AN APPLICATION OF PETRI NET REDUCTION FOR ADA TASKING DEADLOCK ANALYSIS

S. M. Shatz, *Senior Member, IEEE*, S. Tu, *Member, IEEE*,
T. Murata, *Fellow, IEEE*, and S. Duri

Concurrent Software Systems Laboratory
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607

ABSTRACT

As part of our continuing research on using Petri nets to support automated analysis of Ada tasking behavior, we have investigated the application of Petri net reduction for deadlock analysis. Although reachability analysis is an important method to detect deadlocks, it is in general inefficient or even intractable. Net reduction can aid the analysis by reducing the size of the net while preserving relevant properties. We introduce a number of reduction rules and show how they can be applied to Ada nets, which are automatically generated Petri net models of Ada tasking. We define a reduction process and a method by which a useful description of a detected deadlock state can be obtained from the reduced net's information. A reduction tool and experimental results from applying the reduction process are discussed.

1. INTRODUCTION

One major difficulty that faces developers of concurrent and distributed software is analysis for concurrency-based faults like deadlocks. Techniques for such analysis are generally limited in their applicability due to the high complexity of most interesting analysis problems. For example, Reif and Smolka [1] prove some undecidable and NP-completeness results for reachability problems of various models of communicating processes. Rauchle and Toueg [2] provide a PSPACE-hardness result for deadlock detection in a bounded communication channel model for communicating finite state machines. With regards to concurrency analysis for models based on Ada tasking, Taylor [3] proved NP-Completeness results related to the classical (static) deadlock detection problem.

This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-90-J-1446 and the National Science Foundation (NSF) under grant numbers CCR-8913580 and CCR-9321743.

An earlier version of this paper has been published in the Proceedings of the 10th International Conference on Distributed Computing Systems.

Despite the difficulty of the concurrency analysis problem, a number of techniques have been proposed, especially for static analysis of Ada tasking [4-16]. We previously defined a Petri net framework for this type of analysis [6] and developed a toolkit that supports this approach [10]. The architecture of the toolkit, called TOTAL, is shown in Fig. 1.1. The FETS (Front-End Translator Subsystem) translates Ada source or Ada Tasking Language (ATL) design descriptions [10] into Petri net format, i.e., a set of appropriately labeled place nodes, transition nodes and arcs. We refer to the resulting Petri net as an (original) Ada net. These nets are suitable only for analysis of programs composed of a static set of tasks. The BIDS (Back-end Information Display Subsystem) is defined to receive users' queries and present tasking analysis results based on analysis of the Ada net [11]. BIDS uses different support tools to carry out the desired analysis. For example, deadlock detection using the method described in [9] would invoke the tool to calculate net invariants. For the approach that is the subject of this paper, BIDS first invokes the net reduction tool and then uses a reachability graph generator tool. Unlike pure static analysis methods, which ignore all variables, our approach can model boolean variables, which can increase the accuracy of the analysis. General implications of using static analysis are discussed elsewhere [4,10]. A few other Petri net-based tools aimed at Ada analysis have been proposed, including one developed by the Software Productivity Consortium (SPC); a primary concern of that work is performance modeling [12].

Taylor's "concurrency history" method [4] was one of the earliest Ada tasking analysis methods. As discussed in [6], his definition of a concurrency graph is very similar to a classical reachability graph for a Petri net. The goal of Taylor's paper was to propose a systematic method for generating a program's concurrency history for "general analysis"; it was not concerned with methods for improving analysis effectiveness for any specific type of analysis (like deadlock, our concern here). Yet, Taylor's work also simplifies the model by focusing on synchronization and concurrency issues. Long and Clarke [5] introduced a "task interaction graph" (TIG) representation for creating concurrency graphs that are smaller than those created by Taylor's method. We provide some experimentally based comparison of TIG-based results to our reduction-based results in Section 8. The *goal* of the TIG work is similar to ours, although we feel that our method has an advantage of being based on a model that is both theoretically mature (and continues to be widely and actively studied) and is supported by many available tools. Also, our approach is distinguished by the methodology of first deriving a semantically rich model independent of any specific analysis issue, and then manipulating this model with algorithms that are designed for the specific analysis issue of concern. Recently, Dwyer, et al. presented a compact Petri net representation for concurrent programs that is based on the TIG model [17]. Standard deadlock preserving Petri net reduction techniques cannot be applied to these nets, but some forms of deadlock preserving reductions that are applicable to these nets are suggested.

Some analysis methods that have shown particular promise in handling reasonably large examples include those that use binary decision diagrams (BDDs) to represent symbolically a state space [18,19], and those that use some special Petri net concepts such as stubborn sets of Petri net transitions [20], partial orders of Petri net transitions [21], and net symmetry [22]. Some BDD-methods have been very successfully applied to hardware system analysis, but it is not yet clear how directly suitable the method is for concurrent software analysis. The other methods mentioned are well defined for Petri nets, but do not exploit any particular type of concurrency, such as that defined by Ada's tasking model. The work of [23] explores the use of these methods with Ada nets.

In this paper, we consider analysis for the classical case of global deadlock detection, i.e., all tasks being blocked. In the context of the Petri net approach, a straightforward way to detect deadlocks would be to generate the Ada net's reachability graph and then search the reachability graph for states that have no successor states. This is very easy to implement with our toolkit system [10], but is computationally expensive since the complexity for generating a reachability graph for a (safe) net is exponential in the number of nodes [24]. Previously, we studied the use of Petri net structural invariants to aid deadlock detection in Ada tasking [9]. In keeping with this research technique, i.e., seeking to exploit and adapt existing general Petri net theory, in this paper we consider how to reduce the computation of deadlock analysis by use of another Petri net technique, net reduction. This is a model reduction method. Our experimental results indicate that this technique is quite powerful and can significantly extend the range of problem sizes that can be handled. We have found that our reduction method compares very favorably with other automated tasking analysis methods, especially those that are also rooted in explicit reachability analysis. In addition, we have found that our model reduction method facilitates the use of some state reduction methods -- the combination of model reduction followed by state reduction allows problems with more than 100 tasks to be analyzed. Our experimental results with such problems fare very well in comparison to the results reported for both the constrained expressions tool [8,16] and the PAL tool (which is based on using process algebra and compositional analysis) [15], which are the only other automated methods we know of that have been shown to be able to analyze an Ada program of more than 100 tasks.

Early work on reduction of graph models expressing concurrency was reported by Gostelow et al. [25]. Their reductions were based on a narrow set of substitutions in arbitrary graph models. A more comprehensive set of Petri net specific reduction rules was later presented by Berthelot [26]. Our work adapts some of Berthelot's rules to the specific modeling domain of Ada tasking and casts the rules into a technique and tool for the specific software analysis problem of deadlock analysis.

2. BACKGROUND: PETRI NETS, ADA NETS AND NET REDUCTION

For this paper, we assume that the reader has a basic understanding of Petri nets [24,27] and general knowledge of Ada tasking [28]. This section provides background in three areas: Some terminology and notations from Petri net theory are presented; some terminology and properties of Ada nets are introduced; and the idea of Petri net reduction is explained.

Definition 2.1

A (general) *Petri net* is a 5-tuple, $PN=(P, T, F, W, M_0)$, where P is a finite set of place nodes, T is a finite set of transition nodes, F is a set of arcs (flows) defined by $F \subseteq P\times T \cup T\times P$, W is a weight function defined as W: $F \rightarrow N$, and M_0 is the initial marking (state) defined by $M_0:P\rightarrow IN$, where N stands for the set of positive integers and IN for the set of the non-negative integers. A Petri net that is restricted by having W: $F \rightarrow \{1\}$ (i.e., arc weights are always equal to one) is called an *ordinary Petri net*. []

Definition 2.2

For a Petri net, PN, $x^{\bullet} =\{y|\ (x,y)\in F\}$ and ${}^{\bullet}x=\{y|\ (y,x)\in F\}$, $x,y \in P\cup T$. If x is a place (transition), ${}^{\bullet}x$ is the *input transition (place) set* of x and $x^{\bullet}$ is the *output transition (place) set* of x. []

Definition 2.3

For a Petri net, PN, a *marking (state)* M is a mapping M: $P\rightarrow IN$. A marking (state) M is graphically represented by an assignment of tokens (shown as small black dots) to places. The notation $M(p)$ denotes the number of tokens in place p at marking M. []

Definition 2.4

Given a marking M of a Petri net, PN, for a transition $t\in T$, if $\forall p\in {}^{\bullet}t$, $M(p) \geq W(p,t)$, t is said to be *enabled*. Note that for an ordinary net, the transition is enabled if $\forall p\in {}^{\bullet}t$, $M(p) > 0$. []

Definition 2.5

For a Petri net, PN, a transition t that is enabled in marking M can *fire*, resulting in a new marking M' where 1) $\forall p\in {}^{\bullet}t - t^{\bullet}$, $M'(p) = M(p) - W(p,t)$; and 2) $\forall q\in t^{\bullet} - {}^{\bullet}t$, $M'(q) = M(q) + W(t,q)$. Note that for an ordinary net, the firing of the transition results in a new marking M' where 1) $\forall p \in {}^{\bullet}t - t^{\bullet}$, $M'(p) = M(p) - 1$; and 2) $\forall q \in t^{\bullet} - {}^{\bullet}t$, $M'(q) = M(q) + 1$. []

Definition 2.6

For a Petri net, *PN*, a marking M is said to be *reachable from a marking* M' if there exists a sequence of markings M_j, M_{j+1}, ..., M_{j+k}, where $M_j=M$, $M_{j+k}=M'$, and M_{j+i} results from firing some enabled transition in marking M_{j+i-1}, for i=1,2, ... k. A sequence of transition firings that leads from some marking M to a marking M' is called a *transition firing sequence*. []

Definition 2.7

For a Petri net, *PN*, the set of *reachable markings* is denoted by $R(PN,M_0)$,

$R(PN,M_0)=\{M \mid M \text{ is a marking reachable from } M_0\}$.

Definition 2.8

A Petri net, *PN*, is said to be *safe* if the number of tokens in every place does not exceed one for any reachable marking, i.e., $\forall M \in R(PN,M_0)$ and $\forall p \in P$, $M(p) \leq 1$. []

Definition 2.9

A Petri net, *PN*, can *deadlock* if there exists a reachable marking in which no transition is enabled. []

A Petri net that is derived from an Ada tasking program by the translation scheme of [6,10] is called an *original Ada net*. An original Ada net models a number of communicating finite state machines. The communication is based on Ada's notion of rendezvous. Each state machine represents the local control flow of a task. The intertask communications are represented by extra nodes connecting different state machines. Control flow constructs such as if-else, case, and loop are modeled by structures in the state machines if the control flow construct can impact the choice of a tasking construct like an entry call or accept statement. The translation algorithm that produces original Ada nets uses a set of Petri net templates corresponding to Ada statements. Details of this translation are given in [6]. Here we simply describe some of the key features of Ada nets that are germane to our upcoming discussion on reduction.

In an original Ada net, each place is automatically labeled by the translation program and has a well-defined role in modeling the program's behavior. Each place label is prefixed with a keyword indicating the type of the place and a label also contains a statement number. For example, the template of an entry call statement is shown in Fig. 2.1(a) and that of an accept statement with an accept body is shown in Fig. 2.1(b). Accept statements without accept bodies are modeled in a more compact way as can be seen in the example of Fig. 2.2. Generally, a rendezvous is represented with the net structure shown in Fig. 2.1(c). The model does not directly synchronize the "entry" place and the "accept" place for the following reason. Because the model is created automatically during translation of source code, we chose to have an entry call construct that models the making of an entry call and a separate accept construct that models accepting a call that has been made. This is convenient for the translation algorithm and also provides a model that can distinguish a task having made a call that is not yet accepted, from a task that is ready to make a call. We explain this a bit more after the upcoming example.

