*, 88:105–155, 1990.

[MP89] Sandro Morasca and Mauro Pezzè. Validation of concurrent Ada programs using symbolic execution. In *ESEC '89: 2nd European Software Engineering Conference*, pages 469–486. Springer-Verlag, September 1989. Lecture Notes in Computer Science.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE–13(10):1080–1091, October 1987.

[MSGS90] Jawahar Malhotra, Scott A. Smolka, Alessandro Giacalone, and Robert Shapiro. Winston — a tool for hierarchical design and simulation of concurrent systems. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, pages 140–152. Springer-Verlag, 1990.

[MZGT85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, 10(1):43–61, 1985.

[Rei85] Wolfgang Reisig. *Petri Nets.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[SC88] S. M. Shatz and W. K. Cheng. A Petri net framework for automated static analysis of Ada tasking behavior. *Journal of Systems and Software*, 1988. To appear.

[SMBT90] Sol M. Shatz, Khanh Mai, Christopher Black, and Shengru Tu. Design and implementation of a petri net based toolkit for ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[Smo84] Scott A. Smolka. *Analysis of Communicating Finite State Processes*. PhD thesis, Department of Computer Science, Brown University, 1984. Department of Computer Science Technical Report No. CS-84-05.

[SMS86] B. Shenker, T. Murata, and S. M. Shatz. Use of Petri net invariants to detect static deadlocks in Ada programs. In *Proceedings of the Fall Joint Computer Conference*, pages 1072–1081, November 1986.

[Sun81] Carl A. Sunshine, editor. *Communication Protocol Modeling*. Artech House, Dedham, MA, 1981.

[Tai85] K. C. Tai. Reproducible testing of concurrent Ada programs. In *Proceedings of SoftFair II*, pages 49–56, December 1985.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TKO91] K.C. Tai, R.H. Karver, and E.E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.

[TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, 1980.

[Win84] Glynn Winskel. A new definition of morphism on petri nets. In *STACS 84, Symposium of Theoretical Aspects of Computer Science*, pages 140–150, Paris, FRA, April 1984. *LNCS 166*.

[Win87] Glynn Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.

[Wol86] Pierre Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the ACM Symposium on Principles of Programming Languages (13th)*, pages 184–193, St. Petersburg, Fla., January 1986.

[YT88] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[SMBT90] Sol M. Shatz, Khanh Mai, Christopher Black, and Shengru Tu. Design and implementation of a petri net based toolkit for ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):424–441, October 1990.

[Smo84] Scott A. Smolka. *Analysis of Communicating Finite State Processes.* PhD thesis, Department of Computer Science, Brown University, 1984. Department of Computer Science Technical Report No. CS-84-05.

[SMS86] B. Shenker, T. Murata, and S. M. Shatz. Use of Petri net invariants to detect static deadlocks in Ada programs. In *Proceedings of the Fall Joint Computer Conference*, pages 1072–1081, November 1986.

[Sun81] Carl A. Sunshine, editor. *Communication Protocol Modeling.* Artech House, Dedham, MA, 1981.

[Tai85] K. C. Tai. Reproducible testing of concurrent Ada programs. In *Proceedings of SoftFair II*, pages 49–56, December 1985.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TKO91] K.C. Tai, R.H. Karver, and E.E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.

[TO80] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, 1980.

[Win84] Glynn Winskel. A new definition of morphism on petri nets. In *STACS 84, Symposium of Theoretical Aspects of Computer Science*, pages 140–150, Paris, FRA, April 1984. *LNCS 166*.

[Win87] Glynn Winskel. Petri nets, algebras, morphisms, and compositionality. *Information and Computation*, 72:197–238, 1987.

[Wol86] Pierre Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the ACM Symposium on Principles of Programming Languages (13th)*, pages 184–193, St. Petersburg, Fla., January 1986.

[YT88] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[YY91] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *4th Workshop on Testing and Verifications*, pages 49–59, Victoria, Canada, October 1991. ACM Sigsoft, ACM Press.