Definition 2.10

In an Ada net, we call a place having a label prefixed with the string "begin-" "wait-", "accept-", "entry_ex-" or "select-", a *begin-place*, *wait-place*, *accept-place*, *entry-ex-place* or *select-place*, respectively. []

By their method of construction, original Ada nets have the following characteristics [6,10]:

1) Original Ada nets are ordinary and safe Petri nets.

2) Original Ada nets have at most linear number of nodes with respect to the statements in the corresponding Ada program.

3) The only places that contain a token in the initial marking are those that represent the beginning of a task's control flow; these are the begin-places defined above.
4) Decision statements, such as if-else and case statements in local control flow, are represented by a branching net structure (i.e., a place has multiple output arcs).
5) In every branching net structure, at least one branch (choice) contains a net structure for some communication statement. (The translation algorithm ignores decision statement that involve no intertask communication.)

To give the reader a feel for the labeling convention used in Ada nets and a sense of net reduction, we use an example program called the gas station program [29]. The statement numbers are not a part of the program itself, but are used for reference.

Example 2.1 (The gas station program)

```
1    task body Customer is
2    begin
3    loop
4       Operator.Prepay;
5       Pump.Start;
6       Pump.Finish;
7       accept Change;
8    end loop;
9    end Customer;
```

```
10   task body Pump is
11   begin
12   loop
13         accept Activate;
14         accept Start;
15         accept Finish do
16            Operator.Charge;
17         end Finish;
18   end loop;
19   end Pump;
```

```
20   task body Operator is
21   begin
22   loop
23   select
24         accept Prepay do
25            Pump.Activate;
26         end Prepay;
27   or
28         accept Charge do
29            Customer.Change;
30         end Charge;
31   end select;
32   end loop;
33   end Operator;
```

The original Ada net structure corresponding to the above 3-task program, is shown in standard Petri net graphical form in Fig. 2.2. For purpose of later reference, the token distribution shown in Figure 2.2 indicates a deadlock marking rather than the initial marking of this Ada net. We give the following interpretations of a few place labels appearing in the Ada net shown in Fig. 2.2. Note that place label ack_entry_Operator_4 is abbreviated as aeO4 in the figure.

ack_entry_Operator_4 -- An entry call (at line 4) has been made to, but not yet accepted by, the Operator task

wait_ack_Operator_4 – The entry call at line 4 has been executed and the calling task is waiting for the acknowledgment from the Operator task (the call may or may not have been accepted yet)

entry_ex_15_6 – A rendezvous is in progress (the entry is executing) for the entry call at line 6 and the accept statement at line 15

[]

Original Ada nets are intended to serve as a base model for general tasking analysis (as discussed in [10]), not just deadlock analysis. It can be observed that original Ada nets contain atomic steps (transitions) that are not strictly necessary to model tasking semantics, (for example end_loop transitions), but do aid the translation process. Part of the effect of the reduction discussed in this paper is just to remove these "extra" steps. On the other hand, there are also some net structures (for example, some places) that are unnecessary for deadlock analysis but may be supportive of other forms of analysis. For example, a wait-place being marked indicates that a task has made an entry call and is waiting for a response. In a distributed environment, this may correspond to states where a request message is being transmitted, or states where the remote task is performing the service, or states where an acknowledgment message is being sent to the calling task. So, to determine the maximum number of entry calls that can be "waiting" simultaneously, we can search for a state that contains the maximum number of marked wait-places.

Petri net researchers have developed various reduction techniques for general Petri net analysis [26]. Petri net reduction aims to transform a given Petri net, *PN*, into another Petri net, *PN'*, such that *PN'* is easier to analyze algorithmically (typically this means that *PN'* has a smaller reachability graph than *PN*). To illustrate this, we show in Fig. 2.3 a net that is produced by applying our net reduction rules to the net of Fig. 2.2. It is important to observe that the reduced net does not explicitly model all the task interactions defined by the source program (as does the Ada net in Fig. 2.2). For instance, the source program specifies 6 entry calls and 6 accept statements, yet the reduced net contains only 4 transitions. But, by definition of the reduction rules that we will introduce, it is true that the reduced net does preserve the deadlock characteristics of the original Ada net.

Definition 2.11

A net that is derived from an original Ada net by some number of reduction steps using the reduction rules presented in this paper is called a *reduced Ada net*. A reduced Ada net is called a *fully reduced Ada net* if no reduction rule can be applied to it. An *Ada net* can be either an original Ada net, a reduced Ada net, or a fully reduced Ada net. []

A fully reduced Ada net is not unique; it depends on the interplay of the reduction operations that are applied.

3. SOME GENERAL REDUCTION RULES

For practical use on Ada nets, we selected from [26] some transition fusion reduction rules that are applicable to general Petri nets. Our interest is in the fact that these rules preserve the deadlock properties of a net, i.e., any deadlock state is preserved and no new deadlock state is introduced. Furthermore, these rules preserve the safeness of a net. Rigorous proofs can be found in [26]. We did not use all of the rules proposed in [26] because some of them are of high computational complexity, i.e., it is computationally expensive to check the applicability of those rules. In this section we also introduce a simple rule from [27] that allows removal of a place or transition in cases where there exist "parallel redundant" nodes. In the next section we will derive some other reduction rules that are special case rules useful for Ada nets. Note that whenever a reduction rule calls for the removal of a node (a place or a transition), it is implied that the incident arcs of the node are also removed.

Rule 1 (Post-Fusion of Transitions) [26]

A non-empty subset G of T, the set of transitions in a Petri net, is post-fusable with another subset H of T iff there exists a place p such that the following four conditions are satisfied:

(a) $\forall f \in G, {}^\bullet f = \{p\}$ -- the only input of f is p,
$p \notin f^\bullet$ -- p is not an output of f.

(b) $\forall h \in H,\ p \in h^\bullet$ and $p \notin {}^\bullet h$ -- p is an output, but not an input, of h.

(c) $\forall t \in T\text{-}(H \cup G),\ p \notin {}^\bullet t$ and $p \notin t^\bullet$ -- p is disconnected from other transitions except for those belonging to H or G.

(d) $M_0(p)=0$.

The operation of Rule 1 is to fuse each $h \in H$ with each $f \in G$, by producing a transition $<hf>$ such that $^{\bullet}<hf>=^{\bullet}h$ and $<hf>^{\bullet} = f^{\bullet} \cup (h^{\bullet} - \{p\})$, and then deleting p and all the transitions $t \in (H \cup G)$. []

Conditions (a) and (b) imply that $H \cap G = \emptyset$. Fig. 3.1 shows the Post-Fusion of transitions h_1 and h_2 with f_1 and f_2. In the reduced net, if we consider the firing of every new transition, $<hf>$, as a firing sequence in which both h and f fire once, the number of firings of each transition in the reduced net is the same as that of the original net.

Rule 2 (Pre-Fusion of Transitions) [26]

A subset G of T, the set of transitions in a Petri net, may be pre-fused with a transition $h \in T$ iff there exists a place $p \in P$ such that the following conditions are satisfied:

(a) $h^{\bullet} = \{p\}$ and $p \notin {}^{\bullet}h$ -- p is the sole output of h and p is not an input to h.

(b) $\forall f \in G$, $p \in {}^{\bullet}f$ and $p \notin f^{\bullet}$ -- all transitions of G have p as an input, but none has p as an output.

(c) $\forall t \notin (G \cup \{h\})$, $p \notin {}^{\bullet}t$ and $p \notin t^{\bullet}$ -- p is disconnected from all other transitions.

(d) $\forall q \in {}^{\bullet}h$, $|q^{\bullet}|=1$ -- h does not share its input places with other transitions.

(e) $M_0(p)=0$.

The operation of Rule 2 is to fuse h with each $f \in G$, by substituting $^{\bullet}f$ with $(^{\bullet}f \cup {}^{\bullet}h)-\{p\}$ and then deleting h and p. []

Fig. 3.2 shows transition h fusing with f_1 and f_2. The properties preserved through pre-fusion are the same as for Rule 1, post-fusion (proof in [26]). By applying Rule 2 to a place p, we will reduce the number of transitions in the net by one and the number of places by one.

The third reduction rule is a special case of Rule 1. We explicitly specify it because it is easier to apply and commonly occurring. Fig. 3.3 illustrates the operation of Rule 3.

Rule 3 (Serial-Fusion of Transitions)

If a single-input/output place p has no token and is the unique output place of t_1 and the unique input place of t_2,, i.e., if $M_0(p)=0$, $t_1^{\bullet} = \{p\} = {}^{\bullet}t_2$,, and $|^{\bullet}p|=|p^{\bullet}|=1$, then t_1 and t_2 can be fused into $<t_1t_2>$ such that $^{\bullet}<t_1t_2>={}^{\bullet}t_1$ and $<t_1t_2>^{\bullet}=t_2^{\bullet}$. []

Rule 4 (Parallel Redundant Nodes) [27]

If two nodes (two transitions, or two places without tokens), x and y, have identical inputs and identical outputs (possibly empty), then they are a pair of parallel nodes, and either one of these nodes can be removed. That is, if $\bullet x = \bullet y$ and $x\bullet = y\bullet$, then either x or y (but not both) can be removed. []

It is obvious that Rule 4 preserves deadlock and safeness properties of a net.

4. SPECIAL CASE REDUCTIONS FOR ADA NETS

Intuitively, a place in a Petri net can be considered to be *redundant* if its removal will not alter any possible transition firing sequences of the net. Throughout this paper, removal of a redundant place p means that p and all of p's incident arcs are removed. A benefit of identifying such places is that their removal can generally facilitate further reductions such as those that involve transition fusions. Clearly the safeness and deadlock properties will be preserved by the removal of redundant places. Unfortunately, in general, there is no efficient algorithm for identifying that a place is redundant. Berthelot gave a formal structure-based characterization of a redundant place for general Petri nets [26], which is based on the existence of a place subset (called the *I-set*) that meets certain conditions. Our interest is in applying this concept to Ada nets, so we seek some efficient rules that are special case instances of Berthelot's characterization.

We will begin by introducing a general characterization for a specific case of a redundant place in a safe and ordinary Petri net; but first we want to establish the fact that Ada nets are always safe and ordinary nets. Recall that original Ada nets are safe and ordinary nets by construction. As we noted earlier, the reduction rules introduced in Section 3 preserve safeness. As established by Berthelot [26], Rule 1 always keeps an ordinary net ordinary if the net is safe to start with. It is not difficult to see that Rules 2, 3 and 4 will keep an ordinary net ordinary. Finally, the new rules that we will introduce in this section only remove nodes; so they cannot change an ordinary net into one that is not ordinary. We will prove that every node to be removed by the new rules is redundant in the sense that the removal does not alter any possible transition firing sequence. So, the safeness property of the nets will be preserved. Therefore, it can be concluded that Ada nets are always safe and ordinary nets. This is important since we want to appeal to a general characterization of a redundant place in a safe and ordinary net to prove some special case instances of redundant places in Ada nets.

Theorem 4.1 (Case 1 of Place Redundancy in a Safe and Ordinary Net)

For a safe and ordinary Petri net, *PN*, place p is redundant if the following conditions are satisfied: 1) $p^{\bullet}=\{t\}$; 2) $M_0(p)=0$; and 3) there exists some $b\in {}^{\bullet}t$, where $b\neq p$ and $\forall M\in \mathrm{R}(PN,M_0)$, if $M(b)=1$ then $M(p)=1$, i.e., whenever b is marked, p is also marked. []

Proof

Since t is the only output transition of p, p cannot impact the firing of any transition except t. One of the necessary conditions for t to be enabled is that b is marked. Condition 3 establishes that whenever b is marked, p is also marked. So, the removal of p will not impact the firing of t in any reachable marking, meaning that the place p is redundant. []

While the above theorem does help define a narrower class of redundant places in comparison to the general characterization given by Berthelot, it is computationally expensive to use this theorem to identify a redundant place due to the behavioral condition implied by Condition 3. But, we can appeal to this theorem to define some special case instances of redundant places in Ada nets. Remember, in an original Ada net each place is automatically labeled by the translation program and has a well-defined role in terms of modeling the program's behavior. So, the semantics of the net must correspond to the semantics of an Ada program, if the net is a sound model -- which we can assume is the case for this paper. We can exploit this sense of semantics to identify cases where Condition 3 of the above theorem must be true. Thus, for these cases, we will have simple and efficient reduction rules. Since our goal is to use these rules to define a reduction process, we will explain how and why we select some ordering dependencies for the rules.

Rule 5 (Redundancy of Wait-Places)

In an original Ada net structure corresponding to an entry call statement, the wait-place is redundant and can be removed. The same holds for nets obtained by applying this rule one or more times to an original Ada net. []

Proof

Fig. 2.1(a) shows the original Ada net structure for modeling an entry call. The ack_entry place represents a call that has been made, but not yet accepted. The ack_accept place represents an acknowledgment from the called entry to indicate that the rendezvous is terminating. The wait_ack place represents that the caller is waiting for the termination acknowledgment. Appealing to Theorem 4.1, we can see that Conditions 1 and 2 of Theorem 4.1 are true when the places *p* and *b* are represented by the places wait_ack and ack_accept, respectively. By Ada semantics, a rendezvous that is terminating must have been previously initiated by the caller. In the original Ada net structure of Fig. 2.1(a), it can easily be seen that once a rendezvous is initiated by a caller, the wait_ack place become marked and remains marked until the ack_accept place becomes marked. Therefore, Condition 3 of Theorem 4.1 is also true, and the wait-place is redundant. Since the removal of the wait-place associated with some entry call statement does not alter the Ada net structure of any other entry call statement, we can apply this reduction one or more times, i.e., to all wait-places. []

The importance of Rule 5, as will be the case for Rules 6 and 7, is that it provides an opportunity for further reductions to apply. To illustrate this, let us consider Example 4.1. Its corresponding Ada net is shown in Fig. 4.1(a).

Example 4.1

```
1   task body T1 is
2   begin
3       T2.E;
4   end T1;

5   task body T2 is
6   begin
7       accept E do
            ...
8       end E;
9   end T2;

10  task body T3 is
11  begin
12      T2.E;
13  end T3;
```

Applying Rule 5, we remove immediately wait_ack_T2_3 and wait_ack_T2_12. Then, applying Rule 3 (Serial-fusion) and Rule 2 (Pre-fusion) to t_1, t_2, t_3, t_4, t_9, t_{10}, t_{11} results in the net in Fig. 4.1(b). []

In our reduction process for deadlock detection, we always apply Rule 5 first, meaning that we remove all wait-places as the first step in the reduction process.

Rule 6 (Redundancy of Entry-Ex-Places)

In an original Ada net structure corresponding to an accept statement, the corresponding entry-ex-place is redundant and can be removed if this accept statement has at most one caller. The same holds for nets obtained by applying this rule one or more times to an original Ada net. []

Proof

Fig. 2.1(b) shows the original net structure for modeling an accept statement. The end_accept place represents that the accepting task has reached the end of the accept body and is ready to terminate the rendezvous initiated by some call. The entry_ex_j_i place represents that a particular rendezvous is in progress -- a rendezvous associated with a particular entry call statement *i* and a particular accept statement *j*. Appealing to Theorem 4.1, we can see that Conditions 1 and 2 of Theorem 4.1 are true when the places *p* and *b* are represented by the places entry_ex_j_i and end_accept, respectively. By Ada semantics, when a task reaches the end of an accept statement, there must be a corresponding rendezvous in progress. In the case of an accept statement with only one possible caller, there is only one corresponding entry-ex place (as shown in the case of Fig. 2.1(b), which is in contrast to the multiple caller case as shown by the example in Fig. 4.1(a)). Thus, whenever the end_accept place is marked, the entry_ex_j_i place will be marked. So, Condition 3 of Theorem 4.1 is true, and the entry-ex-place is redundant. Since removal of entry-ex places does not alter the net structures associated with other accept statements (recall the structures in Fig. 2), the removal of these places does not affect the application of Rule 6 to those structures. So, we can apply this reduction one or more times. []

It is clear that removal of a wait-place does not affect the validity of applying Rule 6 to a corresponding entry-ex place. So, in our reduction process, we always apply Rule 6 after all possible applications of Rule 5. We could have just as well chosen to define our reduction process to start with applications of Rule 6, followed by Rule 5 -- there is no important difference.

We now introduce a second theorem that is a generalization of Theorem 4.1. By appealing to this theorem, we can prove two more special case rules for application with Ada nets.

Theorem 4.2 (Case 2 of Place Redundancy in a Safe and Ordinary Net)

For a safe and ordinary Petri net, PN, place p is redundant if the following conditions are satisfied: 1) $M_0(p)=0$ and $|p^\bullet|=r$, and 2) for every transition $t_{Ok} \in p^\bullet$, $k=1, 2, \ldots,$ r, there is a place $q_{Ok} \neq p$, where $q_{Ok}{}^\bullet=\{t_{Ok}\}$ and $\forall M \in \mathrm{R}(PN,M_0)$, if $M(q_{Ok})=1$ for any $k \in \{1, 2, \ldots, r\}$, then $M(p)=1$, i.e., if any q_{Ok} place is marked in any state, then p is also marked in that same state. []

Fig. 4.2 contains an example net structure that corresponds to the structure defined by Theorem 4.2 (for the case when r=2). At this time, one can ignore the labels of the nodes not defined in Theorem 4.2.

Proof

Condition 2 and the safeness property of the net imply that at any reachable state, at most one of the q_{Ok} places can be marked. If more than one of these places could be marked then one of the associated enabled transitions could fire, resulting in a marking in which some q_{Ok} place is still marked but place p is not marked. This would contradict Condition 2. Now we can infer that at any reachable marking, p can only impact the firing of at most one transition, call it t. A necessary condition for t to be enabled is that the corresponding q_{Ok} is marked. Since Condition 2 establishes that whenever any q_{Ok} is marked, p is also marked, the removal of p will not impact the firing of t in any reachable marking, meaning that the place p is redundant. []

We now present two rules that appeal to Theorem 4.2 to identify some specific cases of redundant places that can occur in Ada nets. Both of the next two rules can be applied at any time during the reduction process since the rules do not depend on Ada net issues other than to help identify candidate places (by labeling) to check for applicability of the rule. This improves the efficiency of the reduction process. For example, Rule 7 directs its attention to accept-places. In particular, if an Ada net indicates that a task calls two consecutive entries in another task, then the net structure that enforces the called task's control flow can be simplified by the removal of one of the accept-places in the called task.

Rule 7 (Redundancy of Accept-Places)

In an Ada net, let p be an accept-place with $^\bullet p=\{t_{I1}, t_{I2} \ldots , t_{Ik}\}$, $p^\bullet=\{t_{O1}, t_{O2}, \ldots , t_{Ok}\}$, and $M_0(p)=0$. If there exists another accept-place q such that $q^\bullet={}^\bullet p$, and p and q correspond to two accept statements in the same task T, then place p can be removed if the following conditions are also satisfied:

1) For each transition $t_{Ik} \in {}^{\bullet}p$, $k \in \{1, 2, ..., r\}$, there exists a corresponding transition $t_{Ok} \in p^{\bullet}$, where $t_{Ik}^{\bullet}=\{p, q_{Ik}\}$ and ${}^{\bullet}t_{Ok}=\{p, q_{Ok}\}$, and each pair of places (q_{Ik}, q_{Ok}) correspond to statements of the same task, k, where $k \neq T$.
2) For each pair of transitions (t_{Ik}, t_{Ok}) defined in Condition 1), t_{Ik} always fires first, t_{Ik} cannot fire twice without a firing of t_{Ok}, and t_{Ok} cannot fire twice without a firing of t_{Ik}. []

Fig. 4.2 illustrates the net *structure* corresponding to Rule 7 for the case when r=2.

Proof

Without loss of generality, we can refer to Fig. 4.2, which illustrates the net structure corresponding to Rule 7 for the case when r=2. It can be seen easily that Condition 1 of Rule 7 establishes the structural conditions of Theorem 4.2. The behavioral condition specified in Condition 2 of Theorem 4.2 can be shown to be true as follows: If t_{Ik} is enabled (for some k) in some marking M, then $M(p)$=0, otherwise p may contain multiple tokens and the net would not be safe -- but we have previously established that Ada nets are safe nets. Thus, no transition t_{Ok} (for any k) can be enabled in marking M. Whenever the transition t_{Ik} does fire, p becomes marked. But, in this case before t_{Ok} fires, no transition t_{Ik} can fire; otherwise the net will again not be safe. Condition 2 further ensures that no transition other than t_{Ok} can fire. Thus, the token in place p will not be removed until transition t_{Ok} fires. Note that transition t_{Ok} cannot be enabled unless place q_{Ok} is marked. Therefore, $\forall M \in R(PN, M_0)$, if $M(q_{Ok})$=1 for any $k \in \{1, 2, ..., r\}$, then $M(p)$=1. The conditions of Theorem 4.2 hold. []

Example 4.2

Fig. 2.2 (the gas station example) can be used to illustrate a simple case of a redundant accept-place as defined by Rule 7. Using the notation of the Rule 7, let accept_15 correspond to the place p and let accept_14 correspond to the place q. By checking the figure, it can be seen that $q^{\bullet}={}^{\bullet}p$ and that r=1 for this example. The task T, as defined by the rule, is the task that contains the two accept statements in question --- this is the Pump task (as can be seen from the source code given in Section 2). Now, considering Condition 1, transition t14 corresponds to t_{I1} and transition t15 corresponds to t_{O1}, where task Customer corresponds to task 1. Checking Condition 2, we can see that ack_accept_Pump_5 $\in$ t14$^{\bullet}$ corresponds to q_{I1} and ack_entry_Pump_6 $\in$ $^{\bullet}$t15 corresponds to q_{O1}. From the Ada net model of the task Customer, it is easy to confirm that in every iteration transition t14 fires before t15 fires (implying that execution of Statement 5 must precede the execution of Statement 6); So, Condition 2 is true. Thus by Rule 7, accept_15 is a redundant place. []

To implement Rule 7 we need to automatically check both conditions of the rule. Condition 1 is not difficult to check since the correspondence between places in an Ada net and statements in the Ada program is well defined based on the type of place labeling that is used and was discussed earlier. Assuming now that Condition 1 is true, the main difficulty is in checking Condition 2. Without loss of generality, we consider only the checking of Condition 2 for one calling task (i.e., when r=1). Let us denote the statement that corresponds to the place q_{Ik} as *s1* and the statement that corresponds to place q_{Ok} as *s2*. By Condition 1, *s1* and *s2* belong to the same task, call it task k. As mentioned in Section 2, an Ada net contains subnets that are communicating finite state machines representing the local control flow of each task. A subnet consists of all the places that correspond to the statements in a particular task, the input and output transitions of these places, and the arcs associated with these place and transition nodes. Condition 2 can be checked by confirming the following two properties: a) In the subnet corresponding to task k, all (directed) paths starting from the begin-place of task k to the place q_{Ok} go through a place corresponding to the statement *s1*; and b) if a loop in the subnet corresponding to the task k contains the place q_{Ok}, then this loop must also contain a place corresponding to the statement *s1*. To confirm these properties, we temporarily remove every place that corresponds to statement *s1*. Then we perform two searches: 1) a search for a path starting from the begin-place of task k to place q_{Ok}, and 2) a search for a loop starting from q_{Ok} and leading back to q_{Ok}. If both searches fail, then Condition 2 is satisfied; otherwise, Condition 2 is not satisfied.

Rule 8 (Redundancy of Accept-Body Places)

For a safe and ordinary Petri net, *PN*, place p is redundant if the following conditions are satisfied: 1) $|{}^{\bullet}p|=|p^{\bullet}|=r$, and 2) for each transition $t_{Ik}\in {}^{\bullet}p$, k=1, 2,..., r, there exists a unique place $q_{Ok}\neq p$, where ${}^{\bullet}q_{Ok}=\{t_{Ik}\}$, $|q_{Ok}{}^{\bullet}|=1$, and for every transition $t_{Ok}\in p^{\bullet}$, k=1, 2,..., r, ${}^{\bullet}t_{Ok}=\{p, q_{Ok}\}$, and $M_0(p)=M_0(q_{Ok})=0$, k=1, 2,..., r. In Ada nets, a structure representing an accept-body that is modeled by a single place p and has multiple callers is a good candidate for this instance of redundancy. If the conditions hold, the place p is removed. []

Proof

Rule 8 is a special case of Theorem 4.2 in which ${}^{\bullet}q_{Ok}=\{t_{Ik}\}$. Clearly, the structural conditions for Theorem 4.2 are true. We establish Condition 2 of Theorem 4.2 as follows: i) Since ${}^{\bullet}q_{Ok}=\{t_{Ik}\}$ and $t_{Ik}\in {}^{\bullet}p$, when q_{Ok} gets marked, so does p. ii) Since $p^{\bullet}=\{t_{O1}, t_{O2}, ..., t_{Or}\}$, the token in p will not be removed until some transition t_{Ok} (k=1, 2, ..., r) fires. iii) At any marking M where $M(q_{Ok})=M(p)=1$, no transition t_{Ik}, k=1, 2, ..., r, can fire; otherwise p will have more than one token and the safeness property will not hold. Thus, only one q_{Ok} place can be marked in any reachable marking. iv) Because of ii) and iii), the token in p will not be removed until t_{Ok} fires, which will remove the token in p and q_{Ok} simultaneously. Therefore, $\forall M\in \mathrm{R}(PN,M_0)$, if $M(q_{Ok})=1$ for any $k\in\{1, 2, ..., r\}$, then $M(p)=1$. The conditions of Theorem 4.2 hold. []

Now let us return to Fig. 4.1(b). Rule 8 allows us to remove place end_accept_8. Then serial fusions of $<t_1\ t_2\ t_4\ t_5>$ with $<t_3\ t_7>$ and $<t_4\ t_6\ t_9\ t_{10}>$ with $<t_8\ t_{11}>$ lead to the further reduced Ada net of Fig. 4.3.

Our final rule is a simple rule that applies to marked places only and removes places and transitions under very restricted conditions.

Rule 9 (Redundancy of Begin-Places)

In a safe and ordinary net, if there exists a place $b\in P$, such that $M_0(b)=1$, $b^{\bullet}=\{t\}$, $|{}^{\bullet}t|=1$, and ${}^{\bullet}b=\emptyset$, then b and t can be removed and for all $d\in t^{\bullet}$, $M_0(d)$ is set to 1. In Ada nets, *begin-places* are initially marked and do not have input transitions. So, they are used as candidates for the place b. []

Proof

Marking every output place of t and then removing t is equivalent to firing t once. Since ${}^{\bullet}t=b$ and b is marked initially, t can fire initially. But since ${}^{\bullet}b=\emptyset$, t can only fire once in any transition firing sequence. So, for this case, removal of the nodes b and t and reassignment of the tokens as defined does not change the safeness or deadlock properties of the net. []

Note that it is possible for Rule 9 to reduce a net into an isolated marked place, which would represent a deadlock state. If different reduction rules are to be applied to a net, Rule 9 should be applied last since all the other rules (Rules 1 through 8) assume the places are unmarked, but Rule 9 can induce a marking. This marking could be lost if further reductions are allowed after Rule 9 is applied.

5. A REDUCTION PROCESS AND METHOD FOR DEADLOCK IDENTIFICATION

To apply the reduction rules discussed in the previous sections we define a reduction process for deadlock detection and identification for Ada nets. The process starts with an original Ada net and then applies the rules in a particular order. To preserve the soundness of the reduction process with respect to Rules 5 through 8, we follow a rule ordering that is consistent with that mentioned in Section 4. The ordering is defined as follows: we repeatedly apply rule 5 until it does not apply, then do the same in sequence for rules 6, 7, and 8; then the rules 3, 2, 1, and 4 are repetitively applied in that order until none of those four rules can be applied; and finally Rule 9 is applied.

Deadlock identification requires that we be able to interpret a program state corresponding to a detected deadlock marking in a reduced Ada net. The difficulty arises because the reduction process removes places whose labels are helpful to defining the semantics of a program state corresponding to a reachable marking. Fortunately, we can provide a method for easily constructing an original Ada net's deadlock marking from a reduced net's deadlock marking. The method is based on an application of the Petri net state equation [27],

$$M=A^{T}(V)+M_0 \tag{5.1}$$

where M_0 is the initial marking of the net, A is the incidence matrix [27] of the net, and V is a *firing count vector*. A firing count vector is an m-vector (m is the number of transitions in the net), each element of which is the number of times that the corresponding transition fired [27]. Given A, M_0, and V, Equation (5.1) returns the marking that the net will reach after the transitions represented by V fire. Our concern is with how to obtain V in terms of the original Ada net.

During reduction, the application of each transition fusion rule (Rule 1, 2 or 3) will remove some transitions and generate some new transitions. When some transitions are fused, their labels are concatenated. The newly generated transition (the fusion of some transitions) then is labeled with this concatenation. This kind of label concatenation immediately gives us the firing count of each fused transition, but it ignores the firing order. Every transition's label in the original Ada net will appear in some label(s) of the reduced Ada net before Rule 9 is applied. With these labels, it is easy to generate a firing count vector X during the search for the deadlock in a reachability graph. However, if Rule 9 is applied, some transition labels are lost. As we saw in Section 4, the removal of a transition by Rule 9 means that this transition has been fired. Thus we generate another m-vector, Y:

$$Y=(y_1, y_2, ..., y_m)^{T}, \quad \begin{array}{ll} y_i=1, & \text{if } t_i \text{ is removed, or a fused transition containing } t_i \text{ is removed, by Rule 9;} \\ y_i=0, & \text{otherwise.} \end{array}$$

By substituting the vectors X and Y for$V$ in Equation (5.1), we can obtain the full description of the deadlock marking in the original Ada net. That is,

$$M=A^{T}(X+Y)+M_0 \tag{5.2}$$

where A is the incidence matrix of the original Ada net and M_0 is the net's initial marking. The example of the next section illustrates this method.

If we record a state sequence (or transition sequence) leading to a deadlock state (either during or after reachability graph construction), the states of this net sequence will be defined with respect to a reduced Ada net. To provide an interpretation of this state sequence in terms of the program's execution sequence, we can apply the same basic method defined above (for interpreting a deadlock state), only now we would be applying the interpretation to intermediate states. Since the necessary transition information and the initial state are well defined, the method is still sound.

6. AN EXAMPLE OF DEADLOCK DETECTION AND IDENTIFICATION

Now we can explain how to reduce the net of Fig. 2.2 (with an initial marking where all begin-places are marked with one token) into that of Fig. 2.3, and how to use this new net to detect and identify deadlocks. First, by Rule 5, we can remove all the wait-places. Then by Rule 6 we remove the entry-ex-places, entry_ex_24_4, entry_ex_28_16, and entry_ex_15_6. By Rule 7, we remove the accept place, accept_15. Now we can first apply Rule 3 and then Rules 1 and 2 to many groups of transitions. Place accept_14 will be removed by Rule 4 when it becomes a parallel place of place ack_entry_Pump_5. We have the reduced Ada net shown in Fig. 6.1. Finally we apply Rule 9**.** The fully reduced Ada net is shown in Fig. 2.3 with a marking corresponding to the deadlock state. The reachability graph of the reduced Ada net of Fig. 2.3 is shown in Fig. 6.2; it is composed of 4 states in a chain. The deadlock state corresponds to the marking with a token in place "ack_entry_Customer_29". For comparison, we note that the reachability graph of the original Ada net has 78 states. The token allocation in Fig. 2.2 shows the deadlock marking in the original net, which can be represented by the following string:

"wait_ack_Pump_6, entry_ex_15_6, wait_ack_Operator_16, entry_ex_28_16, ack_entry_Customer_29, wait_ack_Customer_29" (6.1)

From this deadlock marking, it is easy to interpret the corresponding program state as a circular deadlock: task Customer is in a rendezvous with task Pump at entry Finish (statement 6) and task Pump is in a rendezvous with task Operator at entry Charge (statement 16), but task Operator has issued the entry call Customer.Change (statement 29). The number of marked wait-places gives the number of tasks that are in a waiting state.

Using the method outlined in Section 5, we can also identify the deadlock with the fully reduced Ada net. The path from the initial state to the deadlock state in the reachability graph of Fig. 6.2 identifies the fired transitions, T1 (t23,t24), T2 (t13,t25,t26,t4,t5,t32,t33,t21), T3 (t14,t6,t7,t15,t16,t28,t29). Thus, we have the following "firing count vector" with respect to the 31 transitions:

$$X = (0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1)^T.$$

The positions in the vector correspond to the following ordering of transitions: t1-t21, t23-t26, t28-t33 (for reasons unimportant to this paper, the original Ada net does not have transitions labeled t22 and t27).

The application of Rule 9 removes the fused transitions containing (t1,t2,t3), (t11,t12), and (t20,t21) (compare Fig. 6.1 and Fig. 2.3). This yields another vector as defined by the reduction process:

$$Y = (1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^T.$$

Now,

$$X+Y = (1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 2\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1)^T,$$

M_0(begin_2_Customer)=M_0(begin_11_Pump)=M_0(begin_21_Operator)=1, and $M_0(p)$=0 for every other place p. By applying Equation (5.2) defined in Section 5, we can obtain the marking vector that represents the same state as that of string (6.1).

7. A NET REDUCTION TOOL AND EXPERIMENTAL EVALUATION

We have implemented a prototype reduction tool that is compatible with our current research toolkit used for construction and evaluation of Ada nets [10]. As was mentioned in Section 1, the toolkit automates the construction of Ada nets and supports various forms of net-based analysis, including now, reduction-based analysis. The Ada nets are represented in a textual, production-rule form that is compatible with the format of Petri nets for input to the P-NUT system [31], a set of Petri net tools that allows for description of nets as well as reachability graph generation, analysis, and simulation. In the textual representation of a net, each transition is specified by a production rule -- the left-hand-side gives the transition label (enclosed between colons) followed by the names of the input places, and the right-hand-side gives the names of the output places.

Our net reduction tool, NRT, is implemented in C and runs on SunOS Unix. The major input to NRT is the Ada net produced by the FETS tool. The output of NRT is a reduced version of the input Ada net, in the same textual format. NRT is executed by the following command:

nrt [-r] [-n] [- [1] [2] [3] [4] [5] [6] [7] [8]]

where the -r option calls for the output to report on every reduction rule attempted and applied; the -n option calls for the output to list all intermediate nets produced during the reduction process; and the final option specifies the id numbers (as used in the earlier sections of this paper) of those reduction rules that are to be attempted and their order of consideration. Rule 9 is automatically attempted last. For example, -132 means that first Rule 1 is considered (i.e., it is repetitively applied until no further reductions are possible by this rule), then Rule 3 is considered, then Rule 2 is considered, and finally Rule 9 is considered. The typical use of NRT is to provide no id option, in which case the rules are applied in the order defined by the reduction process discussed earlier in Section 5.

To evaluate our net reduction method we performed a number of experiments with a variety of Ada tasking programs. The experiments provide quantitative measures of the effect of using our net reduction method (and associated tools) for deadlock analysis. For some experiments (in particular the dining philosophers example), these measures can be compared to existing data for a different tasking analysis approach and toolset. The comparison is intended to show the benefit of using a "reduced" concurrency model when performing deadlock analysis. Since our reduced Ada nets are intended to support only deadlock analysis, it is not meaningful to interpret the comparative data in a broader sense.

We discuss two types of experiments using our NRT tool. The first experiments are to evaluate our idea of net reduction using different versions of two "benchmark" tasking programs: the gas station program[1] and the dining philosophers program. In these experiments, net reduction was followed by full reachability graph generation. As mentioned before, this gives an opportunity to see some comparison of the effectiveness of the reduction method itself for deadlock analysis in comparison to another reachability-oriented and automated Ada tasking analysis technique. Our second experiments evaluated the effectiveness of using our net reduction method in conjunction with some other existing state space reduction methods (i.e., methods that reduce the state space *during* state space

[1] The source versions we used were developed (and provided to us) by G. Avrunin for experimental work with the constrained expressions analysis method.

generation -- in contrast to our reduction method, which performs reductions on the net model prior to state space generation). For both types of experiments, we observed significant benefit from the net reduction method.

Net reduction on the example Ada nets resulted in a more than 95% reduction in the state space size (comparing original net state space with fully reduced net state space). Our analysis of these reduced Ada nets accurately reported the existence or nonexistence of deadlock for all the examples considered. But, recall that some of the reduction gain is due to the removal of atomic steps (transitions) that are not strictly necessary to model tasking semantics. We do not discuss further the amounts of state space reduction achieved in terms of original vs. reduced nets; the more significant measures are those related to resultant state space sizes (for comparison of reduced nets to other tasking models that rely on reachability analysis) and analysis time.

All experiments were run using our reduction-based analysis tools on a Sun Sparcstation 2 with 64MB of memory. For fully reduced nets, we give the number of unique states, the number of arcs in the reachability graph (which indicates the number of states generated), and the time to reduce the net plus generate the state space, in seconds as measured by user+system time in Unix. For all experiments, the time to do the reduction step was a small fraction of the time to generate the state space and always less than one second. Lack of space prevents us from listing source code for the examples studied, but these are all available from the authors.

7.1 Net Reduction with Full Reachability Graph Generation

7.1.1 The Gas Station Example

The gas station program simulates an automated gas station system consisting of an operator, pumps and customers. We consider both one-pump and two-pump versions of this system, and program-versions that do and do not have deadlock. In all cases, we used "unrolled" versions in which different customers call different entries in the operator task. The operator task uses boolean variables to keep track of which customers have prepayed for gas and this control is automatically modeled in our Ada nets. Our current program-to-net translator does not yet handle the more general use of counter variables. Our reachability graph generator tool is intelligent enough to consider transition sequences that correspond to boolean variable evaluations as atomic actions -- thus states corresponding to interleavings of these evaluation steps are not stored (they cannot correspond to a deadlock state). This issue is discussed in some detail in [23]. The deadlock cases were created by having the operator "improperly" handle a flag variable that indicates when a pump is busy. The result is that after a customer finishes pumping and paying, the operator still thinks the pump is busy and so it does not allow any customer to get access to the pump. The experiments reported in [23] introduced deadlocks in a slightly different way. Table 1 shows some sample data based on varying the number of customers, the number of pumps, and the potential for deadlock.

No. of Customers, Pumps, Tasks	Deadlock?	Fully Reduced Ada Net Reachability Graph (states, arcs, time)
3, 1, 5	No	170, 300, 1
5, 1, 7	No	1622, 3180, 2
9, 1, 11	No	74222, 156636, 237
10, 1, 12	No	180204, 383960, 707
3, 1, 5	Yes	214, 364, 1
5, 1, 7	Yes	2150, 4156, 2
9, 1, 11	Yes	103150, 216796, 288
3, 2, 6	No	2723, 7050, 2
5, 2, 8	No	212015, 777590, 344
5, 2, 8	Yes	198023, 720227, 331

Table 1. Experiments for the Gas Station Program

7.1.2 The Dining Philosophers Example

For the well-known dining philosophers problem we experimented with a few different versions that have become "standard" examples for a number of different concurrency analysis research projects. These examples are especially relevant to this paper since they allow some quantitative comparison of our approach with another state-space oriented analysis technique, the modified (optimized) version of the Task Interaction Concurrent Graphs (TICG) method [32,33]. We assume that the reader is familiar with the dining philosopher problem. The program simulating *n* philosophers consists of 2*n* Ada tasks, one for each philosopher and one for each fork. Four versions of this program are discussed in [32,33].

Version 1 is the classic dining philosopher problem without any deadlock avoidance. Table 2 shows some comparisons of the state space sizes for our fully reduced Ada net model and the published data for the modified TICG models [33]. Note that with our fully reduced Ada net for this problem, the number of states is 2^P, where P is the number of philosopher tasks. For the modified TICG model, the number of states is 3^P-1 -- but remember, the TICG model is a more general model in terms of preserving tasking properties (beyond deadlock potential).

In Version 2 each philosopher picks up its lower-numbered fork first. Similar to the situation reported in [32], the state space for Version 2 is approximately equal in size to that of Version 1. In Version 3, a dining room butler-task is added, which restricts the number of room occupants to be less than *n* by checking the value of a counting variable. Thus no deadlock should happen. However, a deadlock was superfluously reported for this version by pure static analysis because the counting variable was not modeled. Although static analysis in this case reveals a "false deadlock," the numerical results with respect to state sizes can still be compared for the reduced Ada net model and the modified TICG model since both models ignore the counting variable. Table 3 shows the comparison of state space sizes for Version 3. To provide a further check on the reduction method we manually added some nodes to the automatically generated (original) net models to simulate the control enforced by the butler's counting variable. For example, if the counter could take on the values from 0 to 10, we would add 11 places to the Ada net, each modeling a unique value for the counter. As expected, the experiments with these nets showed that all superfluous deadlocks were suppressed in these modified models. The state space sizes of the reduced nets obtained from the modified net model are shown in the rightmost column of Table 3.

No. of Phils, Tasks	Reachable States		
	Fully Reduced Ada Net (states, arcs, time)	Modified TICG [33] (states, arcs, time)	Difference in States (Approx. %)
3, 6	8, 18, 1	26, 51, 2	69
5, 10	32, 120, 1	242, 805, 2	87
7, 14	128, 672, 1	2186, 10199, 5	94
8, 16	256, 1536, 1	6560, 34984, 26	96
9, 18	512, 3456, 1	19682, 118089, 337	97
10, 20	1024, 7680, 1	59048, 393650, 8090	98
14, 28	16384, 172032, 8	not available	–
18, 36	262144, 3538944, 175	not available	–
20, 40	1048576, 15728640, 894	not available	–

Table 2. The Dining Philosophers Problem: Version 1

No. of Phils, Tasks	Reachable States			
	Fully Reduced Ada Net (Ignoring Variables) (states, arcs, time)	Modified TICG [33] (Ignoring Variables) (states, arcs, time)	Difference (%)	Fully Reduced Ada Net (Simulating Counting Variable) (states, arcs, time)
2, 5	16, 30, 1	28, 50, 1	43	5, 6, 1
3, 7	64, 180, 1	154, 411, 2	58	37, 78, 1
4, 9	256, 960, 1	832, 2964, 3	69	175, 524, 1
5, 11	1024, 4800, 1	4474, 19925, 7	77	781, 2980, 1
6, 13	4096, 23040, 1	24040, 128478, 56	83	3367, 15438, 1
8, 17	65536, 491520, 31	not available	–	58975, 354616, 31
9, 19	262144, 2211840, 146	not available	–	242461, 1622016, 139
10, 21	1048576, 9830400, 656	not available	–	989527, 7274662, 711

Table 3. The Dining Philosophers Problem with Butler: Version 3

Young, et al [32] provided another program version, Version 4, in which the dining room butler-task simulates a counting variable by unrolling the selective-wait statement. The idea was to simulate the variable values by levels of nesting of selective-wait statements, thereby avoiding spurious deadlocks in the state space analysis. To our surprise, our analysis tool reported deadlock states for Version 4 when the number of philosophers was more than two. After some investigation we realized that the program itself was in error and could indeed deadlock since the method for unrolling the butler's selective wait (provided in [32]) may allow n philosophers to occupy the room simultaneously. In [33] a program with a corrected butler-task is given. Our analysis tools properly detected no deadlocks in this "corrected" Version 4 program. Table 4 summarizes some results using the TICG data reported in [33] for the corrected, unrolled butler task. As expected, the number of fully reduced net states using the unrolled butler is the same as the number of net states given in Table 3 for the net model that simulates the butler's counting variable.

The above comparisons show a significant advantage in state space sizes for reduced nets in comparison to the TICG model. This indicates that our philosophy of using a specific model for a specific problem is a useful strategy in analysis of inherently complex problems. Here we see the benefit of using a problem-specific reduced model, not in comparison just to our own unreduced Ada net models, but to an independently developed concurrency analysis model.

No. of Phils, Tasks	Number of Reachable States: Fully Reduced Ada Net Unrolled Butler (states, arcs, time)	Modified TICG [33] (states, arcs, time)	Difference (%)
2, 5	5, 6, 1	11, 11, 2	55
4, 9	175, 564, 1	511, 1543, 3	66
6, 13	3367, 17694, 1	18263, 90155, 47	82
7, 15	14197, 88809, 7	104679, 625873, 1099	86
8, 17	58975, 427368, 33	not available	–
9, 19	242461, 1995327, 165	not available	–
10, 21	989527, 9108690, 910	not available	–

Table 4. The Dining Philosophers Problem with Unrolled Butler: Version 4

7.1.3 The Waveform Generator Example

To see how the reduction process works on a "real world" example, we obtained an example Ada program that was developed to solve a real problem; it was not designed for the purpose of testing any particular concurrency analysis method. For this example, we did have to perform some hand modifications of the Ada source in order to get a proper net model of the program. This was primarily due to the presence of some Ada specific constructs -- like packages -- that our translator is not designed to handle. The waveform generator program is a 1200 line Ada simulation of electric equipment that generates various waveforms. This is one of the programs used in [23] for experimentally studying various state space reduction methods, including net reduction. As indicated in [23], two versions of the program were created. For the two versions, the numbers of states and arcs in the reachability graphs corresponding to the original Ada nets are 10,996 states with 32,332 arcs, and 4,827 states with 14,250 arcs. For the fully reduced Ada net, the corresponding numbers of states and arcs reduces to 78 states with 326 arcs, and 31 states with 242 arcs. The state space generation time for each of these cases was 2 seconds. It should be noted that the numbers of states and arcs quoted here for the original Ada net are slightly smaller than the "full" state space numbers given in [23]. This is due to the fact that we now consider a particular type of original net reduction related to selective-wait structures to be part of the net *generation* process, not part of the net *reduction* process used for deadlock detection (as was done in [23]). Of course, either way, the reduction process provides significant benefit in reducing the size of the state space.

7.2 Net Reduction with State Space Reduction

Our second series of experiments looked at the effectiveness of using our net reduction method in support of some recently proposed reduced state space generation methods. In particular, we considered three reduced state space generation methods: stubborn sets of Petri net transitions [20], partial orders of Petri net transitions [21], and net symmetry [22]. The basis of our experiments was to generate original Ada nets from a set of Ada programs and then to apply each of the reduced state space generation methods to these Ada nets. Then we applied our net reductions to the original Ada nets and again applied each of the reduced state space generation methods (this time to the reduced nets). The examples we considered included those from the first experiments (versions of the gas station program and versions of the dining philosophers program) as well as versions of a readers/writers program. As an example, for the traditional dining philosophers problem with n philosophers, the combination of net reduction and stubborn sets generates ($n^2 - n + 2$) states and the combination of net reduction and symmetry generates ($n + 1$) states. What we observed is that for all cases (i.e., for all programs and for all reduced state space generation method) there was significant benefit from using the net reduction method prior to applying reduced state space generation -- in effect, our reduction step increases the utility of the state generation methods when they are applied to deadlock analysis in Ada tasking. Full details on experiments using reduction with reduced state space generation methods can be found in [23,34]. It should be noted that the reduction method used in [34] was a bit less powerful than the method described in this paper. In particular, the rule 6 defined in this paper is a more general rule than that used in the experiments reported in [34]. This is why, for example, the state space numbers reported for the dining philosophers problem are better (i.e., smaller) in this paper in comparison to the numbers reported in [34].

The use of net reduction and reduced state space generation appears to be a very powerful combination for deadlock analysis in Ada tasking. We can make the following observations based on a comparison of our experiments with experiments reported for two other fully automated analysis tools that also avoid full state space generation -- the constrained expressions toolset [8,16] and the PAL tools [15]. Both of these approaches have been shown to be capable of handling the dining philosophers problem with more than 100 philosopher tasks. For the dining philosophers problem (without a butler task), a combination of net reduction followed by reduced state space generation is also capable of handling problems with more than 100 philosophers. In fact, as stated before, one combination provides a state space size that grows linearly with the number of philosophers [23]. To our knowledge, the only other work that reports a more favorable outcome for analysis of this problem is the work of Valmari and Tienari [35] using a semantic model called the Chaos-Free Failures-Divergences (CFFD) model. Their approach is compositional, in a way similar to that of the PAL system [15]. By using a combination of compositional analysis and induction, Valmari and Tienari are able to analyze the dining philosophers system by generation of a constant number of states. Although this work is important, the technique's effectiveness on other examples is not yet clear and the results cited require a manual step in recognizing the use of induction. Also, it is unclear whether the type of inductive step leading to constant state space size for the dining philosophers example can be easily automated for general usage. In contrast, our analysis work using net reduction and state space reduction is fully automated.

In comparing analysis time for the constrained expression and PAL tools (which is about all that can be compared for these methods), we find that our analysis is as effective, or more effective, than the other two tools, depending on which state space reduction method we use following net reduction. At this time, no single method has established definite superiority in terms of automated deadlock analysis. Continued development and comparative studies are clearly necessary.

8. CONCLUSION

It is commonly accepted that a major impediment to static analysis of concurrent programs is the complexity associated with producing various forms of state space representations. We have presented a method for dealing with this matter by optimizing the model, which is based on two key ingredients: focusing on deadlock as a specific analysis issue, and adapting an existing model-reduction technique, Petri net reduction. This result is part of our ongoing research into using Petri nets to support automated analysis of Ada tasking. We are motivated by the belief that there is much to be gained by basing Ada tasking analysis research on a model that is both theoretically mature (and also continues to be widely and actively studied) and is already supported by many available tools. By combining Petri net theory and knowledge of Ada tasking semantics, we derived some domain-specific, and thus efficient, reduction rules for Petri net models of Ada tasking.

Generally it is error-prone to construct a model and optimize it manually for program analysis. It is critical that the model produced reflect the program's true behavior, not some behavior that a programmer (or modeler) desires or thinks is true for the program in question. Furthermore, model optimization can benefit from an understanding of the analysis issues that are germane to the program. Compiler techniques can support the automation of building correct models. But the combination of model building and model optimization adds another dimension of difficulty, considering that the issues to be analyzed may vary widely. We apply a two-phase methodology to fully automate modeling and optimization: first deriving a semantically rich model independent of any specific analysis issue, and then manipulating this model with algorithms that are designed for the specific analysis issue of concern. We have observed that the reduced Ada net models are a promising aid to deadlock analysis of concurrent Ada programs, especially when used in combination with other reduced state space generation methods.

ACKNOWLEDGMENTS

We thank Mr. M. Goto for his help with development of the net reduction tool and U. Buy for helpful suggestions on many aspects of this paper. We also thank M. Young for providing us with a copy of reference 33. Finally, we thank the referees for their very valuable comments that have greatly improved the paper both in terms of technical content and presentation.

REFERENCES

[1] J. Reif and S. A. Smolka, "The Complexity of Reachability in Distributed Communicating Processes," *Acta Informatica*, Vol. 25, 1988, pp. 333-354.

[2] T. Rauchle and S. Toueg, "Exposure to Deadlock for Communicating Processes is Hard to Detect," *Information Processing Letters*, Vol. 21, 1978, pp. 63-68.

[3] R. N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica*, 19, 1983, pp. 57-84.

[4] R. N. Taylor, "A General-Purpose Algorithm For Analyzing Concurrent Programs," *Communications of the ACM*, Vol. 5, No. 5, May 1983, pp. 362-376.

[5] D. L. Long and L. A. Clarke, "Task Interaction Graphs For Concurrency Analysis," *Proceedings of the 11th Int. Conference on Software Engineering*, Pittsburgh, PA., May 1989, pp. 44-52.

[6] S. M. Shatz and W. K. Cheng, "A Petri Net Framework For Automated Static Analysis of Ada Tasking Behavior," *Journal of Systems and Software,* 8, Dec. 1988, pp. 343-359.

[7] G. Avrunin and J. Wileden, "Describing and Analyzing Distributed Software System Designs," *ACM Trans. on Prog. Lang. and Systems*, Vol. 7, No. 3, July, 1985, pp. 380-403.

[8] G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileden, "Automated Analysis of Concurrent Systems With the Constrained Expression Toolset," *IEEE Trans. on Software Engineering*, Vol. 17, No. 11, Nov. 1991, pp. 1204-1222.

[9] T. Murata, B. Shenker and S. M. Shatz, "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Trans. on Software Engineering*, Vol. 15, No. 3, March 1989, pp. 314-326.

[10] S. M. Shatz, K. Mai, C. Black, and S. Tu, "Design and Implementation of a Petri Net-Based Toolkit for Ada Tasking Analysis," *IEEE Trans. on Parallel and Distributed System*, Vol. 1, No. 4, Oct. 1990, pp. 424-441.

[11] C. Black, S. M. Shatz, and S. Upp, "TQL: A Tasking Query Language for Concurrent Program Analysis," *Proceedings of the 12th Int. Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992, pp. 382-389.

[12] A. Blakemore and G. Schebella, "Tools for Analyzing Dynamic Properties of System and Software Designs," *Proc. of the IFIP XI Congress*, San Francisco, Ca., 1989.

[13] G. M. Karam and R. J. Buhr, "Starvation and Critical Race Analysis for Ada," *IEEE Trans. Software Engineering*, Vol. 16, No. 8, August 1990, pp. 829-843.

[14] W. J. Yeh and M. Young, "Compositional Reachability Analysis Using Process Algebra," *Proc. of the 1991 Symp. on Software Testing, Analysis, and Verif.* (TAV), Victoria, B. C., Canada, Oct. 1991, pp. 49-59.

[15] W. J. Yeh, *Controlling State Explosion in Reachability Analysis*, PhD Dissertation, Dept. of Computer Science, Purdue University, Aug. 1993.

[16] J. C. Corbett, "Identical Tasks and Counter Variables in an Integer Programming-Based Approach to Verification," *Proceedings of the 7th Int. ACM Workshop on Software Specification and Design*, Dec. 1993, pp. 100-109.

[17] M. B. Dwyer, L. A. Clarke, and K. A. Nies, "A Compact Petri Net Representation for Concurrent Programs," *Proceedings of the 17th International Conf. on Software Engineering*, April, 1995, pp. 147-157.

[18] R. Enders, T. Filkorn, and D. Taubner, "Generating BDDs for Symbolic Model Checking in CCS," *Proc. of the Workshop on Computer Aided Verification (CAV)*, 1991, pp. 263-278.

[19] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, 1990, pp. 428-439.

[20] A. Valmari, "A Stubborn Attack on State Space Explosion," *Proc. Workshop on Computer-Aided Verification*, LNCS 531, Springer-Verlag, New York, pp. 156-165, 1991. Also in *Formal Methods in System Design*, Kluwer Academic Publishers, Vol. 1, 1992, pp. 297-322.

[21] P. Godefroid and P. Wolper, "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties," *Formal Methods in System Design*, Kluwer Academic Publishers, Vol. 2, No. 2, April 1993, pp. 149-164.

[22] P. H. Starke, "Reachability Analysis of Petri Nets Using Symmetries," *Systems Analysis Modelling and Simulation*, Vol. 8, pp. 293-303, 1991.

[23] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, "Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada," *ACM Trans. on Software Eng. Methodology*, Vol. 3, No. 4, Oct. 1994, pp. 340-380.

[24] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, N. J., 1981.

[25] K. Gostelow, V. Cerf, G. Estrin and S. Volansky, "Proper Termination of Flow-of Control in Programs Involving Concurrent Processes," *Proceedings of the ACM Annual Conf.*, Vol. II, Boston, Aug. 1972, pp. 742-754.

[26] G. Berthelot, "Checking Properties of Nets Using Transformations," G. Rozenberg (ed), *Advances in Petri Nets 1985*, LNCS 222, N.Y., Springer-Verlag, 1987, pp. 19-40.

[27] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, April 1989, pp. 541-580.

[28] United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Feb. 17, 1983.

[29] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol. 2, No. 2, March, 1985, pp. 47-57.

[30] S. M. Shatz, S. Tu, T. Murata, and S. Duri, "Theory and Application of Net Reduction for Ada Deadlock Analysis," Tech. Report, Dept. of EECS, University of Illinois at Chicago, Oct. 1993.

[31] E. T. Morgan and R. Razouk, "Interactive State-Space Analysis of Concurrent Systems," *IEEE Trans. on Software Engineering*, Vol 13, No. 10, Oct. 1987, pp. 1080-1091.

[32] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck, "Integrated Concurrency Analysis in a Software Development Environment," *Proceedings of ACM SIGSOFT'89, TAV 3*, Key West, Florida, Dec. 1989, pp. 200-209.

[33] M. Young, R. Taylor, D. Levine, K. Forester, and D. Brodbeck, "A Concurrency Analysis Tool Suite: Rationale, Design, and Preliminary Experience," *Technical Report, TR-128-P*, Software Engineering Research Center, Purdue University, Oct. 1992.

[34] S. Duri, U. Buy, R. Devarapalli, and S. Shatz, "Using State Space Reduction Methods for Deadlock Analysis in Ada Tasking," *Int. Symp. on Software Testing and Analysis*, Boston, Mass., June 1993, pp. 51-60 (Reprinted in *ACM Software Engineering Notes*, Vol. 18, No. 3, 1993).

[35] A. Valmari and M. Tienari, "An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm," *Proceedings of the 11th IFIP Working Group 6.1 Symposium on Protocol Specification, Testing and Verification*, Stockholm, Sweden, North-Holland Pub., June 1991, pp. 3-18.

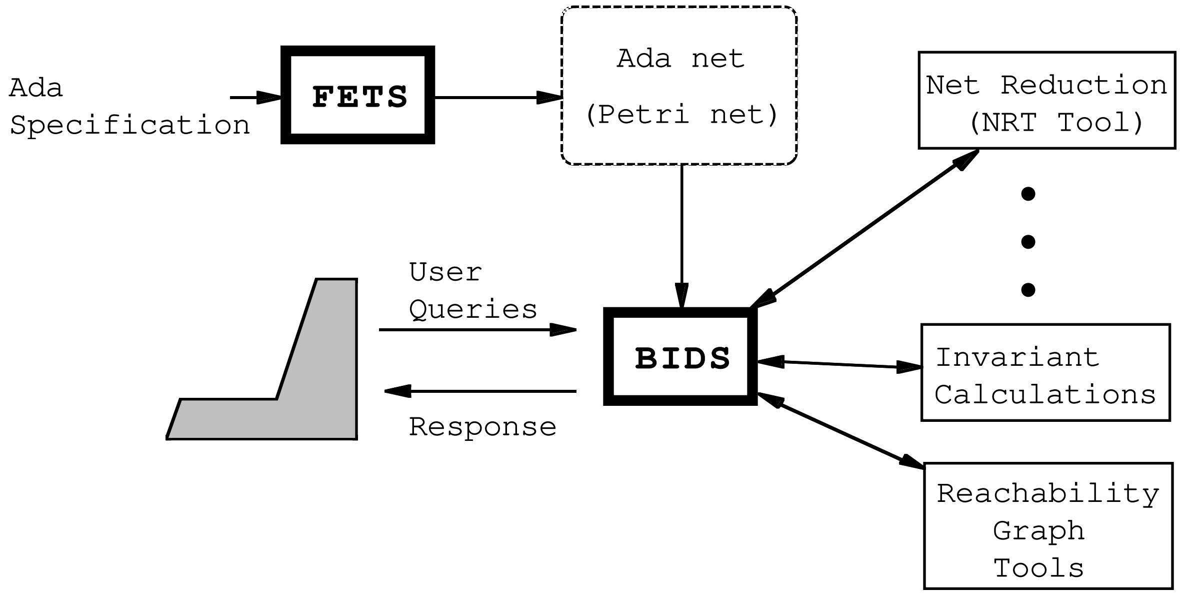


Fig 1.1 The TOTAL Toolkit Architecture

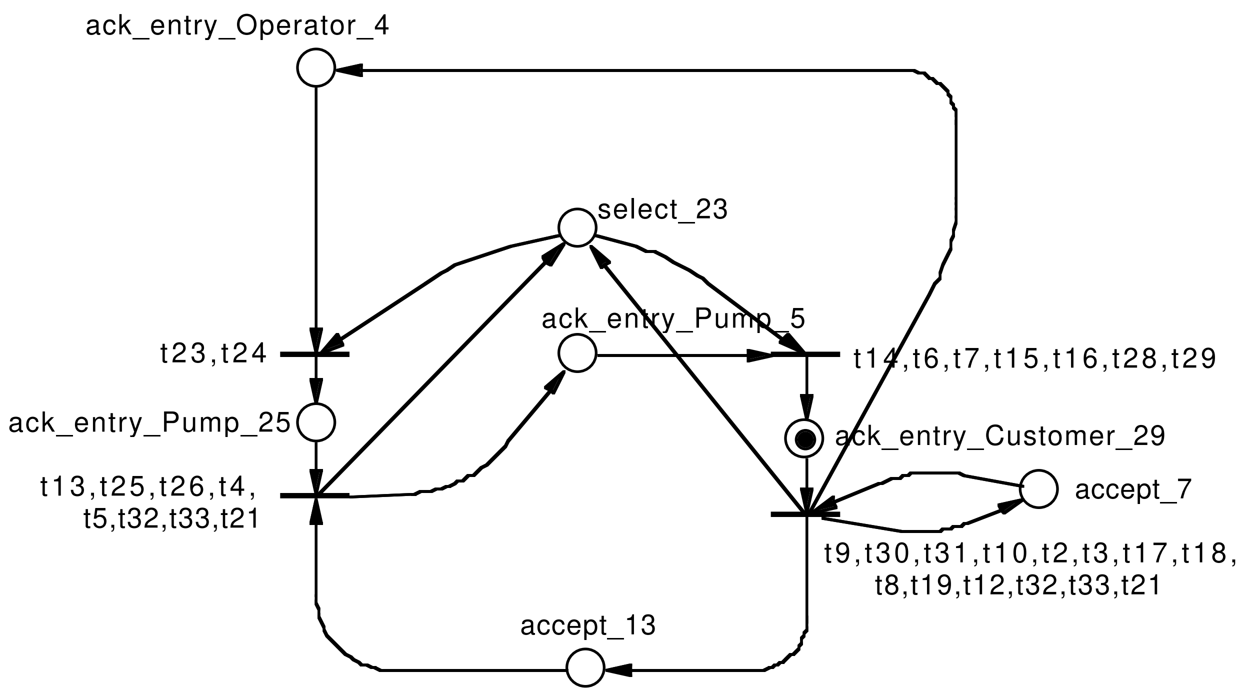
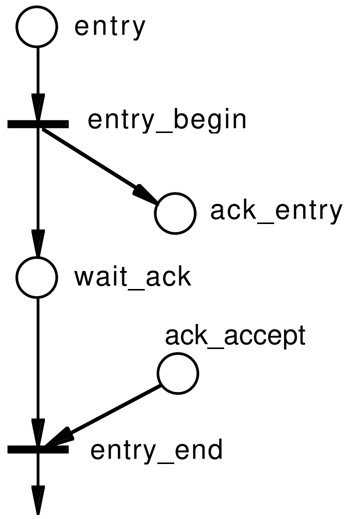
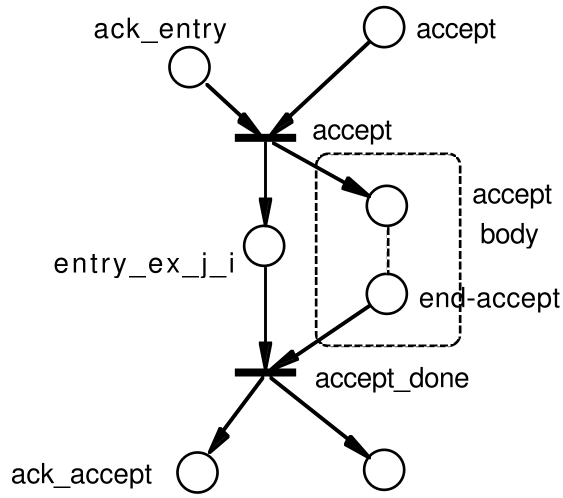


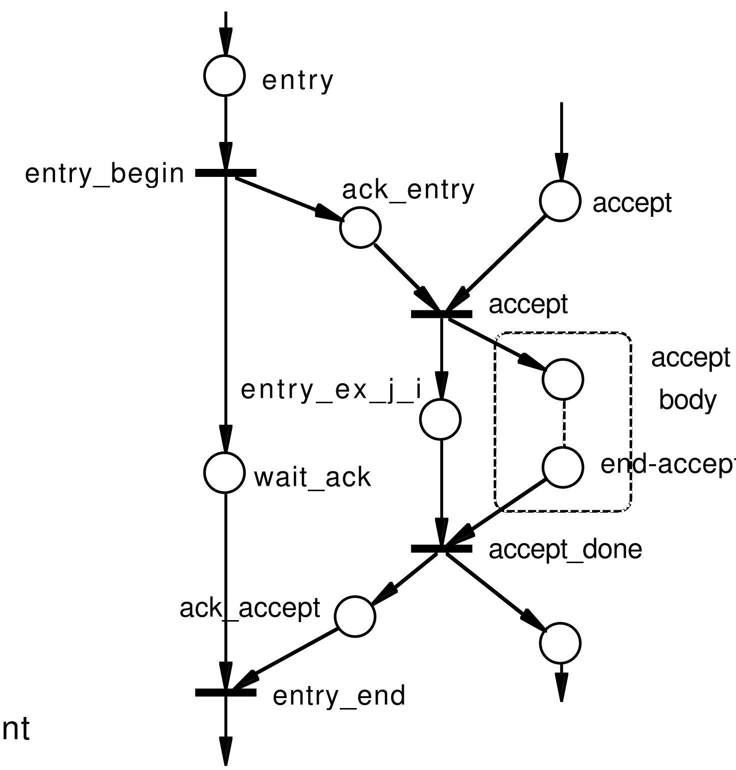
Fig 2.3 Fully Reduced Net Model of the Gas Station Program at Deadlock



(a) An Entry Call



(b) An Accept with
Accept Body



Note for Labels:

i is the entry call statement number; j is the accept statement number.

(c) A Rendezvous

Fig 2.1 Net Structures for Entry Call Statements and Accept Statements

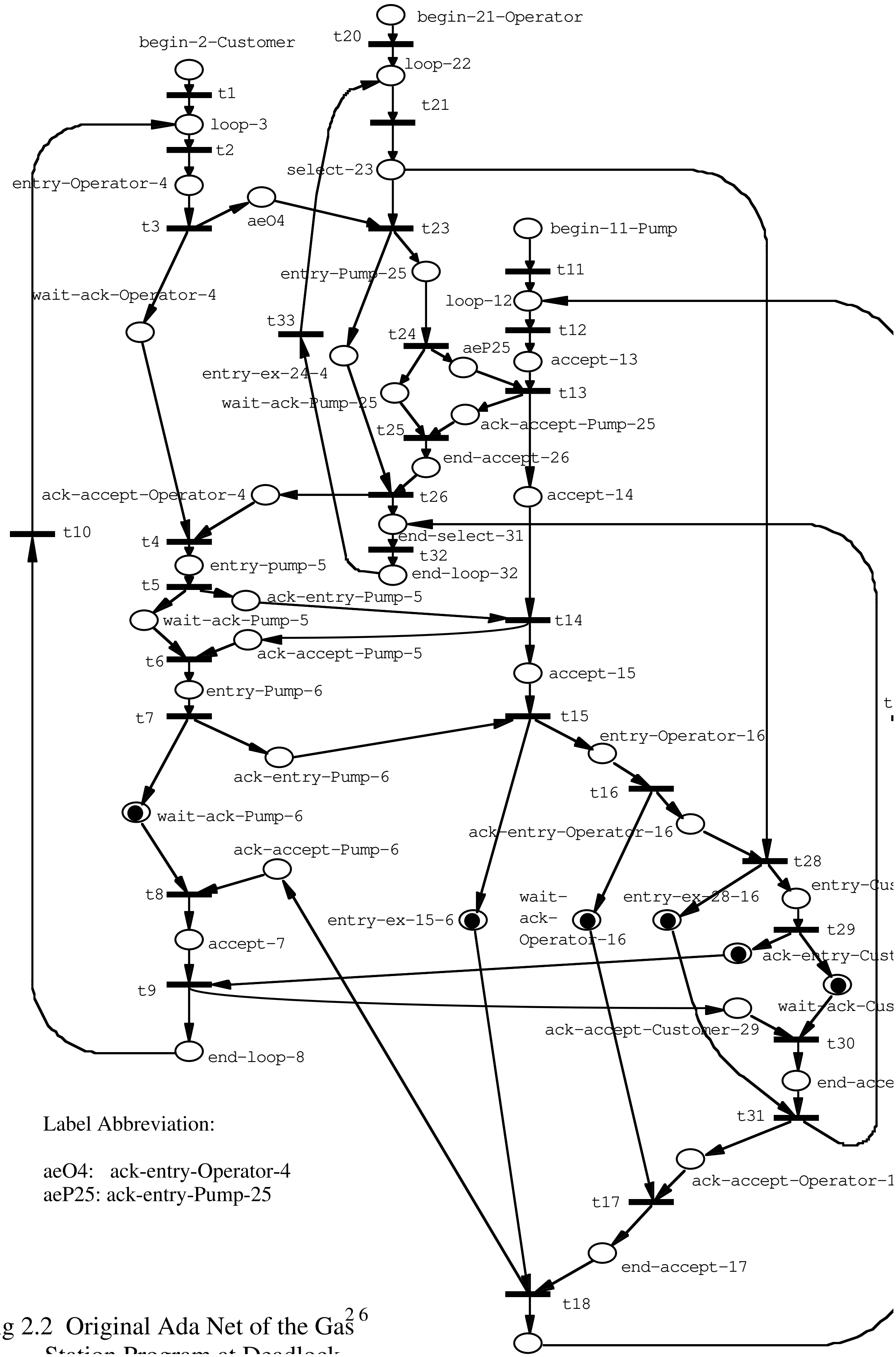


Fig 2.2 Original Ada Net of the Gas
Station Program at Deadlock

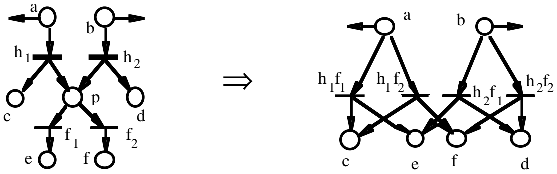


Fig 3.1 Post-Fusion of Transitions

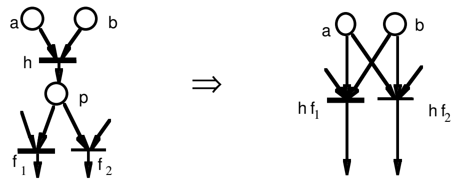


Fig 3.2 Pre-Fusion of Transitions

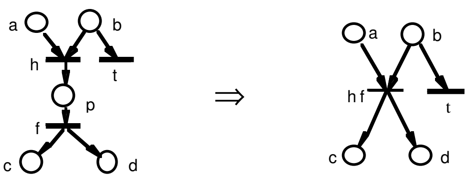


Fig 3.3 Serial-Fusion of Transitions

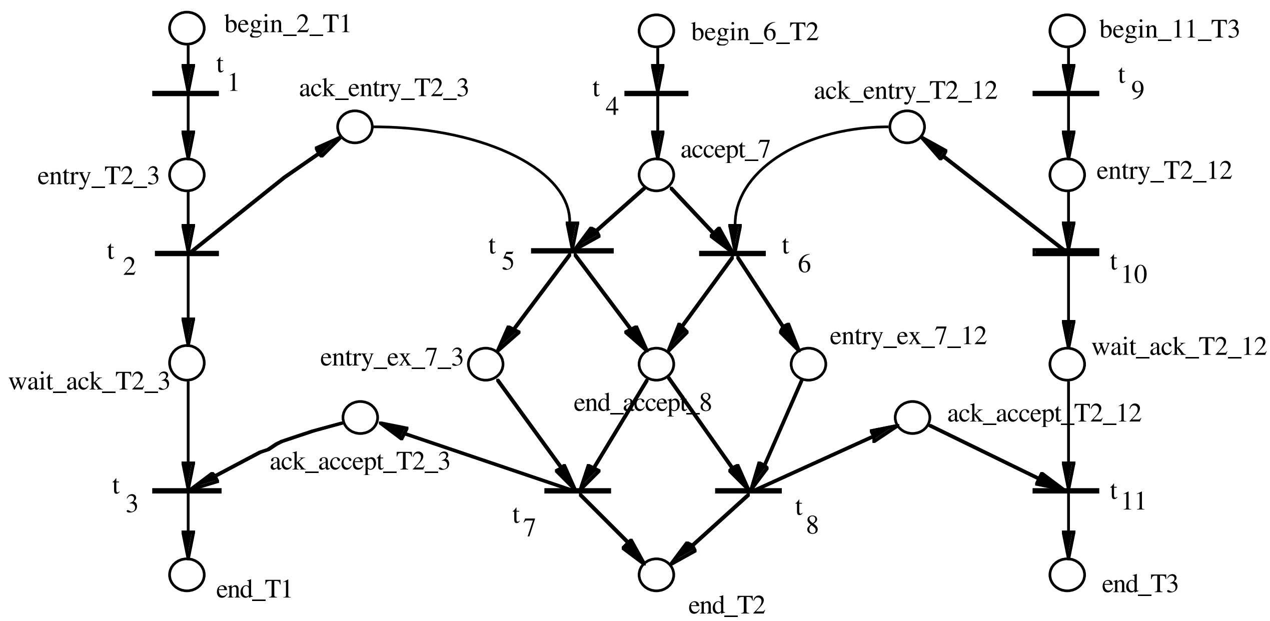


Fig 4.1 (a) Original Ada Net of Example 4.1

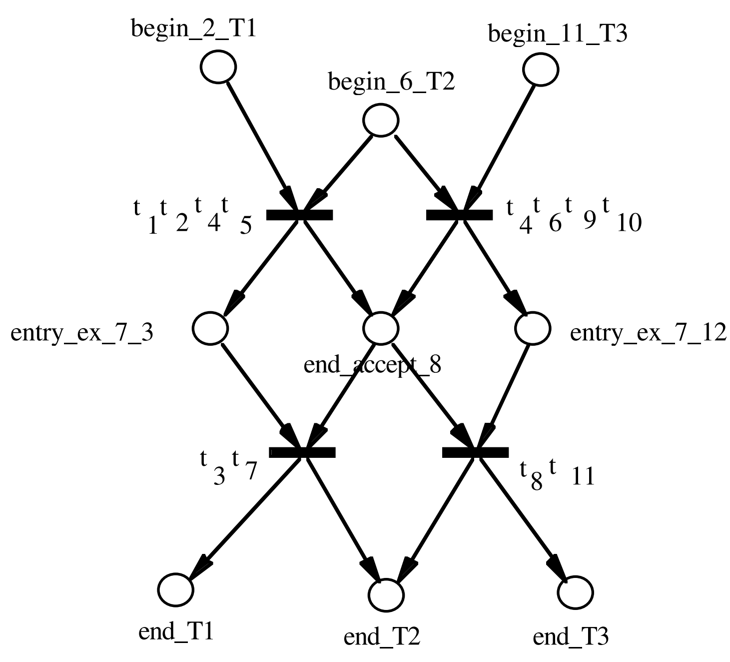


Fig 4.1 (b) The Reduced Net of Fig 4.1 (a)

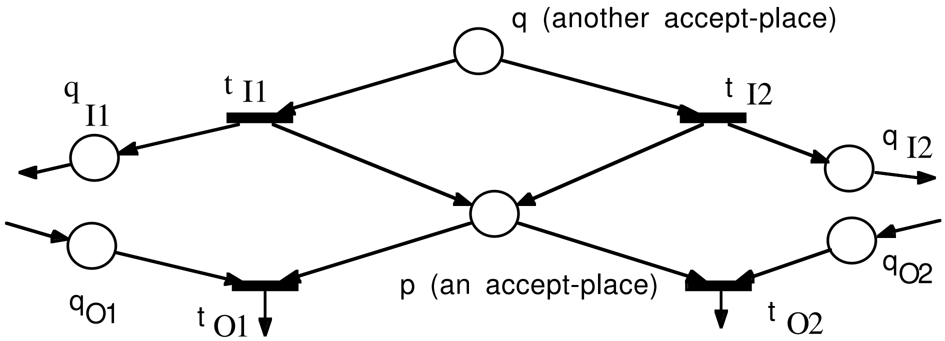


Fig 4.2 Redundancy of an Accept-Place

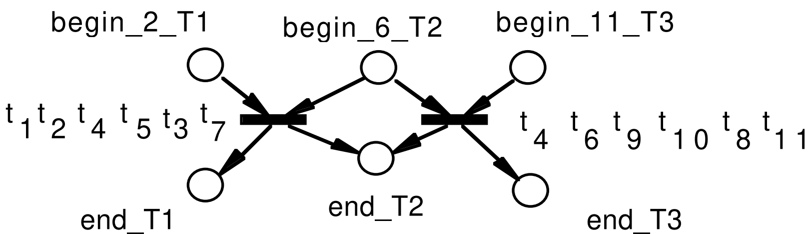


Fig 4.3 Further Reduced Net of Example 4.1

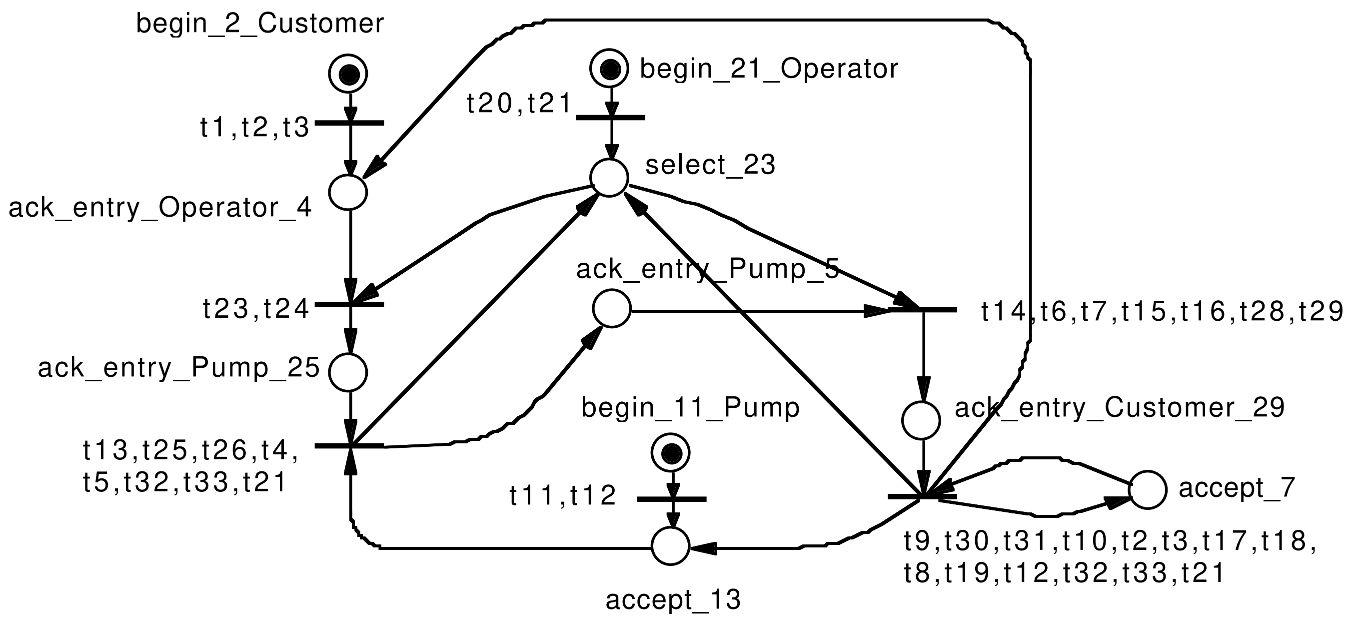
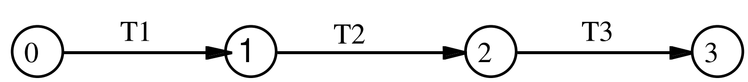


Fig 6.1 A Reduced Net Model From Fig 2.2



STATES:
0: select_23,ack_entry_Operator_4,
 accept_13
1: ack_entry_Pump_25,accept_13
2: ack_entry_Pump_5,select_23
3: ack_entry_Customer_29

TRANSITIONS:
T1: t23,t24
T2: t13,t25,t26,t4,t5,t32,t33,t21
T3: t14,t6,t7,t16,t28,t29

Fig 6.2 The Reachability Graph of the Reduced Net (Fig 2.3)